UNIVERSITY OF TWENTE. ASTRON Netherlands Institute for Radio Astronomy

Streaming Array Modelling and Generating VHDL with Python

R.C.M. (Reinier) van der Walle

Master Thesis

Committee:

Dr. Ir. A.B.J. Kokkeler

Ir. E. Molenkamp

Dr. Ir. J.F. Broenink

Ing. D. van der Schuur

November 2019

Master's programme: Electrical Engineering
Specialization: Dependable Integrated Systems
Research group: Computer Architecture for Embedded Systems

Abstract

This work presents the first working prototype of Streaming Array Modelling for Python (SamPy), a proposed concept to describe streaming systems in the commonly used programming language Python, as well as the development of a VHDL generator to generate VHDL code from a described streaming data system. The objective of this work is to overcome several problems introduced when working on large and complex streaming data systems. This includes the fact that communicating about such a system by the people involved can be rather difficult. SamPy enables a clear and organized way of describing these large systems, reducing the communication difficulty. Another issue is that system properties and stream definition are not directly related when using conventional methods. SamPy solves this by enabling programming in a stream-oriented way. The last problem addressed in this work is that the development of interfaces is very time consuming. This is improved by the realisation of the VHDL generator. This generator interprets a detailed SamPy system description and can generate complete hardware descriptions of pipelines to be implemented on FPGAs.

After development of both SamPy and the VHDL generator, several use cases are tested to verify this prototype. The first use case is an implementation of unpacking a video stream from a standard high-speed input. The whole process is described in SamPy with four interface definitions. The system description is automatically evaluated by SamPy, then it is passed to the VHDL generator which creates all the VHDL files needed in order to directly simulate the system. A more complex use case is realised that integrates not only the generated functions but also a user implemented function. To prove that SamPy can be used for describing a more abstract system, a third use case is implemented that describes a completely different type of streaming process based on an automated factory. Such a system is, of course, not generatable. However, SamPy is still able to evaluate certain characteristics of the system.

Table of Contents

List of Acronyms

BNF	Backus-Naur form
CNL	Controlled Natural Language
CPU	Central Processing Unit
CQL	Continuous Query Language
DSL	Domain Specific Language
FPGA	Field Programmable Gate Array
GPU	Graphical Processing Unit
HDL	Hardware Description Language
IP	Intellectual Property
META	Middleware for Events, Transactions, and Analytics
OpenCL	Open Computing Language
RDF	Resource Description Framework
SPL	Streams Processing Language
SQL	Structured Query Language
Verilog	A Hardware Description Language
VHDL	Very high speed integrated circuit Hardware Description Language
XML	Extensible Markup Language

1 Introduction

Streaming processes can be found in many places like traffic, automated factories and stream processing of data. This project aims to develop a language for describing streaming processes. During this project, the focus will be on stream processing of data. Streaming data processing applications represent an important class of high-performance computations. Defined by the processing of sequences of data, streaming applications appear most commonly in the context of audio, video, and digital signal processing, though also in networking, encryption, and other areas.

1.1 Project purpose

The purpose of this project is mainly to overcome three problems that are introduced when working on large and complex streaming data systems. These problems are:

1. Communicating about a large streaming data system is very difficult.
2. System properties and stream definitions are often not directly related.
3. Interface development is very time consuming.

The aim is to develop a system modelling language that can be used to describe the physical and timing properties of interfaces in a streaming system. This description should be detailed enough to be able to derive functions that perform data format alterations, such as serializing and de-serializing. During this project, the focus is on generating VHDL for these functions to implement it on an FPGA platform. The modelling language should also be abstract enough to serve as a communication tool between competence groups.

The problem statement and the objective of this project are explained in more detail in Chapter 3. Chapter 2 discusses the related work and answers the question: What is a suitable approach to describe streaming systems in order to reduce development effort?

1.2 Interested parties

ASTRON is the Netherlands Institute for Radio Astronomy. In addition to doing fundamental astronomy research, ASTRON also designs and manages some of the world's leading radio telescopes. These telescopes generate a tremendous amount of data which is too much to interpret without data reduction. This is dealt with by utilizing stream processing. A streaming array modelling language can be very helpful for communicating the system between the different competence groups. Also, being able to generate a VHDL pipeline for FPGA implementation from a high-level description can potentially improve the development time.

Other interested parties may include system architects, design engineers, and other people that are involved in the design of a streaming system.

1.3 Report organisation

The remainder of this report is organized as follows. Chapter 2 presents the related work. Chapter 3 discusses the problem statement and objective. The working of the developed solution is explained in Chapters 4 and 5. The results of the implementation of the use cases are presented in Chapter 6. A conclusion is drawn in Chapter 7. Chapter 8 discusses several future recommendations. In addition to these chapters, several appendices are included to explain extra details of this work.

2 Related Work

In preparation for this project, three subjects have been investigated to find related work. First, several stream processing languages and abstractions are discussed in Section 2.1. Secondly, a language for describing hardware that was already proposed in 1998 is discussed in Section 2.2. Section 2.3 describes examples of streaming application generators. In addition to these three subjects, a rough concept of a stream modelling language is proposed in Section 2.4.

2.1 Stream Processing Languages and Abstractions

An article by IBM Research [1] surveys a total of eight approaches for stream processing languages and abstractions. The findings are grouped into three categories, these are streaming languages driven by the data model, the execution model, or by the target user.

Data-model driven streaming language

A data-model driven streaming language is based on the attitude that data is most central since streams are data in motion. According to that attitude, the language should be built around a data model. The relational data model for database systems has inspired streaming dialects of the SQL database language. An example is the CQL language, it complements standard relational operators of SQL with operators to transform streams into relations and vice versa [2]. XML is another data model which has inspired XML-based streaming languages. A rich ecosystem of XML tools and standards together with the fact that XML documents are self-describing is taken advantage of by these languages. The Resource Description Framework (RDF) is another data model which is used as an inspiration for a streaming language.

Execution-model driven streaming language

An execution-model driven streaming language focuses on the execution model for processing the dataflows efficiently. This can be done by enforcing timing constraints or exploiting distributed hardware. An example is Lustre, a language for high-level specifications in the form of stream functions specifying variable values at each step or instant of a synchronous dataflow program [3]. A more recent example of an execution-model driven streaming language created for describing big-data streams is SPL [4]. Streaming big data is a lot of data that streams into a system at a fast rate. Such as system must deal with diverse data and functionality, and with uncertainty. SPL lets users specify an explicit stream graph that can be easily distributed with minimal synchronization and is extensible by operators in widely adopted general-purpose languages.

Target-user driven streaming language

Target-user driven streaming languages focus on enabling the end user to develop streaming applications in high-level or familiar abstractions such as complex events, spreadsheets, or natural language. Reaching the maximum number of target users might be achieved by a streaming language based on natural language. However, a controlled natural language (CNL) is a better choice since natural language is ambiguous. META is an example of a CNL for specifying event-condition-action rules, temporal predicates, and data types [5].

2.1.1 Relevance

A data-model driven streaming system would not be a good approach for solving the stated problems. As these languages are bound to a certain data-model, the number of different types of streaming systems that could be described is very limited. Using execution-model driven languages might be a better approach as these can also describe distributed systems. However, these languages are object-oriented. This causes the problem that system properties and stream definitions are not directly related. These languages would not be able to serve as a proper communication tool since the languages are very specific and complex. A target-user driven streaming language would serve as a good communication tool as it is easier to interpret. However, this would result in a system description that is not suitable to generate VHDL from. A language that is a mix of a target-user and an execution-model driven language might be a considerable approach.

2.2 Pebble: A Language for Parametrized and Reconfigurable hardware Design

The article by Wayne Luk and Steve McKeever describes the language “Pebble”. Pebble is an alias for Parametrized Block Language [6]. It is designed to improve the productivity and effectiveness of hardware design. It does that by adopting reusable word-level and bit-level descriptions which can be customized by different parameter values, such as design size and the number of pipeline stages. Pebble code can be compiled into various VHDL dialects without flattening. It improves design effectiveness by supporting optional constraint descriptions at various levels of abstraction, such as placement attributes.

2.2.1 Relevance

Pebble is not a language that is designed specifically for streaming applications. It does however show that adopting reusable and customizable word-level and bit-level descriptions improves productivity and effectiveness of hardware design. This might be relevant when designing the VHDL generator.

2.3 Streaming Application Generator

A popular programming paradigm is the streaming approach to parallel programming. In this paradigm, an application is decomposed into functional blocks which are connected using explicit communication channels. Like traditional programming paradigms, a block of one streaming application can be reused in another. Also, application developers do not have to be concerned with the implementation details within a block, making it more abstract.

Auto-Pipe

An example that uses the streaming approach is Auto-Pipe [7]. In Auto-Pipe, blocks are written in a language suitable for the deployment target. For example, a language such as C or C++ is used for a block targeted for a traditional processor whereas a block targeted for an FPGA device would be implemented in VHDL or Verilog. Auto-Pipe makes use of the X language to specify how the blocks are connected and to what resources they should be assigned. The X language description is used by the Auto-Pipe compiler that generates the necessary runtime infrastructure to connect the blocks, which might be on separate resources. Using X language and Auto-Pipe, streaming applications that run on both traditional processors and FPGA devices simultaneously can easily be written without concern for the low-level details of data movement.

ScalaPipe

A similar example is a streaming application generator for heterogeneous platforms called ScalaPipe. It allows creation of streaming applications that can run on a variety of hardware, including traditional processors, graphics processors, and FPGAs. ScalaPipe makes it easy to generate, modify, and instrument large, complex topologies and resource mappings. ScalaPipe also exposes optimization opportunities [8]. ScalaPipe aims to overcome several deficiencies of Auto-Pipe. In summary, there are a total of 5 points that ScalaPipe addresses.

1) The development and maintenance of X code applications can be very tedious and error prone. This is because creating applications can require writing a great deal of X code. The X language serves essentially to declare the configuration of the blocks. X code specifications are not programs and thus cannot be extended or proceduralized in the manner of most programming languages.
2) It may be required to perform many modifications to the X descriptions for conceptually simple changes to the application or resource mapping. This is due to the static nature of the X language.
3) The X language does not allow type abstraction via polymorphism. Therefore requiring completely separate implementations for simple variations.
4) Blocks must be re-implemented if a change in target is required since Auto-Pipe provides no common language for block implementations.
5) Communication overhead can become substantial even for blocks that are mapped to the same resource. This is due to the separate implementation and coordination languages in Auto-Pipe.

ScalaPipe proposes an approach where it uses a Block Domain Specific Language (Block DSL) and an Application DSL. The Block DSL is used to describe the processing kernels or blocks, and its interfaces. From this Block DSL, code can be generated depending on the targeted platform. This would be C code for a CPU, OpenCL C code for a GPU and Verilog for FPGAs. The HDL code that ScalaPipe generates is functionally correct, but it makes no attempt to optimize it. It creates a state machine to execute the block code sequentially and will therefore not take advantage of the parallel processing capabilities of the FPGA. The Application DSL is used to specify how blocks are connected using functional composition. It also allows to describe the resource mapping. This will generate the corresponding code to interface between each block.

2.3.1 Relevance

Generating a pipeline is an objective that is achieved in both Auto-Pipe and ScalaPipe. However, the system properties and stream definitions are not directly related when using one of these languages, resulting in an unclear relation between the system description and system implementation. Also, both languages are relatively complex in order to serve as a communication tool.

2.4 SamPy concept

A rough concept of a language proposed by ASTRON is SamPy [9]. It is a conceptual language that makes use of the Python syntax. It is based on array notation as described in [10]. The idea of SamPy is to describe a set of interfaces which include its static and dynamic properties. A small example of such a description is given below in Code Snippet 1. A possible block diagram of this description is shown in Figure 1. The interpretation of this description is as follows:

- At level 0, all dimensions are defined (no fields are 'None').
- Higher levels only show the altered dimensions, others are left 'None'.

- Level 0 is lowest, physical level:
  - Serial link = 1-bit wide data.
  - Each period of 1 second, 80 groups of 11968 bits are transferred.
- Level 1 is carried on top of level 0:
  - Data stream is 64b wide.
  - Each period of 1 second, 80 groups of 187 words are transferred.
  - Data rate stays the same.
- Level 2 is carried on top of level 1:
  - Each group is now 180 words (7 words are stripped off).
  - Data rate goes down.
- Level 3 is carried on top of level 2:
  - Data stream width changes from 64b to 48b.
  - The number of words increased from 180 to 240.
  - Data rate stays the same.
- Level 4 is carried on top of level 3:
  - The 48b data is de-concatenated into 2 by 24 bits.
  - Data rate stays the same.
- Level 5 is carried on top of level 4:
  - The dimensions `[ ('Group',80), ('Word' , 240) ]` can now be interpreted as `[ ('Beam',120), ('Sample' , 80), ('Sub', 2) ]`

```
( [ [('Data',  1)               ], [('Period',1), ('Group',80), ('Bit' , 187*64)        ] ], # 0
  [ [('Data', 64)               ], [  None      ,   None      , ('Word', 187)           ] ], # 1
  [ [  None                     ], [  None      ,   None      , ('Word', 180)           ] ], # 2
  [ [('Data', 48)               ], [  None      ,   None      , ('Word', 240)           ] ], # 3
  [ [('Complex', 2), ('Data', 24)], [  None      ,   None      ,   None                  ] ], # 4
  [ [  None,           None,     ], [  None      , ('Beam',120), ('Sample',80), ('Sub',2) ] ]) # 5
```

Code Snippet 1: SamPy concept example

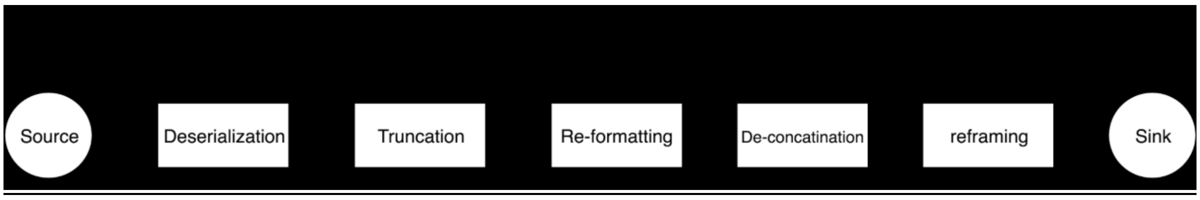


Figure 1: SamPy concept example block diagram

2.4.1 Relevance

The idea of using a popular programming language such as Python might be a good approach as it will have a relatively large audience. This means that SamPy could make communicating about a streaming system easier. Describing a system by a set of interfaces ensures that the system properties and stream definitions are directly related such that the relation between system description and system implementation becomes clear. SamPy can be detailed enough to derive certain functions which may be generatable by a VHDL generator. This could reduce development effort. Considering these points, it seems like a good approach to develop the SamPy concept into a first working prototype.

3 Problem Statement

Large and complex streaming data systems often make use of several types of technology such as analog to digital converters, FPGAs, networking hardware, CPUs, and GPUs. Multiple documents are often studied in order to fully understand these streams. This includes documents such as architectural design documents, detailed design documents, and interface definition documents. Communication between competence groups can therefore be very difficult.

The implementation of the pipeline in such a system is often based on object-oriented design. This implies that the focus during the design process is on the objects, their methods and their attributes. System properties often clearly relate to implementation of an object. However, the system properties are often not related to the stream definitions since this is determined by the object implementation as they have their own interface and parameters. This results in an unclear understanding of the relation between the system properties and the stream definitions.

The output of one object can rarely stream directly into the next object in the pipeline. Extra objects are needed that perform operations such as moving or re-formatting data. Therefore, a lot of time goes into the development of interfaces between objects.

3.1 Objective

Based on the related work discussed in Chapter 2, the most promising approach for solving the stated problems is to expand and implement the SamPy concept into a first working prototype. Additionally, a VHDL generator will be developed that takes the SamPy description as an input and first derives the necessary operations and then creates a VHDL pipeline using those operations. To realize this, the following research questions are defined to serve as a guideline during this project.

1. What information should be contained in one interface definition to describe all of its necessary static and dynamic properties?
2. What interface altering operations need to be derivable from a set of interface definitions and how can this be achieved?
3. Once the operations are derived, how can they best be realized in a VHDL description?
4. How to handle multiple streams (multiple sets of interface definitions) and splitting or merging them?
5. In what way and to what extent can the soundness of a described system be evaluated in an early stage?

In addition to answering the research question, the SamPy and VHDL prototypes developed in this work should meet the following requirements:

- SamPy must be importable as a stand-alone Python module.
- SamPy uses simple syntax to describe systems.
- It must be possible to describe systems hierarchically with SamPy.
- The VHDL generator can generate a complete VHDL pipeline from a SamPy system.
- Generated VHDL code must be human readable and structured.
- Generated VHDL code must be easily relatable to the original system description.
- A VHDL testbench must be generated.

3.2 Use Cases

The resulting prototype will be validated by means of three use cases. These are defined below, the first use case is to test a general streaming data system implementation to test the basic capabilities of the prototype. The second use case is to test a rather complex system specifically designed for ASTRON. The last use case is to show that it is possible to describe abstract systems and tests the extent of system evaluation in an early stage.

3.2.1 Video stream use case

In this use case, a video stream is sent over an optical fiber to the FPGA platform. It is desired to perform some kind of processing which is done in a processing block on the FPGA. This block expects to receive 60 frames of 1080 rows of 1920 pixels in one second. Each pixel consists of four channels: red, green, blue and alpha. Each channel has a width of 8 bits. A block diagram of this system is shown in Figure 2. The optical input contains all the necessary data but it is packaged differently. The input receives 62208 packets of 1000 words in one second, where each word is 64 bits. So in this case, the data rate is the same as:

$$\frac{62208 * 1000 * 64\ b}{1\ s} = \frac{60 * 1080 * 1920 * 4 * 8\ b}{1\ s} = 3981.312\ Mb/s$$

Equation 1: Video stream data rate calculation

Using SamPy, the complete interface from input to processing block can be described. This description can then be used to generate VHDL with the VHDL generator.

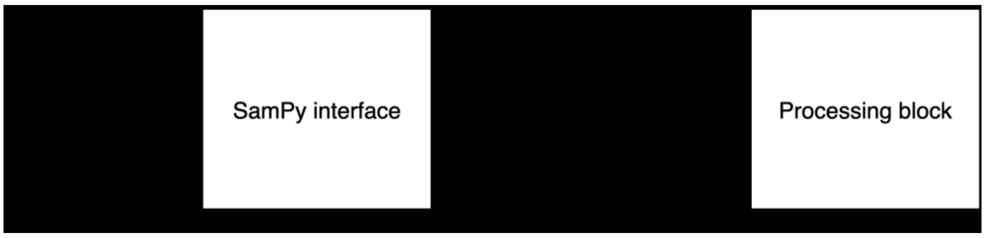


Figure 2: Video stream use case block diagram

3.2.2 ASTRON Streaming data system use case

This use case is relevant for ASTRON as described in Section 1.2. A design dubbed *Science Case 3* is an FPGA application for performing digital signal processing. Roughly, this design consists of three main stages: input, processing and output.

The input of the design consists of 24 optical fiber inputs each transferring 9 Gb/s. The input stage reorders these 24 inputs into 8 sets of 8 streams where each stream transfers 3.375 Gb/s to comply to the ingoing interface of the processing stage. The processing stage reduces the data rate, its outgoing interface is defined as 8 sets of 9 streams where each stream transfers 187.5 Mbps. The output stage reorders the 8 sets of 9 streams into 8 streams to send over 8 optical fiber outputs, each transferring 1.6875 Gb/s. See Figure 3 below.

The alterations to the interfaces performed in the input and output stage should be describable using SamPy. Using this description, a VHDL pipeline should be generated using the VHDL generator. The focus will be on describing and realizing the input stage.

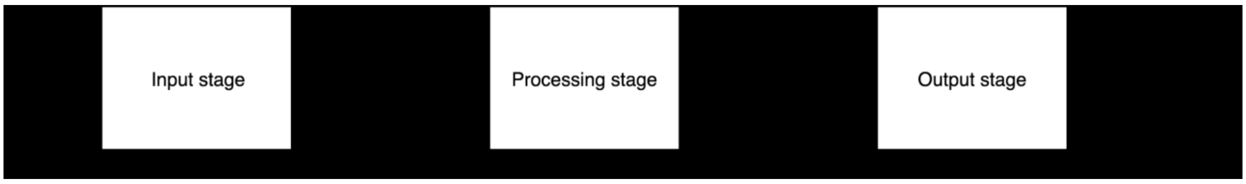


Figure 3: ASTRON Science Case 3 Block Diagram

3.2.3 Fictional cookie factory use case

SamPy should be generic enough to describe other types of streaming systems. A very simplified fictional cookie factory will be used as an example. In this use case, there will be no VHDL generation. The purpose of this use case is to test whether the SamPy language can be used to describe more abstract streaming systems. Figure 4 below shows a block diagram of the system in this use case.

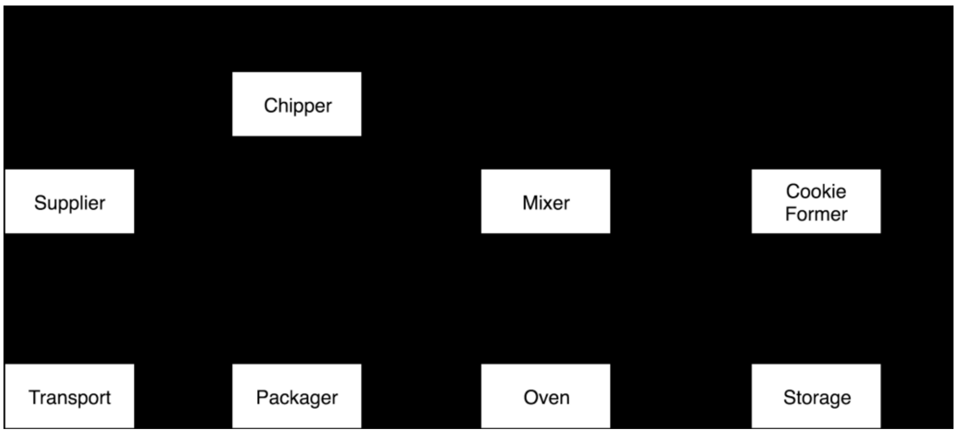


Figure 4: Fictional Cookie Factory use case block diagram

4 SamPy

Streaming Array Modelling with Python (SamPy) is a way of defining the interfaces of a streaming system. In this chapter it is explained how a SamPy system can be described in Python using the SamPy module. A SamPy system description consists of three main parts. First the interfaces must be defined, these are written in a certain way that is explained in Subsection 4.1. The interface definitions are then used to describe a stream, this is explained in Subsection 4.2. Finally, the system description can be obtained by combining the streams as shown in Subsection 4.3.

4.1 SamPy Interface

This chapter describes the format and interpretation of a SamPy interface. Subsection 4.1.1 shows the format in which an interface can be defined. In Subsection 4.1.2, the interpretation of an interface definition is explained. Subsection 4.1.3 contains three examples to further explain the SamPy interface description. Subsection 4.1.4 shows the various attributes that a SamPy interface has.

4.1.1 Format

A SamPy interface definition has the format as defined below. The format of each field in a SamPy interface definition is explained in the following subsections. Please see Appendix A: SamPy BNF Notation for a formal BNF notation.

```
Interface( source, sink, instances, signals, dynamic )
```

4.1.1.1 Source and Sink

The source and sink fields have the same format and can be one of the following options:

- String, e.g. “Component” or ‘Component’
- Tuple of a string and an integer, e.g. (“Component”, 1)
- SamPy system, e.g. mySamPySystem
- Tuple of a SamPy system and an integer, e.g. (mySamPySystem, 3)

4.1.1.2 Instances and Signals

The instances and signals fields have the following format, they must be a list of one or more tuples containing a string and an integer. Note that a string in Python can be defined between single or double quotes. For example:

```
[ (“stringA”, 4), (“stringB”, 2) ]
```

An optional time parameter can be added using a number to indicate seconds or a number and a string to indicate time in a different time scale. For example:

```
[ (“stringA”, 4, 1.3), (“stringB”, 2, (2, “ms”)) ]
```

In the example above, the time parameter of *stringA* is 1.3 seconds and the time parameter of *stringB* is 0.002 seconds. The different time scale strings that can be used are defined in Table 1.

Table 1: Time scales

Time scale	**String**
days	“d”
Hours	“h”
Minutes	“m”
Seconds	“s”
Milliseconds	“ms”
Microseconds	“us”
Nanoseconds	“ns”
Picoseconds	“ps”
Femtoseconds	“fs”

4.1.1.3 Dynamic

The dynamic properties must be a list of two or more tuples where a tuple has the format (S, N, T) where S is a string and N is a positive integer. N can also be -1 if it is in the first tuple of the list. T is an optional time parameter. A time parameter can either be a number to indicate a time in seconds or a number followed by a string to indicate the timescale as defined in the table above. At least one of the tuples in the dynamic field must have a defined T. For example:

```
[(“Period”, -1, (1.0, ‘s’)), (“Packet”, 10), (“Element”, 80)]
```

The time parameter T must be equal or smaller than T/N of the previous tuple, i.e.

[(S_0, N_0, T_0), (S_1, N_1, T_1)], where $T_1 <= T_0 / N_0$

4.1.2 Interpretation

The interpretation of a defined SamPy interface is explained in the subsections below.

4.1.2.1 Source

The source of an interface is an output of a component. This is defined by the name of the component and the port number or index of the used output, e.g. (“component”, 1). When no port number is given, it is assumed to be port 0. Instead of the name of a component, it is also possible to use a predefined SamPy system.

4.1.2.2 Sink

The sink of an interface is an input of a component. This is defined by the name of a component and the port number or index of the used input, e.g. (“component”, 1). When no port number is given, it is assumed to be port 0. Instead of the name of a component, it is also possible to use a predefined SamPy system.

4.1.2.3 Instances

An instance in the SamPy context is a group of signals or a group of other instances. The example below shows two instances describing a total of four Streams where each *Stream* is a group of 2 *Channels*. *Channel* is the last instance in the list, this means that a *Channel* must be a group of signals. Signals are explained in the following subsection.

```
[ (“Stream”, 4), (“Channel”, 2)]
```

The format definition of an instance shows that it can also contain a time parameter, e.g.

```
[ (“Stream”, 4, (4.0, ‘ms’)), (“Channel”, 2, (1.0, ‘ms’))]
```

The time parameter implies a maximum misalignment. In the example above, the misalignment of *Channel* is 1 millisecond. Therefore when the Nth element in *Channel* 0 is transferred, the Nth element in *Channel* 1 is expected to be transferred at most 1 millisecond earlier or later. Similarly, the misalignment of *Stream* is 4 milliseconds. This implies that when the Nth element of *Stream x, Channel q* is transferred, the Nth element in *Stream y, Channel r* is transferred at most 4 + 1 millisecond earlier/later. Where:

- $x \neq y$
- x and y must be in range [0,3] since there is a total of 4 streams.
- q and r must be in range [0,1] since there is a total of 2 channels per stream.

For example, the misalignment between *Channel* 0 of *Stream* 2 and *Channel* 1 of *Stream* 2 is 1 ms as they are within the same *Stream*. The misalignment between *Channel* 0 of *Stream* 0 and *Channel* 0 of *Stream* 2 is 5 ms as they are in separate *Streams* and therefore have an additional 4 ms misalignment.

4.1.2.4 Signals

Signals are used to transfer an element. Signals must have a name and a width. The list:

```
[(“Data”, 32)]
```

only contains one signal named *Data* and has a width of 32. This implies that an element consists of 32 smaller parts. In this example these smaller parts are bits which are transferred using one *Data* signal with a width of 32.

A Signal can also have a time parameter that indicates the misalignment between the bits of an element, e.g.

```
[(“Data”, 32, (1, ‘ns’))]
```

implies that the time difference between receiving the first bit and the last bit of an element is at maximum 1 nanosecond.

A list with multiple signals implies that an element is separated in multiple groups of smaller parts (or bits). For example, a complex number could be transferred using a 16 bit real part and a 16 bit imaginary part, e.g.

```
[(“Real”, 16), (“Imaginary”, 16)]
```

4.1.2.5 Dynamic

As stated in the format definition, the dynamic part is a list of two or more tuples with the format (S, N, T), where T is optional in all but one tuple. An example of a dynamic definition can be:

```
[(“Period”, -1, 1.0), (“Packet”, 10), (“Element”, 80)]
```

The first tuple indicates that there exists a time frame named *Period* that is repeated indefinitely (-1 indicates infinite) which takes exactly 1 second. The second tuple implies that in one *Period* there exist 10 time frames named *Packet*. As no time parameter is given for *Packet*, it is assumed to be 1/10 = 0.1 seconds. Similarly, the last tuple indicates that in one *Packet* there exist 80 time frames named *Element*. No time parameter is given for *Element,* therefore it is assumed that 1 *Element* takes 0.1/80 = 0.00125 seconds. The last defined time frame is exactly the time in which one element is transferred over the signals.

Another example that shows the function of the time parameter is defined below:

```
[(“Period”, -1, 1.0), (“Packet”, 10), (“Element”, 80, (1, ‘ms’))]
```

The first and second tuple are identical to the previous example. However, the last tuple now indicates that the *Element* frame takes 1 millisecond. Therefore all 80 *Element* frames take 80 milliseconds. One *Packet* frame takes 100 milliseconds and contains the 80 *Element* frames. This implies that a *Packet* frame has a gap of 20 milliseconds. This timing property is defined such that the first frame starts simultaneously with the larger timeframe it is in. In this example, the first *Element* frame takes place at the start of a *Packet* frame. The remaining 79 *Element* frames can be spread out over the remaining 99 milliseconds in that *Packet* frame.

4.1.3 Examples

Examples of SamPy interfaces are explained in the subsections below.

4.1.3.1 Example A

An example of a SamPy interface is given below. The example is visualized in Figure 5.

```
myInterface = Interface(

'A', 'B', [ ('Stream', 2) ], [ ('dataLine', 32) ], [ ('Period',-1,1.0), ('Word', 5) ]

)
```

Code Snippet 2: Example Interface A

When compared to the definition of a SamPy interface you can see that:

source = `'A'`
sink = `'B'`
instances = `[ ('Stream', 2) ]`
signals = `[ ('DataLine', 32) ]`
dynamic = `[ ('Period', -1, 1.0), ('Word', 5) ]`

This interface should be interpreted as follows:

- It is an interface going from the output of *A* (see source) to the input of *B* (see sink).
- The interface consists of 2 parallel *Streams* (see instances).
- For every *Stream* there exist 32 parallel *DataLines* (see signals).
- The system will run for an infinite number of periods (-1). In each period of 1.0 second, 5 *Words* are transferred on every *Stream* (see dynamic). In this example, a *Word* consists of 32 bits and therefore needs the 32 *DataLines* I order to be transferred.

So on this interface, a total of *Stream*Word/Period time* = 2*5/1.0 = 10 *Words/second* or 10*32 = 320 bits/second are transferred from *A* to *B*.

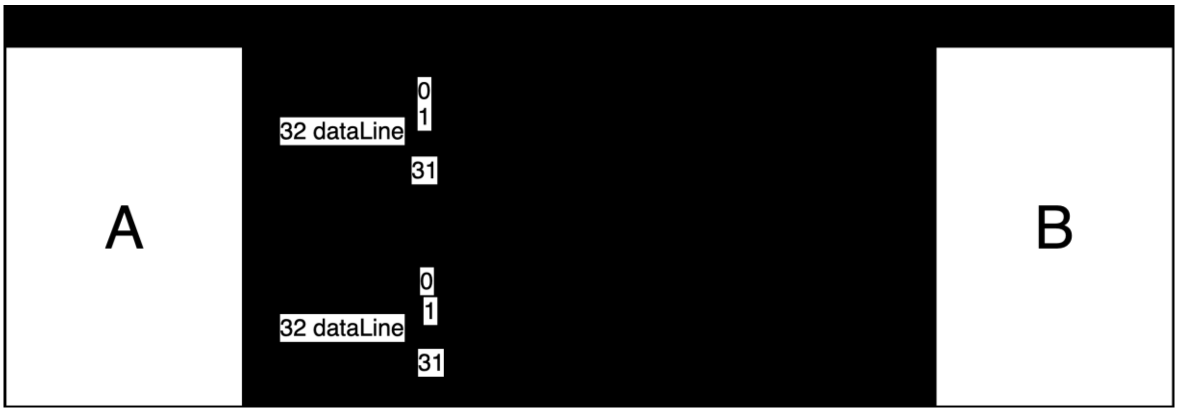


Figure 5: Schematic of the SamPy interface in Example A

4.1.3.2 Example B

An example of a larger interface is given below and visualized in Figure 6.

```
myInterface = Interface(
‘A’,’B’,[(‘Stream’,2), (‘Channel’, 2)], [(‘A_Line’,20), (‘B_Line’,12)],
[(‘Period’, 100, 1.0), (‘Packet’, 8), (‘Word’, 5)]
)
```

Code Snippet 3: Example Interface B

When compared to the definition of a SamPy interface you can see that:

source = `‘A’`
sink = `‘B’`
instances = `[ (‘Stream’, 2), (‘Channel’, 2) ]`
signals = `[ (‘A_Line’, 20), (‘B_Line’, 12) ]`
dynamic = `[ (‘Period’ 100, 1.0), (‘Packet’, 8), (‘Word’, 5) ]`

This interface should be interpreted as follows:

- It is an interface going from the output of *A* (see source) to the input of *B* (see sink).
- The interface consists of 2 parallel *Streams*, every *Stream* contains 2 parallel *Channels* (see instances).
- Each *Channel* consists of the combination of 20 parallel *A_Lines* and 12 parallel *B_Lines* (see signals).
- The system will run for 100 periods of 1 second. Each period, 8 *Packet*s are transferred over every *Channel* where each *Packet* is a group of 5 *Words* (see dynamic). In this example, a *Word* consists of one 20 bits part and another 12 bits part, it therefore needs the 20 *A_Lines* and 12 *B_Lines in order* to be transferred.

So on this interface, a total of *Streams*Channels*Packets*Words/Period time* = 2*2*8*5/1.0 = 160 *Words/second* or (20+12)*160 = 5120 bits/second is transferred from *A* to *B*.

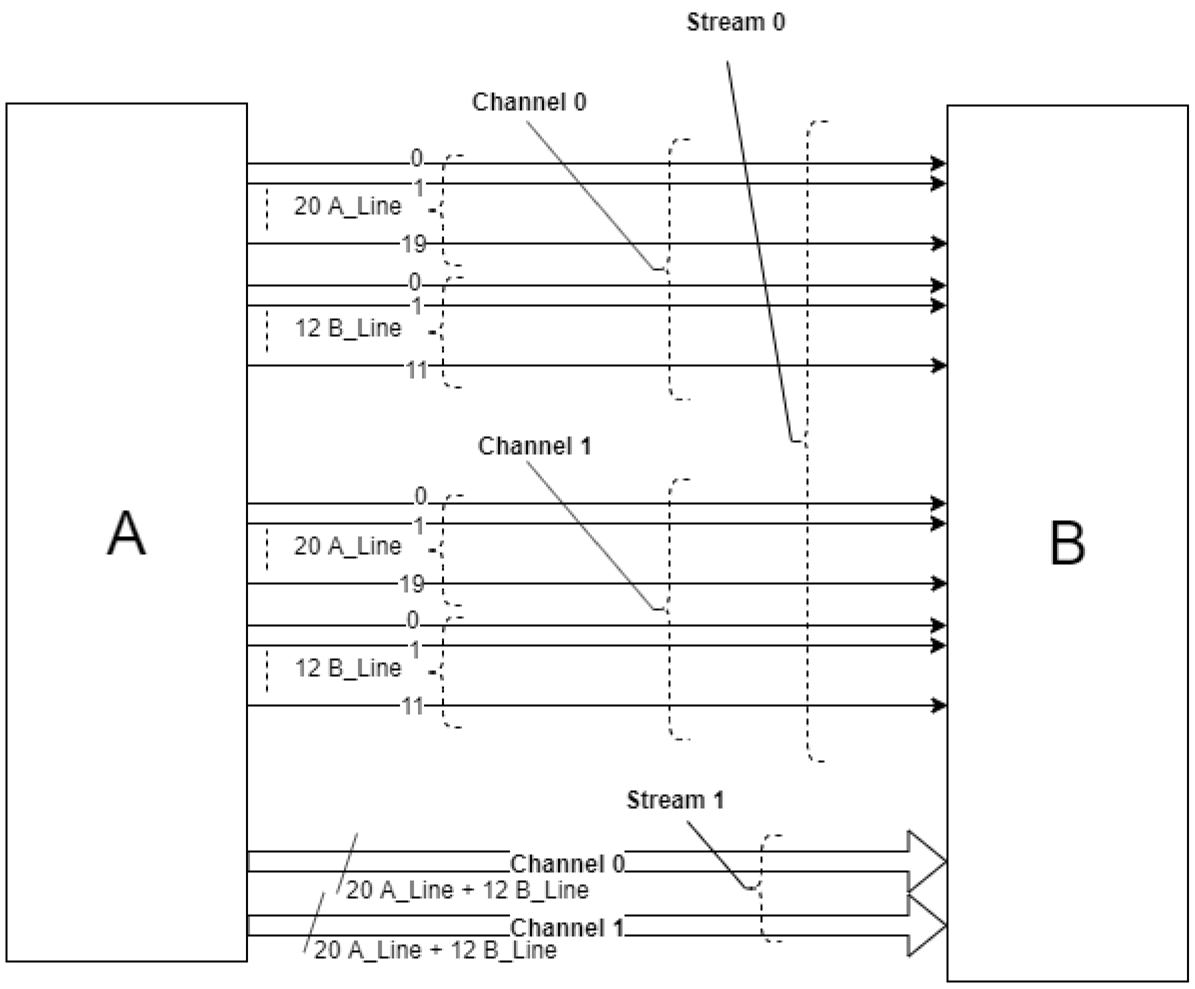


Figure 6: Schematic of the SamPy interface in Example B

4.1.3.3 Example C

An example of a more complex interface is defined below.

```
myInterface = Interface(
'A','B',[('Stream',2), ('Channel', 2, (20, 'ms'))], [('A_Line',20), ('B_Line',12)],
[('Period', 100, 1.0), ('Packet', 8), ('Word', 5, (10, 'ms'))]
)
```

Code Snippet 4: Example Interface C

When compared to the definition of a SamPy interface you can see that:

source = `'A'`
sink = `'B'`
instances = `[ ('Stream', 2), ('Channel', 2, (20, 'ms')) ]`
signals = `[ ('A_Line', 20), ('B_Line', 12) ]`
dynamic = `[ ('Period' 100, 1.0), ('Packet', 8), ('Word', 5, (10, 'ms')) ]`

This interface should be interpreted as follows:

- It is an interface going from the output of *A* (see source) to the input of *B* (see sink).
- The interface consists of 2 parallel *Streams*, every *Stream* contains 2 parallel *Channels* (see instances), the *Channels* have a maximum misalignment of 20 milliseconds with respect to each other.
- Each *Channel* consists of the combination of 20 parallel *A_Lines* and 12 parallel *B_Lines* (see signals).
- The system will run for 100 periods of 1 second. Each period, 8 *Packet*s are transferred over every *Channel* where each *Packet* is a group of 5 *Words*. Each word gets transferred in 10 milliseconds, therefore all 5 words are transferred in 5*10=50 ms. One *Packet* is transferred in 1s/8=125 ms. This means that a packet contains 5 words that take a total of 50 ms, therefore there exist a total gap time of 75 ms (see dynamic). In this example, a *Word* consists of one 20 bits part and another 12 bits part, it therefore needs the 20 *A_Lines* and 12 *B_Lines in order* to be transferred.

So on this interface, a total of *Streams*Channels*Packets*Words/Period time* = 2*2*8*5/1.0 = 160 *Words/second* or (20+12)*160 = 5120 bits/second is transferred from *A* to *B*. The timing properties of this interface are visualized in Figure 7. In this example, it is assumed that the 5 words are transferred consecutively followed by a gap. It could also be possible that the words are more spread out, with gaps in between.

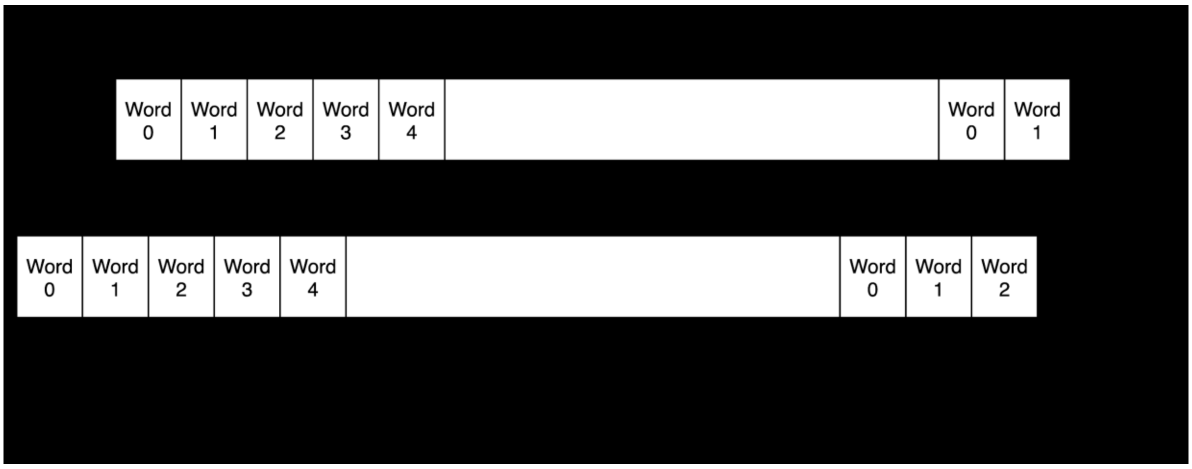


Figure 7: Time properties of the SamPy interface in Example C

4.1.3.4 Example D

In the previous example, it was uncertain where the *Word* frames exist in the *Packet* frame. In this example, it is shown how the dynamic properties of an interface can be described in a way such that the timing of the frames is exact. The following example interface is defined:

```
myInterface = Interface(
‘A’,’B’,(‘Channel’, 1)], [(‘Data’,32)],
[(‘Period’, -1, 1.0), (‘Packet’, 8), (‘Word’, 5), (‘element’, 1, (5, ‘ms’))]
)
```

Code Snippet 5: Example Interface D

When compared to the definition of a SamPy interface you can see that:

source = `‘A’`
sink = `‘B’`
instances = `[ (‘Channel’, 1) ]`
signals = `[ (‘Data’, 32) ]`
dynamic = `[(‘Period’, -1, 1.0), (‘Packet’, 8), (‘Word’, 5), (‘element’, 1, (5, ‘ms’))]`

This interface should be interpreted as follows:

- It is an interface going from the output of *A* (source) to the input of *B* (sink).
- The interface consists of 1 *Channel* (instances)
- The *Channel* consists of one *Data* signal that has a width of 32 (signals).
- The system will run for an infinite number of periods of 1 second. Each period, 8 *Packet*s are transferred on the *Channel* where each *Packet* is a group of 5 *Words*. A *Word* consists of 1 *element* that takes 5 ms. One *Packet* is transferred in 1s/8=125 ms, therefore 1 *Word* is transferred in 125/5 = 25 ms. This implies that a word must first transmit the 1 *element* and then a gap of 20 ms as the first time frame is always at the start of the larger time frame it is part of. This is visualized in Figure 8 below.

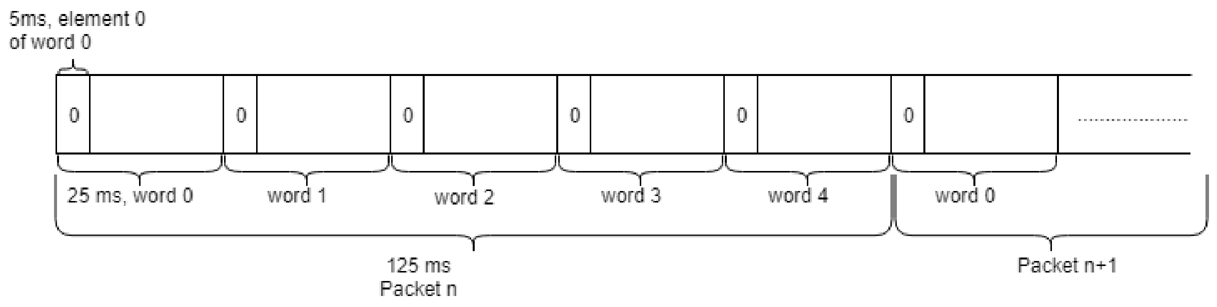


Figure 8: Time properties of SamPy interface in Example D

4.1.4 Attributes

Once an interface is defined, its attributes can be accessed. The attributes of an interface are defined below.

4.1.4.1 Source and Sink

The source and sink attributes are a 3-Tuple containing a string, an integer, and either a SamPy system when a subsystem is used or None if no subsystem is used. The attribute can have one of the four possible values (depending on the initialization of the source or sink) as shown in Table 2.

Table 2: SamPy interface attributes

initialization	**attribute value**
String, e.g. “Component”	(String, 0, None), e.g. (“Component”, 0, None)
Tuple of a string and an integer, e.g. (“Component”, 1)	(String, integer, None), e.g. (“Component”, 1, None)
SamPy system, e.g. mySamPySystem	(SamPy system name, 0, SamPy system), e.g. (mySamPySystem.name, 0, mySamPySystem)
Tuple of a SamPy system and an integer, e.g. (mySamPySystem, 3)	(SamPy system name, integer, SamPy system), e.g. (mySamPySystem.name, 3, mySamPySystem)

4.1.4.2 Instances

The instances attribute is a list of 3-tuples, similar to the initialization. The first two elements of a 3-tuple are identical to that of the initialization. The last element is the time parameter in seconds. For example, if an interface is initialized with the following instances:

```
[(‘Stream’,2), (‘Channel’, 2, (20, ‘ms’))]
```

The instances attribute is then defined as:

```
[(‘Stream’,2, 0.0), (‘Channel’, 2, 0.02)]
```

4.1.4.3 Signals
Just as the instances attribute, the signals attribute is a list of 3-tuples, similar to the initialization. For example, if an interface is initialized with the following signals:

```
[(‘A_Line’,20, (20, ‘ns’)), (‘B_Line’,12)]
```

The signals attribute is then defined as:

```
[(‘A_Line’,20, 0.00000002), (‘B_Line’,12, 0.0)]
```

4.1.4.4 Static
The static attribute is a concatenation of the instances and the signals.

4.1.4.5 Period
The period attribute is the first 3-tuple of the dynamic attribute.

4.1.4.6 Dynamic
The dynamic attribute is a list of 3-tuples similar to the initialization of the dynamic part but the time parameters that were missing in the initialization are now derived from the time parameters that were given. In addition, all time parameters are converted to seconds. For example, the dynamic part is initialized with:

```
[(‘Period’, 100, 1.0), (‘Packet’, 8), (‘Word’, 5, (10, ‘ms’))]
```

The dynamic attribute will be:

```
[(‘Period’, 100, 1.0), (‘Packet’, 8, 0.125), (‘Word’, 5, 0.01)]
```

4.1.4.7 Example
Consider the definition of *myInterface* below. This interface has the attributes as defined in Table 3.

```
from SamPy import Interface, Stream, System

myInterface = Interface(

‘A’,’B’,[(‘Stream’,2), (‘Channel’, 2, (20, ‘ms’))], [(‘A_Line’,20), (‘B_Line’,12)],

[(‘Period’, 100, 1.0), (‘Packet’, 8), (‘Word’, 5, (10, ‘ms’))]

)
```

Code Snippet 6: Example Interface

Table 3: Example attributes of myInterface description

Attribute	Value
myInterface.source	`(“A”, 0, None)`
myInterface.sink	`(“B”, 0, None)`
myInterface.instances	`[ (‘Stream’, 2, 0), (‘Channel’, 2, 0.02) ]`
myInterface.signals	`[ (‘A_Line’, 20, 0), (‘B_Line’, 12, 0) ]`
myInterface.static	`[ (‘Stream’, 2, 0), (‘Channel’, 2, 0.02), (‘A_Line’, 20, 0), (‘B_Line’, 12, 0) ]`
myInterface.period	`(‘Period’, 100, 1.0)`
myInterface.dynamic	`[(‘Period’, 100, 1.0), (‘Packet’, 8, 0.125), (‘Word’, 5, 0.01) ]`

4.2 SamPy Stream

A SamPy stream consists of one or more SamPy interfaces. Creating a Stream can be done by either listing previous defined SamPy interfaces or defining the interfaces within the stream creation. A formal BNF notation can be found in Appendix A: SamPy BNF Notation.

Example:

```
StreamA = Stream(interfaceA, interfaceB, interfaceC)
```

Code Snippet 7: Stream definition by listing previously defined interfaces

or:

```
StreamA = Stream(
  Interface( 'Input', 'A'     , [('Ch', 1)], [('Data', 32)], [('Period',-1,1), ('Word', 100)] ),
  Interface( 'A'    , 'B'     , [('Ch', 1)], [('Data', 16)], [('Period',-1,1), ('Word', 200)] ),
  Interface( 'B'    , 'Output', [('Ch', 2)], [('Data',  8)], [('Period',-1,1), ('Word', 200)] )
)
```

Code Snippet 8: Stream definition with embedded interface definitions

A SamPy stream shows how one interface is transformed into the next interface and by which component. For example, consider the SamPy stream definition of *StreamA* above. The first and second interface definition show that component *A* receives 100 *Words* on 32 *Data* signals in 1 second and transmits 200 *Words* on 16 *Data* signals in 1 second. The second and third interface definitions show that component *B* receives 200 *Words* on 16 *Data* signals for 1 *Ch* in 1 second and transmits 200 *Words* on 8 *Data* signals for 2 *Ch* in 1 second. A visualization of this stream is shown in Figure 9 below.

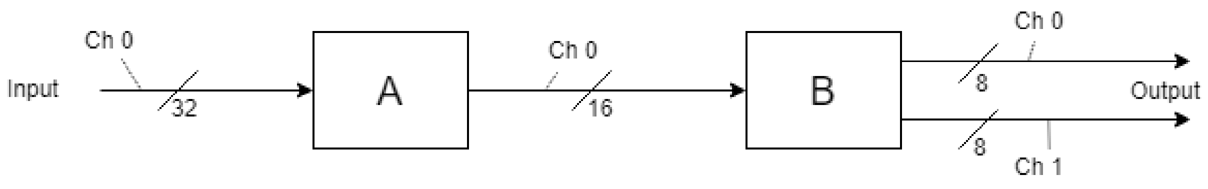


Figure 9: Stream example diagram

4.2.1 Attributes

A SamPy stream only has one attribute which is *interfaces*. This attribute is a list of interface objects.

4.3 SamPy System

This chapter discusses the SamPy system. A SamPy system consists of a name, an optional constraints dictionary and one or more SamPy streams. An example of an unconstrained and a constrained system is given below. Subsection 4.3.1 explains the idea of using constraints. Subsection 4.3.2 explains the evaluation performed by SamPy. The attributes of a SamPy system are explained in Subsection 4.3.3. Subsection 4.3.4 explains the concept of the system components. In Subsection 4.3.5, some examples of complete system descriptions are shown. Please see Appendix A: SamPy BNF Notation for a formal BNF notation.

```
mySamPySystem = System("mySys", stream0, stream1)

constraints = {"myConstraint" : 10}
constrainedSystem = System("constrainedSys", constraints, stream0, stream1)
```

Code Snippet 9: Example of an unconstrained and a constrained SamPy system

4.3.1 Constraints

The user can provide constraints to the system by defining a Python dictionary. SamPy itself does not use these constraints in any way. The constraints are meant to provide implementation specific information. This information can be used by the VHDL generator. A system can have system wide constraints or component specific constraints. A system wide constraint is simply defined in the dictionary by a key and value. To define a constraint for a component, a key is used that is identical to the name of the corresponding component. The value is then a new dictionary containing the actual constraints for that component. An example of such a constraints dictionary is shown below.

```
Constraints = {"aSystemConstraint" : 10,
               "component_name"    : { "aComponentConstraint" : 5 }

}
```

Code Snippet 10: Example constraints dictionary

4.3.2 SamPy System Evaluation

Once a SamPy system description is complete, the Python script can be executed. This will automatically evaluate the system and provides the user with feedback. The user receives an error for describing impossible interface descriptions. For example, connecting multiple output ports to one input port would result in an error. Warnings are provided to inform the user about potentially unwanted system behaviour. Data loss and an altered transfer rate are examples of behaviour that would result in a warning. A SamPy system object also contains a method that generates a simple block diagram. This can be executed by calling:

```
mySamPySystem.generateBlockdiag('Name_of_figure.png')
```

4.3.3 Attributes

A SamPy system has several attributes as defined in Table 4 below. One of these attributes is *components*. This attribute is further explained in Section 4.3.4.

Table 4: SamPy system attributes

Attribute	**Value**
streams	List of all streams in the system
name	String that represents the name of the system
constraints	dictionary containing key-value pairs as defined during initialization.
inputInterfaces	List of all interfaces that are considered an input to the system
outputInterfaces	List of all interfaces that are considered an output to the system
components	List of all components in the system

4.3.4 Components

One of the attributes of a SamPy system is a component list. A component is derived from two or more interfaces that share the same sink or source name. For example, the two interfaces below create a component with the name: 'A'

```
Interface( 'Input', 'A'     , [('Stream', 1)], [('Data', 32)], [('Period',-1,1), ('Word', 100)] )
Interface( 'A'    , 'B'     , [('Stream', 1)], [('Data', 16)], [('Period',-1,1), ('Word', 200)] )
```

In addition to a name, a component has several attributes. These are defined in Table 5 below.

Table 5: SamPy system component attributes

Attribute	**Value**
name	String that represents the name of the component
inputInterfaces	List of all interfaces that are considered an input to the component.
outputInterfaces	List of all interfaces that are considered an output to the component.
inputTransferRate	The sum of the average transfer rates of all input interfaces, e.g. 500K bits per second.
outputTransferRate	The sum of the average transfer rates of all output interfaces, e.g. 500K bits per second.
inputTransferSpeed	The sum of the average transfer speeds of all input interfaces, e.g. 100 Words per second where Word is the last element in the dynamic properties list of the input interfaces.
outputTransferSpeed	The sum of the average transfer speeds of all output interfaces, e.g. 100 Words per second where Word is the last element in the dynamic properties list of the output interfaces.
constraints	dictionary containing all system constraints where the key matches the component name.

4.3.5 Examples

Two examples of SamPy system descriptions are given in the subsections below. Note that the user has to import the SamPy module when creating a SamPy script. This is done using the line of Python code:

```
from SamPy import Interface, Stream, System
```

4.3.5.1 split/merge example

In Section 4.1.2, the concept of a port number is explained. So, a component with two outputs has a port 0 and a port 1. This can be useful when it is desired to split an interface or merge interfaces. An example of a system that makes use of this feature is shown below. In this example, component *A* splits an interface into two interfaces. Output port 0 of *A* goes to *X* and output port 1 of *A* goes to input port 1 of *B*. The output of *X* goes to input port 0 of *B.* Note that using ('A', 0) for source or sink is identical as using just 'A'. This can also be seen in Figure 10.

```
from SamPy import Interface, Stream, System


stream0 = Stream(
  Interface( 'src' ,  'A'   , [('System', 1)], [('Data', 64)], [('Period', -1,1), ('Word', 187)]),
  Interface(('A',1), ('B',1), [('System', 1)], [('Data', 64)], [('Period', -1,1), ('Word', 180)]),
  Interface( 'B'   ,  'snk' , [('System', 1)], [('Data', 64)], [('Period', -1,1), ('Word', 187)])
)


stream1 = Stream(
  Interface(('A',0),  'X'   , [('System', 1)], [('Data', 64)], [('Period', -1,1), ('Word', 7)]),
  Interface( 'X'   , ('B',0), [('System', 1)], [('Data', 64)], [('Period', -1,1), ('Word', 7)])
)


mySys = System("ExampleSystem", stream0, stream1)
```

Code Snippet 11: Split/merge example system

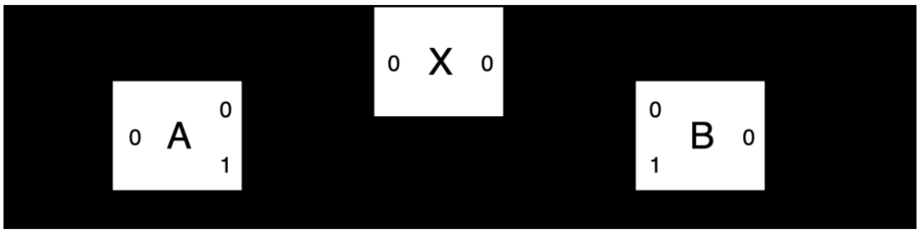


Figure 10: SamPy system, split/merge example block schematic

4.3.5.2 *subsystem example*

For creating hierarchy, a predefined SamPy system as source/sink can be used. An example of this is given below and visualized in Figure 11.

```
from SamPy import Interface, Stream, System


StreamA = Stream(
  Interface( 'src', 'A'  , [('Stream', 1)], [('Data', 32)], [('Period', -1, 1), ('Word', 100)] ),
  Interface( 'A'  , 'B'  , [('Stream', 1)], [('Data', 16)], [('Period', -1, 1), ('Word', 200)] ),
  Interface( 'B'  , 'snk', [('Stream', 2)], [('Data', 8 )], [('Period', -1, 1), ('Word', 200)] )
)
subSystem = System('subSystemName', StreamA)



StreamB = Stream(
  Interface( 'Input', 'A'       , [('Stream', 1)], [('Data', 32)], [('Period',-1,1),('Word', 100)]),
  Interface( 'A'    , subSystem , [('Stream', 1)], [('Data', 32)], [('Period',-1,1),('Word', 100)]),
  Interface( subSystem, 'Output', [('Stream', 2)], [('Data', 8) ], [('Period',-1,1),('Word', 200)] )
)
testSys = System('testSysName', StreamB)
```

Code Snippet 12: SamPy subsystem example

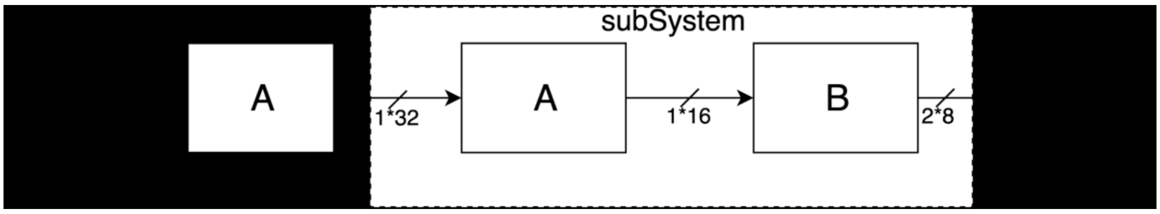


Figure 11: SamPy system, subsystem example

5 VHDL Generator

The VHDL generator improves on interface development time, this is one of the stated problems. The development of the VHDL generator has helped with defining what information should be contained in a SamPy interface definition to describe all its necessary static and dynamic properties. The research question on how to derive a function from a set of interface definitions is answered by the development of the VHDL generator. It is also shown how such a function is realized in VHDL. Finally, the research question on how to split and merge multiple streams is answered by the realization of a derived function.

The first subsection of this chapter will give an overview of the VHDL generator and the generated VHDL. In the following subsections, each part of the VHDL generator is explained in more detail.

5.1 Overview

The VHDL generator is a Python module which can simply be imported in a Python script. The user can generate a system described in SamPy by means of the function *generateSystem*. A variation of this function named *genRadioHDLProject* is available which generates an ASTRON VHDL project. An overview on how all the VHDL files are generated is shown in Figure 13. This shows that from a SamPy system description, four main operations are performed. The components mentioned in this diagram are SamPy components. As explained in Section 4.3.3, a SamPy system contains a list of components. These SamPy components are used in the VHDL generator to derive their function. Then, the VHDL description for that component is generated. Three other VHDL files are generated to make the system complete: top-level pipeline, testbench, and VHDL type package. The pipeline is generated to connect all the generated components. The testbench is to provide a simple simulation setup. Finally, a VHDL type package is generated to provide all the required types. Generating the files is done by means of Python functions that write VHDL code to a VHDL file, these functions are referred to as templates in the coming sections. A diagram of a generated system is shown in Figure 12.

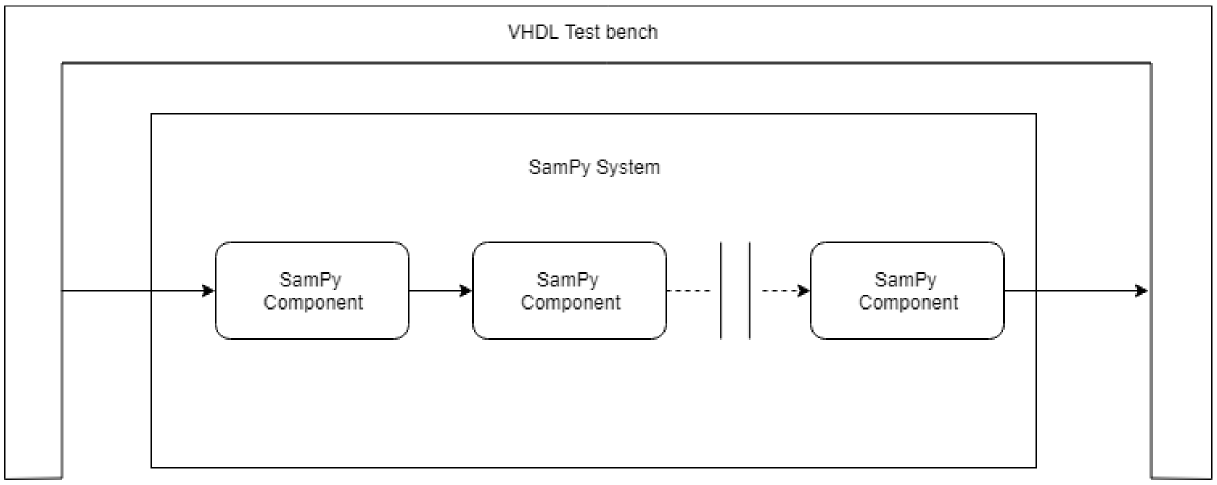


Figure 12: Generated VHDL overview

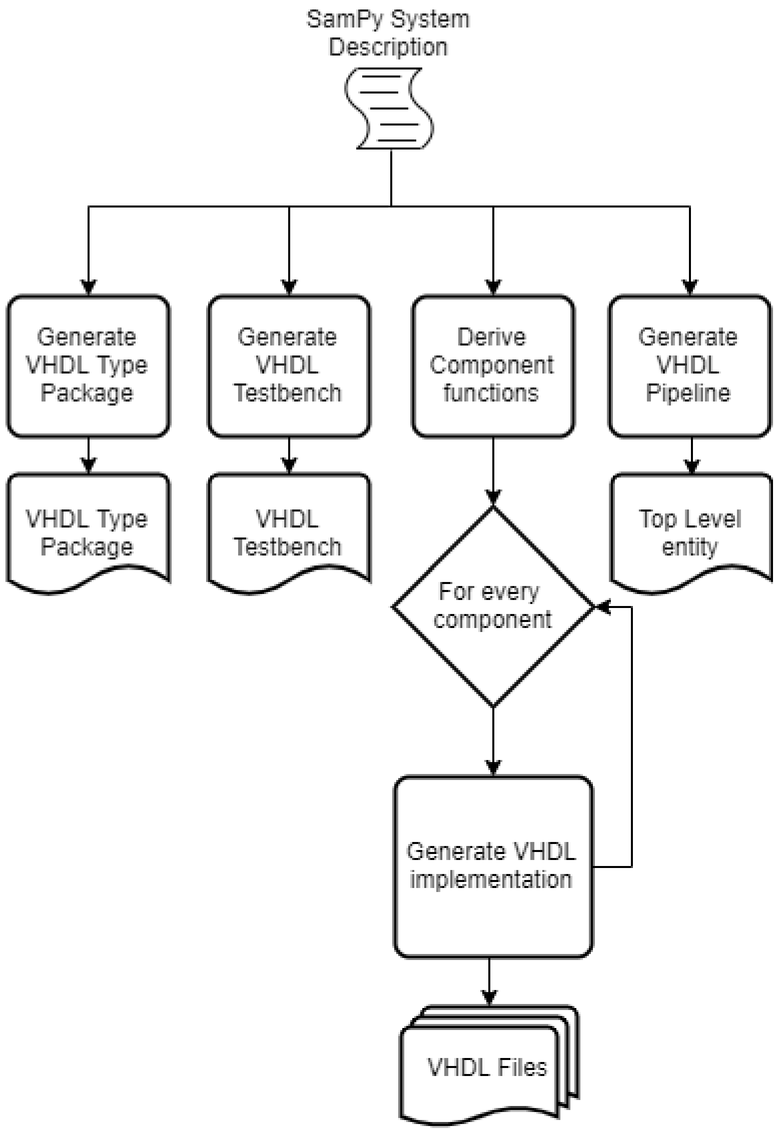


Figure 13: VHDL generator overview

5.2 Limitations

When writing a SamPy description that is going to be used by the VHDL generator, a few things must be considered. Firstly, all the strings used in a SamPy description are directly used in VHDL code. This limits the user to a subset of names. A name must comply to the formal definition of a VHDL basic identifier [11].

The VHDL generator can only generate a system working on a single clock. Therefore, the last dynamic property of all interfaces in a SamPy description must all have the same period as it defines the clock period.

Although it can be specified in a SamPy description, misalignment between signals is ignored by the VHDL generator since it is assumed that the data is available on the active clock edge.

5.3 Generating VHDL Interface Package

Every SamPy interface described in a SamPy system will have a corresponding VHDL type. These types are all defined in one single VHDL package. An interface has a tree-like structure that can consist of multi-dimensional arrays that correspond with the instances defined in the SamPy interface description. The elements in these arrays are of a record type that contain standard logic vectors (std_logic_vector) that correspond to the signals of a SamPy interface and natural numbers that represent the dynamic properties of an interface. In addition, a control record is included which contains some ASTRON specific signals to comply with their existing IPs. To make this a bit clearer, an example is shown below.

5.3.1 VHDL Interface Package example

In this example, consider the following SamPy interface description.

```
Interface('input',  'some_component' ,
          [('Stream', 24), ('Ch', 8)],  [('Data', 64)],
          [('Period', -1, (1.024, ‘s’)), ('Packet', 8000), ('Word', 187, (5, 'ns') ) ]
         )
```

Code Snippet 13: Example SamPy interface description

The generated type package will always contain the ASTRON specific control record, this is shown in Code Snippet 14. The resulting VHDL type for the interface is shown in Code Snippet 15. The tree structure of this interface is shown in Figure 14.

```
TYPE SamPy_control_type IS RECORD
  valid : STD_LOGIC;
  sop : STD_LOGIC;
  eop : STD_LOGIC;
  sync : STD_LOGIC;
  bsn : STD_LOGIC_VECTOR(c_dp_stream_bsn_w-1 DOWNTO 0);
  empty : STD_LOGIC_VECTOR(c_dp_stream_empty_w-1 DOWNTO 0);
  channel : STD_LOGIC_VECTOR(c_dp_stream_channel_w-1 DOWNTO 0);
  err : STD_LOGIC_VECTOR(c_dp_stream_error_w-1 DOWNTO 0);
END RECORD;
```

Code Snippet 14: ASTRON control record

```
--input_0
TYPE record_input_0_Ch IS RECORD
  Data : STD_LOGIC_VECTOR(63 DOWNTO 0);
  Period : NATURAL;
  Packet : NATURAL;
  Word : NATURAL;
  control : SamPy_control_type;
END RECORD;

TYPE array_input_0_Ch IS ARRAY (7 DOWNTO 0) OF record_input_0_Ch;

TYPE record_input_0_Stream IS RECORD
  Ch : array_input_0_Ch;
END RECORD;


TYPE array_input_0_Stream IS ARRAY (23 DOWNTO 0) OF record_input_0_Stream;

TYPE input_0 IS RECORD
  Stream : array_input_0_Stream;
END RECORD;
```

Code Snippet 15: Generated VHDL type of example interface

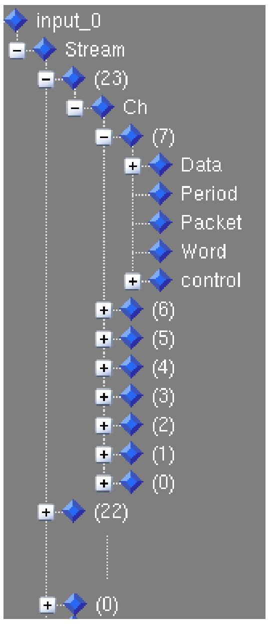


Figure 14: Example interface tree structure

5.4 Derive Component Function

Before generating a VHDL component, it is required to first know what to generate. This is achieved by deriving the function from the input and output interfaces of a SamPy component. This process is done for every component in the system. The objective of this process is to assign a function label to the component. That label is then used to decide which function needs to be generated.

The function of a component is determined by the properties of its input and output interfaces. This process can be seen as a large decision tree. In this decision tree, certain conditions of the interfaces are checked to determine the function. In this prototype, a total of 30 functions can be derived. The conditions for three of these functions are described below. The first function that is explained is the directly connected function, this function is included as an easy example. The other two functions that are mentioned in the subsections below are the *split streams time* and the *merge streams time*. These are included as they are very relevant to one of the research questions. For the remaining functions please see Appendix B: Component functions.

5.4.1 Deriving the Directly Connected Function

The *directly connected* function will simply connect the input interface directly to the output interface. This function is assigned to a component if its input and output interfaces are the same. This is decided if all of the following conditions are met:

- The component only has one input interface and one output interface.
- The static properties of the input interface are identical to those of the output interface.
- The dynamic properties of the input interface are identical to those of the output interface.

An example of a set of interfaces that would result in a directly connected function is shown below.

```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dest”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)])
```

5.4.2 Deriving the Split Stream Time Function

This function splits one stream into multiple streams in the time domain. This means that a certain amount of incoming data will first flow to one output interface and then the remaining data in that time frame will flow to the next output interface. This could be done with two or more output interfaces. This function is assigned if all of the following conditions are met:

- The component has only one input interface.
- The component has more than one output interface.
- The static properties of the input interface and all the output interfaces are identical.
- Only the last dynamic property in the list is being split, just as in the example below.
- The sizes of the time frames at the output interfaces should add up to the size of the time frame in the input interface that is being split.

An example of a set of interfaces that would result in a split time function is shown below. As you can see, the 80 *elm* are split into 10 and 70 for outputs 0 and 1 of the component respectively.

```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)]),
Interface((“comp”,0), “destA”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 10)]),
Interface((“comp”,1), “destB”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 70)])
```

5.4.3 Deriving the Merge Streams Time Function

This function merges multiple streams into 1 stream in the time domain. It is essentially the opposite of the previously described split streams function. In this function there are multiple input interfaces and one output interface. This function is assigned if the following conditions are met:

Conditions:

- The component has more than one input interface.
- The component has only one output interface.
- The static properties of all the input interfaces and the output interface are identical.
- Only the last dynamic property in the list is being merged, just as in the example.
- The sizes of the time frames that are being merged should add up to the size of the time frame in the output interface.

An example of a set of interfaces that would result in a merge time function is shown below. As you can see, the 10 *elm* and the 70 *elm* are merged to create the 80 *elm* at the output.

```
Interface(“srcA”, (“comp”,0), [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 10)]),
Interface(“srcB”, (“comp”,1), [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 70)]),
Interface(“comp”, “dst”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)])
```

5.5 Generating VHDL Component Function

Once a function is assigned to a component, VHDL code can be generated. This is achieved by means of VHDL templates. These templates are Python functions that write certain strings to a VHDL file depending on the properties of the SamPy component. Every component consists of a VHDL entity and a VHDL architecture. Note that not every function that can be derived can also be generated. The total number of generatable functions in this initial prototype is 23 out of the 30 that can be derived.

5.5.1 Generating VHDL Entity

The VHDL entity is generated from the same template for every component. A simplified view of this entity template is shown below in Code Snippet 16. As you can see, it always contains clock and reset inputs of type std_logic. Other ports depend on the input and output interfaces defined in the SamPy description. The input ports are named “sink_in_” followed by the port number of the corresponding interface. Similarly, the output ports are labelled “source_out_” followed by the port number. The type of these ports depend on the corresponding interfaces, these types are defined in the VHDL interface package as described in Section 5.3. This shows that some VHDL code will always be in the file and some code is dependent on the properties of the interfaces.

```
ENTITY SamPy_<component name> IS
  PORT (
          clk : IN STD_LOGIC;
          rst : IN STD_LOGIC;

          --> for every input interface write {
          sink_in_<input port number>     : IN  <VHDL type of the interface>;
          }

          --> for every output interface write {
          source_out_<output port number> : OUT <VHDL type of the interface>;
          }
  );
END SamPy_<component name>;
```
Code Snippet 16: VHDL template for generating Entities

5.5.2 Generating VHDL Architecture

An architecture template is created for every generatable function. These templates work in the same way as the entity template. Some architecture templates make use of an ASTRON IP. Since one of the research questions is related to splitting and merging streams, the implementation of the functions *split streams time* and *merge streams time* are explained in the subsections below. Implementations of all other functions including other functions that split and merge streams can be read in Appendix B: Component functions.

5.5.2.1 Generating the Split Stream Time Function

Splitting streams in the time domain essentially comes down to deciding when an outgoing interface must output the incoming data. To achieve this, the architecture template makes use of the ASTRON dp_split IP. This IP splits one data stream into two according to the configured generics. These generics are first derived from the SamPy interfaces and then applied to the template. Cascading these IPs makes it possible to split one data stream into multiple. This is visualized in Figure 15 below. In the figure it is shown how one input interface is split into three output interfaces. More split IPs are used if more output interfaces are defined. Note that the clock and reset signals are not drawn.

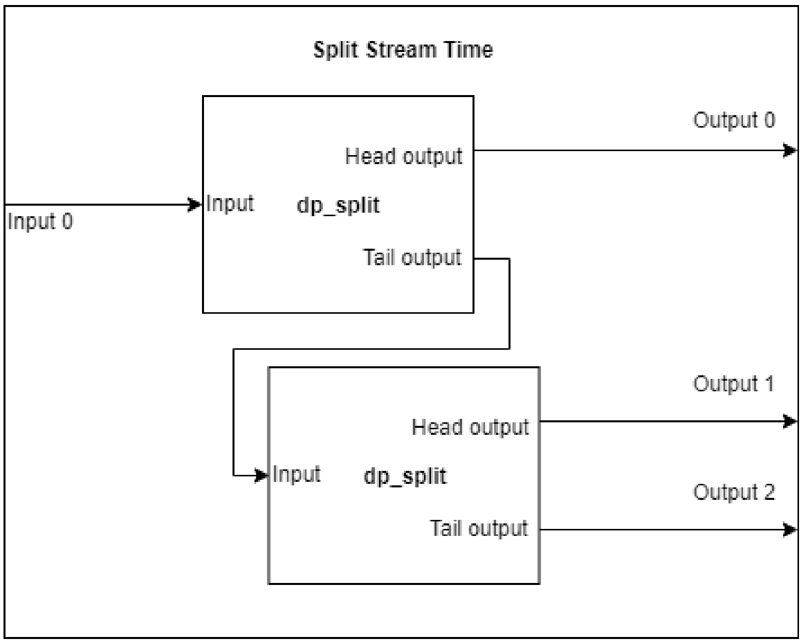


Figure 15: Split Stream Time Function implementation

5.5.2.2 *Generating the Merge Streams Time Function*

The opposite of the previously described split function is merging streams in the time domain. It is necessary to correctly decide when an incoming data stream must transfer its data to the output. This also requires buffering of the incoming data as the multiple incoming streams may not be perfectly aligned due to latency of previous components in the system. This is done by means of FIFOs. These are available as an ASTRON IP and are used in this implementation. The architecture is shown in Figure 16. As you can see, every input first gets buffered by a FIFO. The sizes of these FIFOs are automatically derived from the interface description. It is also possible to set these sizes manually using a SamPy constraint named “fifoSize”. The control process actively reads the data from the FIFOs according to definitions in the input interfaces. For example, if the SamPy description of input interface 0 describes that there are 8 elements in a time frame, the control process will read those 8 elements from FIFO 0 and transfers it to the output. After those 8 elements it outputs the number of elements defined in input interface 1 of FIFO 1, etc. Note that the clock and reset signals are not drawn in the diagram.

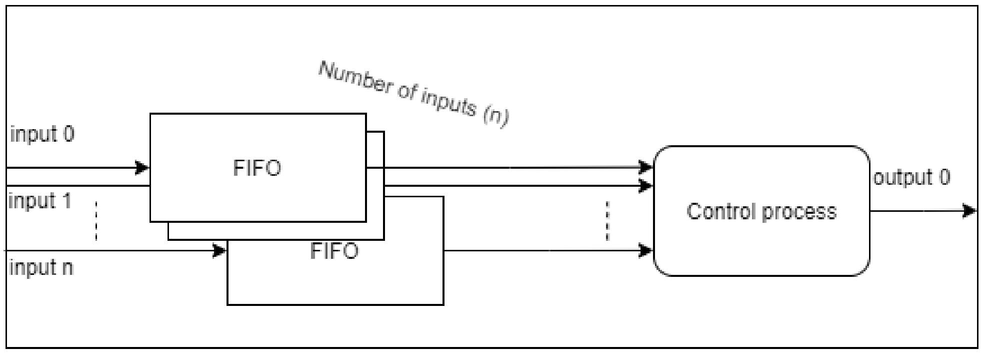


Figure 16: Merge Streams Time Function implementation

5.6 Generating VHDL Top-Level Pipeline

Using the SamPy system description, the VHDL code for the top-level pipeline can be generated. The entity of the pipeline is similar to an entity of a component as described in Section 5.5.1. The architecture connects all the components in the system according to the SamPy system description. A block diagram of the resulting architecture of the example system description below is shown in Figure 17. In this diagram, the clock and reset signals are not drawn. Please see Appendix C: SplitMergeTime VHDL description for the VHDL description of the top-level pipeline.

```
p = ('Period', -1, (300*5, 'ns')) # period defined as constant.
t = (5,'ns') # time of a ‘Word’ = clock period.

splitMergeStream0 = Stream(
  Interface( 'input'   ,  'Split'   , [('Ch', 2)], [('Data', 64)], [p, ('Word', 187,t)]),
  Interface(('Split',2),  'Y'       , [('Ch', 2)], [('Data', 64)], [p, ('Word', 150,t)]),
  Interface( 'Y'       , ('Merge',2), [('Ch', 2)], [('Data', 64)], [p, ('Word', 150,t)]),
  Interface( 'Merge'   ,  'output'  , [('Ch', 2)], [('Data', 64)], [p, ('Word', 187,t)])
)

splitMergeStream1 = Stream(
  Interface(('Split',0), 'X', [('Ch', 2)], [('Data', 64)], [p, ('Word', 7,t)]),
  Interface('X', ('Merge',0), [('Ch', 2)], [('Data', 64)], [p, ('Word', 7,t)])
)

splitMergeStream2 = Stream(
  Interface(('Split',1), ('Merge',1), [('Ch', 2)], [('Data', 64)], [p, ('Word', 30,t)]),
)

splitMergeSys = System("SplitMergeTime", splitMergeStream0, splitMergeStream1,
                                splitMergeStream2)
```

Code Snippet 17: SamPy system definition of the split/merge example

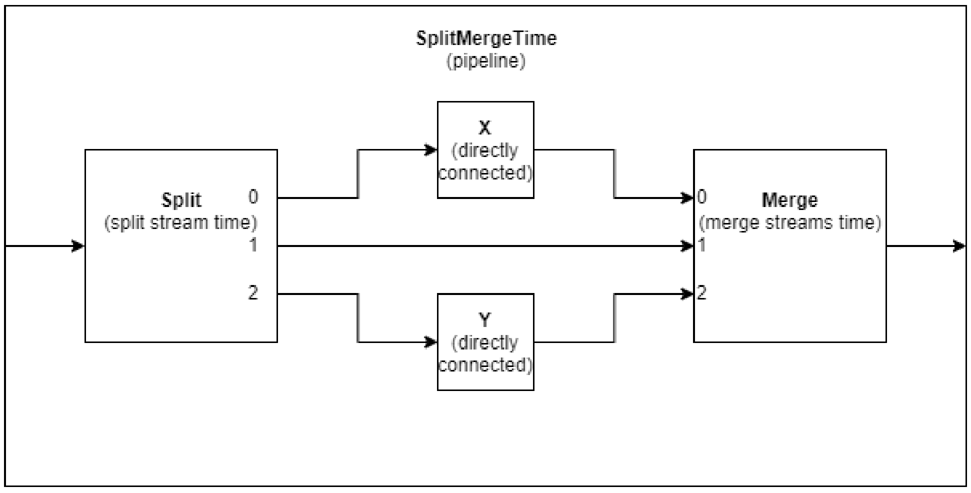


Figure 17: Block diagram of split/merge system

5.7 Generating VHDL Test-bench

A VHDL testbench is generated to provide stimuli for the generated system. The testbench provides the clock and reset signal as well as some counter data for the input interfaces of the system. The counter data is provided by the ASTRON counter IP. The clock will be enabled for a certain amount of time according to the SamPy system definition. This enables the user to easily run a simulation. If the system has no inputs due to an internal data generator, only the clock and reset signals will be provided. Data coming out of the system is not checked in this automatically generated testbench. This is left to the user as the system may contain custom functions that alter the data in ways that is not specified in the SamPy description. A typical testbench that would be generated is shown in Figure 18. As you can see, in this case the pipeline only has one input. In the case that the pipeline has multiple inputs, the counter is automatically configured to generate the same amount of outputs to provide counter data for every system input.

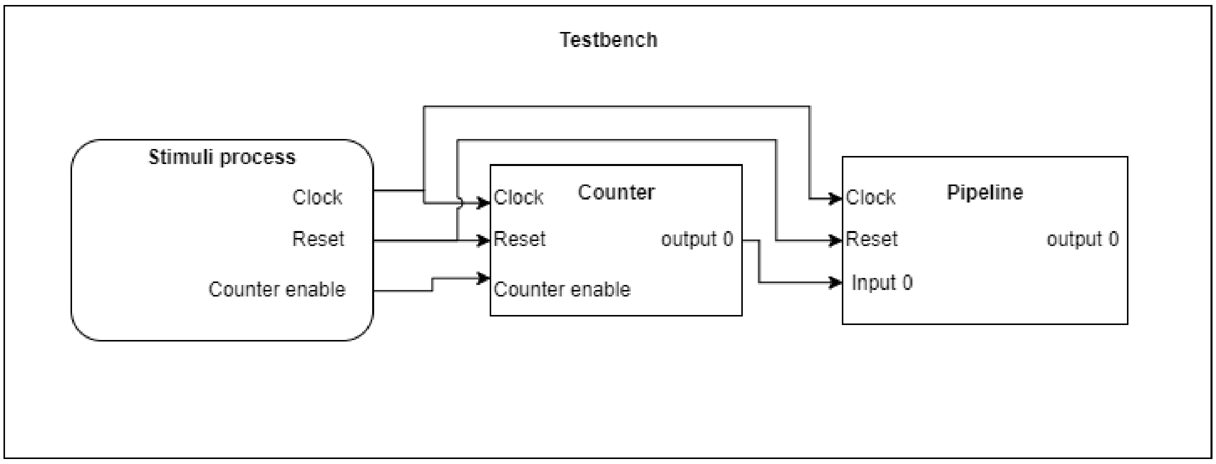


Figure 18: Testbench implementation

6 Results

In this chapter, results from implementing the use cases are discussed. The first subsection shows the results of the video stream implementation. Subsection 6.2 discusses the results of the ASTRON Science Case 3 Input Stage implementation. The SamPy implementation of the Cookie Factory use case is presented in Subsection 6.3.

6.1 Video Stream Implementation

As described in Section 3.2.1, in this use case, it is desired to transform the input interface to the interface required by the processing block. The input interface consists of one 64 bit signal which transfers 62208 packets of 1000 words in one second. The output must have four 8 bit signals which transfer 60 frames of 1080 rows of 1920 pixels. This is accomplished by means of several intermediate interface transformations. This is described in the SamPy description below. A block diagram of this implementation is shown in the Figure 19. As you can see, the complete implementation consists of four interface definitions which results in 3 components. A clock period of 5 nanoseconds is used in this implementation. The output of the SamPy script is explained in Subsection 6.1.1. A simple verification is described in Subsection 6.1.2.

```
from VHDLgen import *
from SamPy import Interface, Stream, System


T = (5, 'ns')
VideoStream = Stream(
  Interface( 'input', 'Repack',
        [ ('Stream', 1)], [ ('Data', 64) ],
        [ ('Period', 120, 1), ('Packet', 62208), ('Element', 1000, T) ]
  ),
  Interface( 'Repack', 'Reframe',
        [ ('Stream', 1)], [ ('Data', 32) ],
        [ ('Period', 120, 1), ('Packet', 62208), ('Element', 2000, T) ]
  ),
  Interface( 'Reframe', 'DeConcat',
        [ ('Stream', 1)], [ ('Data', 32) ],
        [ ('Period', 120, 1), ('Frame', 60), ('Row', 1080), ('Pixel', 1920, T) ]
  ),
  Interface( 'DeConcat', 'output'  ,
        [('Stream', 1)], [ ('Alpha', 8), ('Blue', 8), ('Green', 8), ('Red', 8) ],
        [('Period', 120, 1), ('Frame', 60), ('Row', 1080), ('Pixel', 1920, T) ]
  )
)


VideoSys = System("video_system", VideoStream)
VideoSys.generateBlockdiag('VideoExample.png')
generateSystem(VideoSys)
```

Code Snippet 18: Video stream use case implementation

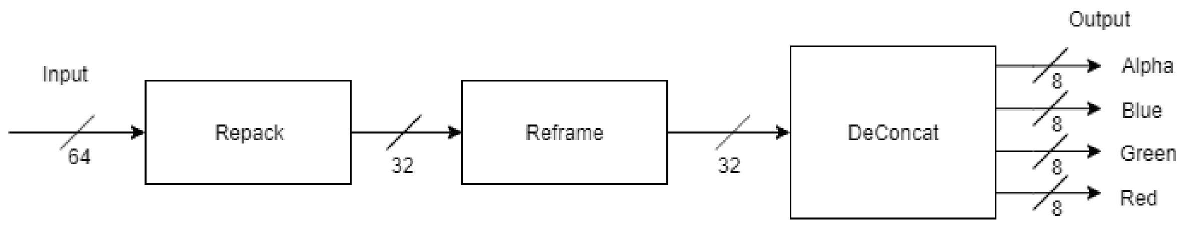


Figure 19: Block diagram of generated VHDL

6.1.1 Script Output

Running the SamPy script gives the user feedback about the described system. This output is shown below. In this case, one warning is created. The warning states that the output transfer speed of the Repack component is doubled with respect to its input. This warning can be justified as the SamPy description states that the interface changes from 1000 elements per packet to 2000 elements per packet. The resulting transfer speed is still below the clock frequency of 200 MHz, an error is therefore not printed.

```
~> python3.4 videoStream.py
UserWarning: Transfer speed of Repack changed from 62208000.00 Element/s to 124416000.00 Element/s
  warnings.warn(message)
```

Derived functions

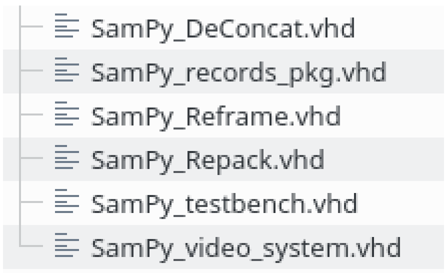
When generating VHDL, the function of each component is first derived. This is shown in the table below. Please see Appendix B: Component functions for the implementation of these functions.

Table 6: Derived functions

Component	Function
Reframe	Reframe_time
Repack	Data_repack
DeConcat	Deconcatenate_signals

Generated VHDL files

The VHDL files that are generated are shown below in Figure 20. As you can see, in addition to a file for each component, a top-level named *SamPy_video_system.vhd* is generated as well as the records package and testbench file.

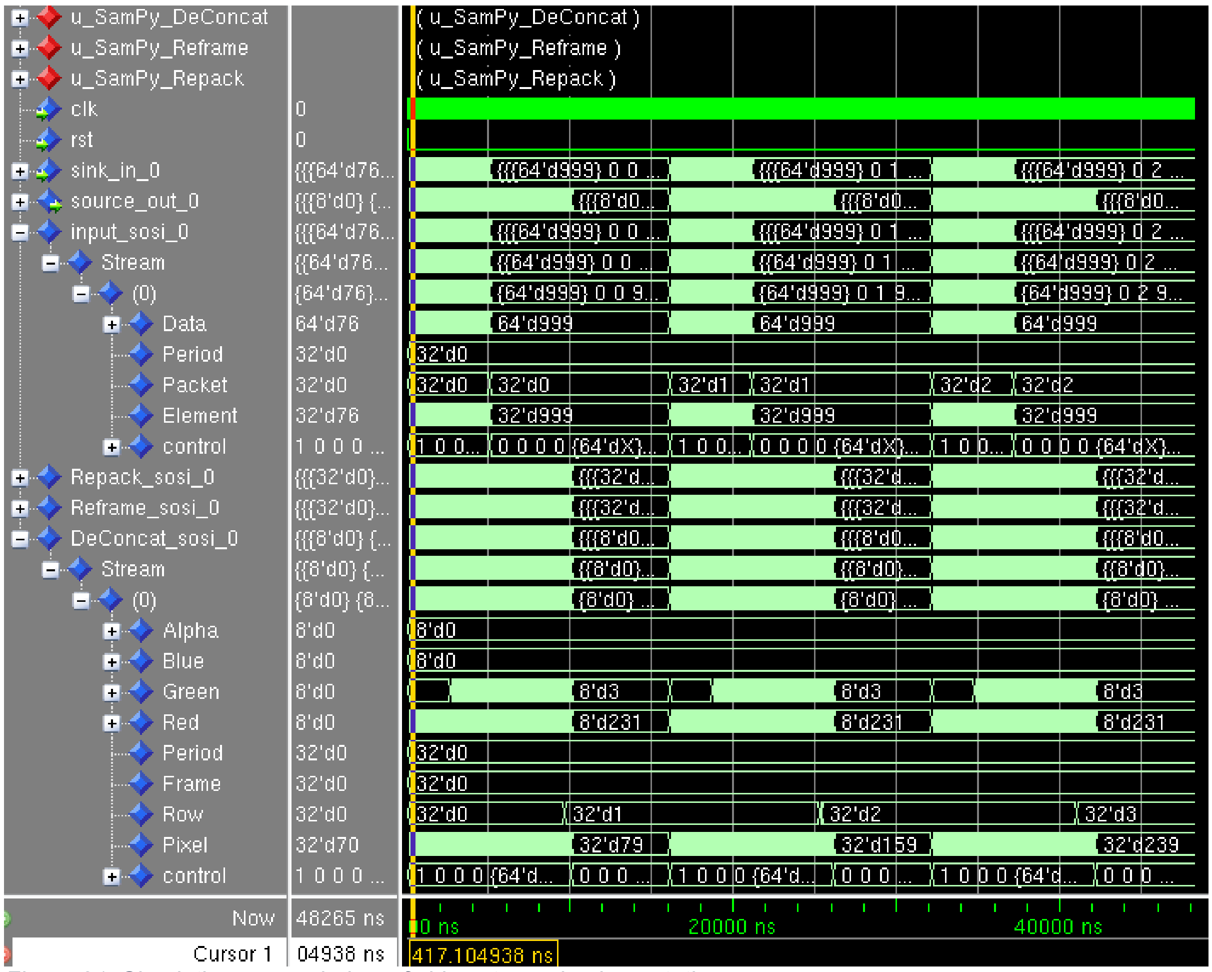


SamPy_DeConcat.vhd
SamPy_records_pkg.vhd
SamPy_Reframe.vhd
SamPy_Repack.vhd
SamPy_testbench.vhd
SamPy_video_system.vhd

Figure 20: Generated VHDL files

6.1.2 Verification

The generated testbench is executed in QuestaSim to verify the behaviour of the generated system. The wave window is shown in Figure 21. The names in the wave window are directly related to the names specified in the SamPy system. The components can be found at the top of the wave window identified by the prefix "u_SamPy_" followed by the component name. The wave window also shows an expanded view of the system input (input_sosi_0) and the output of the DeConcat component (DeConcat_sosi_0). All the static and dynamic properties described in SamPy can directly be related to the signals in the wave window. The dynamic behaviour is checked by making sure that the indices such as Element and Pixel have the correct values. The data fields are checked by making sure their contents are as expected.



```
u_SamPy_DeConcat
u_SamPy_Reframe
u_SamPy_Repack
clk                 0
rst                 0
sink_in_0           {{{64'd76...
source_out_0        {{{8'd0} {...
input_sosi_0        {{{64'd76...
  Stream            {{64'd76...
    (0)             {64'd76}...
      Data          64'd76
      Period        32'd0
      Packet        32'd0
      Element       32'd76
      control       1 0 0 0 ...
Repack_sosi_0       {{{32'd0}...
Reframe_sosi_0      {{{32'd0}...
DeConcat_sosi_0     {{{8'd0} {...
  Stream            {{8'd0} {...
    (0)             {8'd0} {8...
      Alpha         8'd0
      Blue          8'd0
      Green         8'd0
      Red           8'd0
      Period        32'd0
      Frame         32'd0
      Row           32'd0
      Pixel         32'd70
      control       1 0 0 0 ...
Now                 48265 ns
Cursor 1            04938 ns
```

Figure 21: Simulation wave window of video stream implementation

6.2 ASTRON Science Case 3 Input Stage Implementation

The input stage of the ASTRON Science Case 3 design is described in a SamPy script. The description is based on the original VHDL implementation. The complete script can be found in Appendix D: ASTRON Science Case 3 Input Stage SamPy Script. A block diagram of this script is shown below in Figure 22. The design contains one custom function named *merge_custom*. This component assigns the data fields of port 1 to the ASTRON control signals of the interface on port 0. The resulting interface is the output of that component. The output of the SamPy script is shown in Subsection 6.2.1. The system is verified by placing it in the original design. This is further explained in Subsection 6.2.2.

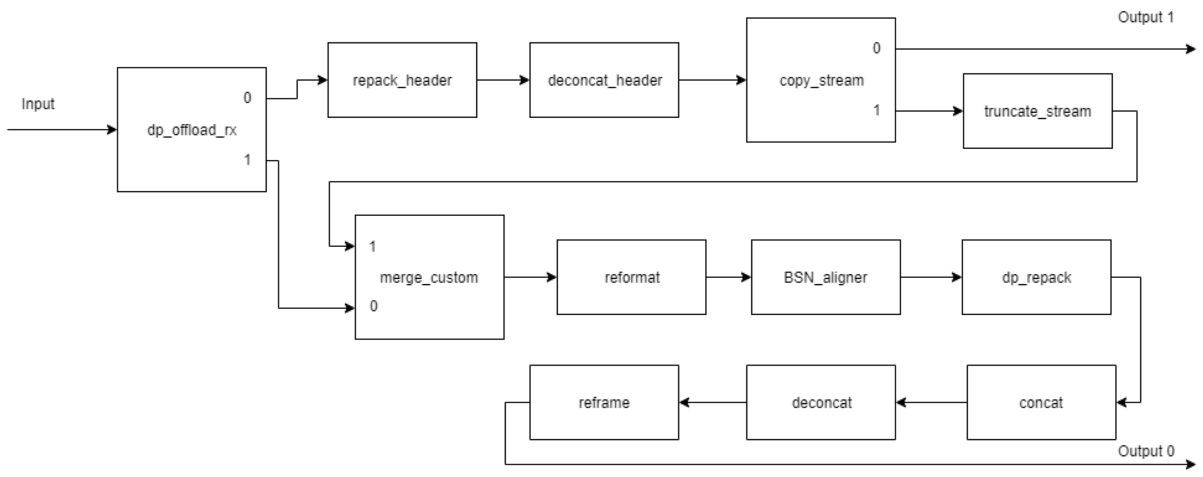


Figure 22: Block diagram of Science Case 3 implementation

6.2.1 Script Output

Running the SamPy script gives the user feedback as shown below. A total of 7 warnings are printed. Al these warnings can be justified. The first two warnings are caused by repacking data, this changes the transfer speed but not the transfer rate. The copy_stream component causes warnings about the total output transfer speed and transfer rate being doubled. This is of course due to the duplication of the interface. The merge_custom component makes some data disappear by assigning it to the ASTRON control signals. This will lower the transfer rate and transfer speed as stated in the warnings. The last warning states that the truncate _stream component lowers the transfer rate. This is caused by the truncation of signals.

```
~> python3.4 SC3_input_stage.py
UserWarning: Transfer speed of repack_header changed from 5468750.00 Word/s to 781250.00 Word/s
UserWarning: Transfer speed of dp_repack changed from 140625000.00 Word/s to 187500000.00 Word/s
UserWarning: The output transfer rate of copy_stream is higher than its input transfer rate,
increased by 100.00%
UserWarning: Transfer speed of copy_stream changed from 781250.00 Word/s to 1562500.00 Word/s
UserWarning: The output transfer rate of merge_custom is lower than its input transfer rate,
decreased by 0.56%
UserWarning: Transfer speed of merge_custom changed from 141406250.00 Word/s to 140625000.00 Word/s
UserWarning: The output transfer rate of truncate_stream is lower than its input transfer rate,
decreased by 85.49%
```

Derived functions

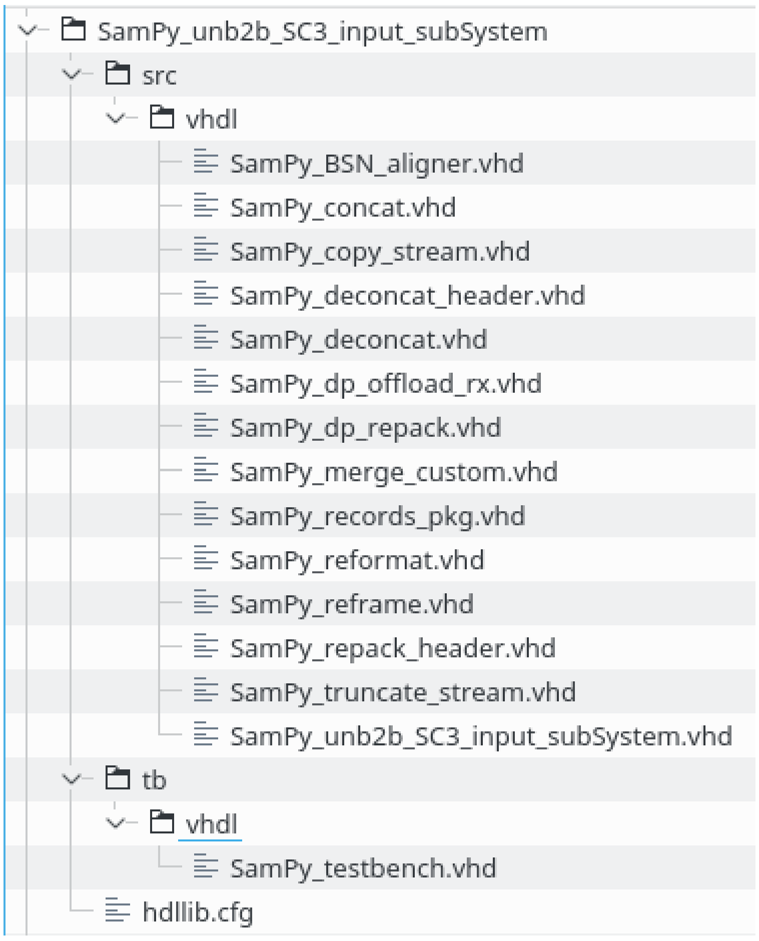
The functions that are derived for each component are shown in Table 7. Please see Appendix B: Component functions for the implementation of each function.

Table 7: Derived functions

Component	Function
BSN_aligner	align_instances
concat	concatenate_space
copy_stream	duplicate_stream
deconcat	deconcatenate_space
deconcat_header	deconcatenate_signals
dp_offload_rx	split_into_2_streams_time
dp_repack	data_repack
merge_custom	custom_function
reformat	deconcatenate_instances
reframe	reframe_time
repack_header	data_repack
truncate_stream	truncate_space

Generated VHDL files

This script makes use of the *genRadioHDLProject* function. This function generates an extra configuration file (hdllib.cfg) for the project infrastructure used at ASTRON. Additionally, the generated VHDL files are placed in dedicated source and testbench directories. This enables the user to use the ASTRON tools to perform simulation and synthesis by means of commands. The files that are generated are shown below in Figure 23.



```
SamPy_unb2b_SC3_input_subSystem
    src
        vhdl
            SamPy_BSN_aligner.vhd
            SamPy_concat.vhd
            SamPy_copy_stream.vhd
            SamPy_deconcat_header.vhd
            SamPy_deconcat.vhd
            SamPy_dp_offload_rx.vhd
            SamPy_dp_repack.vhd
            SamPy_merge_custom.vhd
            SamPy_records_pkg.vhd
            SamPy_reformat.vhd
            SamPy_reframe.vhd
            SamPy_repack_header.vhd
            SamPy_truncate_stream.vhd
            SamPy_unb2b_SC3_input_subSystem.vhd
    tb
        vhdl
            SamPy_testbench.vhd
    hdllib.cfg
```

Figure 23: Generated VHDL

6.2.2 Verification

Since the generated system should have the same behaviour as the original, it could be verified by replacing the original input stage by this generated system in the original Science Case 3 design. It is required to use some simple glue logic to translate the ASTRON interfaces to the SamPy interfaces to be able to place the generated input stage in the design. After replacing the input stage, the self-checking testbench of Science Case 3 is executed. This testbench returned a positive result.

6.3 Cookie Factory Implementation

The SamPy description of the cookie factory use case is given in Code Snippet 19. The system consists of 4 streams. Stream 0 describes the main stream from Supplier to Transport. Stream 1 and 2 describe additional streams between Supplier and Mixer for the transport of butter and sugar. Stream 3 describes the stream from Supplier to Chipper to Mixer. This description is directly related to the block diagram shown in Figure 24. Something that is not included in the description is the latency caused by the Chipper. When starting the system, the supplier will output the ingredients every minute. After one minute, the flour, butter and sugar will arrive at the mixer. However, the first pile of chocolate chips will arrive 2 minutes after the start of the system. This can be solved in several ways. One could add a constraint to the Supplier to delay outputs 0, 1 and 2 for 1 minute at start up. Another way is to place a buffer between the Supplier and the Mixer to synchronise the streams. This result shows that SamPy can be used to describe more abstract systems. A script containing such a system can still be executed. The resulting output is shown in the subsection below.

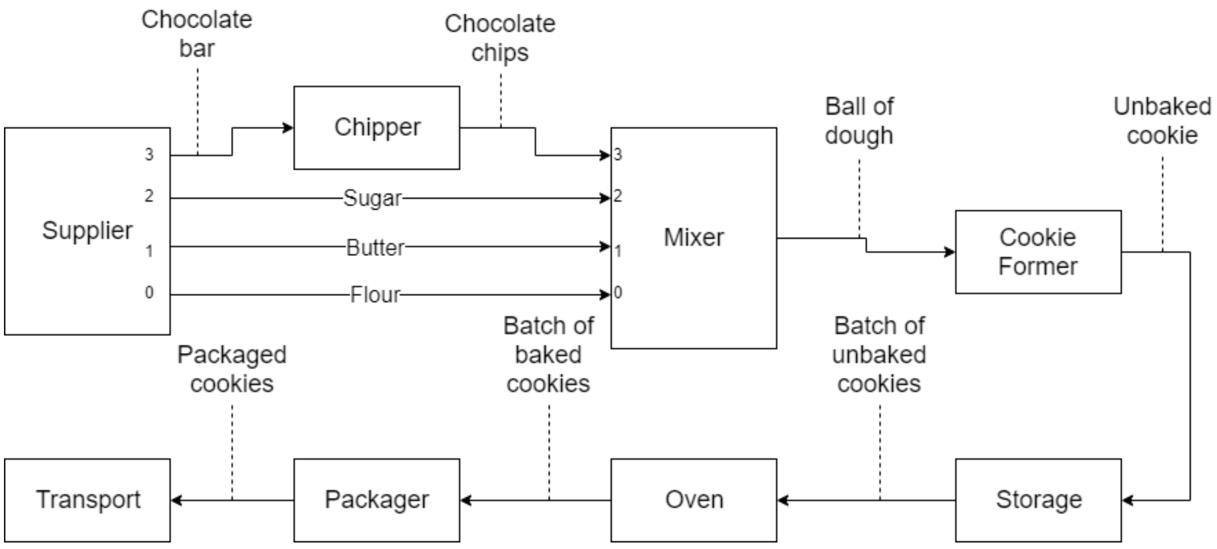


Figure 24: Block diagram of the Cookie Factory implementation

```
Stream0 = Stream(
  Interface( ('Supplier',0), ('Mixer',0),
        [ ('Factory', 1) ], [ ('Conveyor',  1) ],
        [ ('Period', -1, (1, 'm')) , ('pile', 1, (1,'s')), ('grams of flour', 400) ] ),

  Interface( 'Mixer', 'Former',
        [ ('Factory', 1) ], [ ('Conveyor',  1) ],
        [ ('Period', -1, (1, 'm')) , ('Ball of dough', 24) ] ),

  Interface( 'Former', 'Storage',
        [ ('Factory', 1) ], [ ('Conveyor',  1) ],
        [ ('Period', -1, (1, 'm')) , ('Unbaked Cookie', 24) ] ),

  Interface( 'Storage', 'Oven',
        [ ('Factory', 1) ], [ ('Conveyor', 12) ],
        [ ('Period', -1, (20, 'm')), ('Unbaked Cookie Batch', 40) ] ),

  Interface( 'Oven', 'Packager',
        [ ('Factory', 1) ], [ ('Conveyor', 12) ],
        [ ('Period', -1, (20, 'm')), ('Baked Cookie Batch', 40) ] ),

  Interface( 'Packager', 'Transport',
        [ ('Factory', 1) ], [ ('Conveyor',  1) ],
        [ ('Period', -1, (1, 'm')), ('Packaged Cookies', 2) ] )
)

Stream1 = Stream(
  Interface( ('Supplier',1), ('Mixer',1),
        [ ('Factory', 1) ], [ ('Conveyor',  1) ],
        [ ('Period', -1, (1, 'm')), ('pile', 1, (1,'s')), ('grams of butter', 225) ] )
)

Stream2 = Stream(
  Interface( ('Supplier',2), ('Mixer',2),
        [ ('Factory', 1) ], [ ('Conveyor',  1) ],
        [ ('Period', -1, (1, 'm')) , ('pile', 1, (1,'s')), ('grams of sugar',  350) ] )
)

Stream3 = Stream(
  Interface( ('Supplier',3), 'Chipper'  ,
        [('Factory', 1)], [ ('Conveyor',  1) ],
        [ ('Period', -1, 1, 'm'), ('Chocolate Bar', 2) ] ), # bars

  Interface( 'Chipper'  , ('Mixer',3),
        [('Factory', 1)],[ ('Conveyor',  1) ],
        [ ('Period', -1, 1, 'm'), ('pile', 1, (1,'s')), ('grams of Chocolate chips',  200) ] )
)

cookieFactory = System("cookieFactory", Stream0, Stream1, Stream2, Stream3)
cookieFactory.generateBlockdiag('cookie.png', ".")
```

Code Snippet 19: SamPy implementation of the Cookie Factory use case

6.3.1 Script Output

Running the script gives the user feedback about certain properties that are altered throughout the interfaces. The output of this script is shown below. In this case, all warnings can be justified. The first warning is about the lowered transfer rate of the Packager. Since the Packager packages 12 Cookies in 1 package, the transfer rate is decreased to 1/12 which is a total decrease of 91,67%. The 2nd and 3rd warning can be justified by the 2 chocolate bars transformed into the 200 chocolate chips. Warnings 4 and 5 can be explained by the ingredients being in grams and the Mixer transforms it into balls of dough. The last warning is justified by the change of unit from unbaked cookie to unbaked cookie batch. One batch is 12 cookies, therefore the transfer speed changes from 0.4 cookies per second to 0.4 / 12 = 0.03 batches per second.

```
1. UserWarning: The output transfer rate of Packager is lower than its input transfer rate,
   decreased by 91.67%
2. UserWarning: The output transfer rate of Chipper is higher than its input transfer rate,
   increased by 9900.00%
3. UserWarning: Transfer speed of Chipper changed from 0.03 Chocolate Bar/s to 3.33 Chocolate
   chips/s
4. UserWarning: The output transfer rate of Mixer is lower than its input transfer rate, decreased
   by 97.96%
5. UserWarning: Transfer speed of Mixer changed from 19.58 [butter, Chocolate chips, sugar, flour]/s
   to 0.40 Ball of dough/s
6. UserWarning: Transfer speed of Storage changed from 0.40 Unbaked Cookie/s to 0.03 Unbaked Cookie
   Batch/s
```

7 Conclusion

The main objective is the development of the first working prototype of SamPy and the creation of the VHDL generator. SamPy is a Python module that enables the user to describe streaming systems by means of interface definitions. SamPy enables an organized way of describing streaming systems. This makes it possible to describe large detailed streaming systems but also more abstract or high-level systems. This way of describing systems also enables easier communication about large streaming systems. SamPy description also relate most, if not all, important system properties to the SamPy interface definitions. The VHDL generator generates VHDL from a SamPy system description. This makes the interface development less time consuming. Describing a SamPy system using intermediate interfaces in enough detail makes it possible to derive and generate the necessary functions. With this work, the research questions can be answered. Each research question is answered below.

1. What information should be contained in one interface definition to describe all of its necessary static and dynamic properties?
All the necessary information of an interface can be defined in a SamPy interface. A SamPy interface describes: the source and destination of the interface, the physical properties, and the behaviour of elements in time.

2. What interface altering operations need to be derivable from a set of interface definitions and how can this be achieved?
Describing a SamPy system using intermediate interfaces in enough detail makes it possible to derive the necessary functions in a system. This is achieved by checking every component in the SamPy system for certain conditions. This process assigns a function to the concerned component. The operations that need to be derivable are relatively simple operations as presented in this work.

3. Once the operations are derived, how can they best be realized in a VHDL description?
Once it is known which function a component must have, a VHDL file is generated with the use of a template. A template is a python function that writes certain VHDL code to the file depending on the interface properties of the component. Some templates can be very simple such as the implementation of a direct connection. Other functions may be a bit more advanced such as splitting and merging streams involving buffering and external IPs. In addition to all the component files, a top-level pipeline, type package, and testbench file are generated to complete the generation of the system in VHDL.

4. How to handle multiple streams (multiple sets of interface definitions) and splitting or merging them?
The SamPy syntax supports multiple inputs and outputs of a component. Splitting and merging streams are performed by generatable functions. These functions are a bit more advanced involving buffering and external IPs.

5. In what way and to what extent can the soundness of a described system be evaluated in an early stage?
Any system described with SamPy is automatically evaluated when the script is executed. This evaluation provides the user with feedback about the soundness of a described system. This feedback will show the user mistakes or possible unwanted behaviour in the system.

8 Recommendations

Several recommendations for future work is discussed in this chapter. Recommendations for the SamPy module is discussed in Subsection 8.1. The recommendations for the VHDL generator is presented in Subsection 8.2.

8.1 SamPy recommendations

The current prototype of SamPy is able to generate a very basic block diagram. This block diagram can be useful. However, for more complex systems the generated block diagram is not nicely formatted causing poor readability. In future work it may be desired to be able to generate a detailed, well-structured block diagram. In addition, documentation about the properties of the system may be useful to be generatable. This may improve the ability of communicating about the system even further.

Another useful future feature of SamPy could be a way of simulating the system at Python level. This would involve generating data and passing it through several operations to see whether it creates the desired output. The idea is that this makes it possible to verify the systems behaviour very early, helping improve the development time.

8.2 VHDL generator recommendations

The testbench of the current VHDL generator is relatively simple as it does not verify the output of the system. It could be desired to have a fully self-checking VHDL testbench that generates data and verifies the output of the system. The challenge here is when a system makes use of custom functions implemented by the user as its behaviour is not known by the VHDL generator. This could be solved by either describing the behaviour of the custom function such that the testbench can anticipate on that. Or the custom function should be excluded by replacing the custom function by generated data.

To improve on overhead an advanced mechanism could be built that keeps track of all the latencies in the system to decide whether buffering is needed in certain components. The current version assumes that some buffering is always needed and a minimum buffer size is used.

To improve development time even further, it may be desired to make more function templates. This would make it possible to generate more functions.

Another feature that may be desired is to support the generation of multi-clock systems.

References

[1] IBM Research, “Stream Processing Languages and Abstractions,” Springer International Publishing, 2018.

[2] A. Arasu, S. Babu and J. Widom, “The CQL Continuos Query Language: Semantic Foundations and Query Execution,” *J Very Large Data Bases (VLDB J),* vol. 15, no. 2, pp. 121-142, 2006.

[3] P. Caspi, D. Pilaud, N. Halbwachs and J. A. Plaice, “LUSTRE: a declarative language for real-time programming.,” *Symposium on principles of programming languages (POPL),* pp. 178-188, 1987.

[4] M. Hirzel, S. Schneider and B. Gedik, “SPL: an extensible language for distributed stream processing.,” *Trans Program Lang Syst (TOPLAS),* vol. 39, no. 1, pp. 5:1-5:39, 2017.

[5] M. Arnold, D. Grove, B. Herta, M. Hind, M. Hirzel, A. Iyengar, L. Mandel, V. Saraswat, A. Shinnar, J. Siméon, M. Takeuchi, O. Tardieu and W. Zhang, “META: middleware for events, transactions, and analytics.,” *IBM J Res Dev,* vol. 60, no. 2-3, pp. 15:1-15:10, 2016.

[6] W. Luk and S. McKeever, “Pebble: A Language For Parametrised and Reconfigurable Hardware Design,” *LNCS,* no. 1482, pp. 9-18, 1998.

[7] M. A. Franklin, E. J. Tyson, J. Buckley, P. Crowley and J. Maschmeyer, “Auto-pipe and the X language: A Pipeline Design Tool and Description Language,” in *Proceedings 20th IEEE International Parallel & Distributed Processing*, Rhodes Island, Greece, 2006.

[8] J. G. Wingbermuehle, R. D. Chamberlain and R. K. Cytron, “ScalaPipe: A Streaming Application Generator,” in *IEEE 20th International Symposium on Field-Programmable Custom Computing Machines*, Toronto, ON, Canada, 2012.

[9] D. van der Schuur, “Stream Oriented Design with SamPy,” ASTRON, Dwingeloo, Netherlands, 2018.

[10] E. Kooistra, “Representing Streaming Data,” ASTRON-RP-1486, Dwingeloo, Netherlands, 2015.

[11] IEEE, “Behavioural Languages - Part 1-1: VHDL Language Reference Manual,” *IEEE 1076 IEC 61691-1-1 First edition 2004-10,* pp. 1-308, 15 Nov 2004.

Appendix A: SamPy BNF Notation

The BNF notation is for reference only as SamPy uses the Python syntax. Python enables the use of variables which is not described in this notation.

```
<system>          ::= "System(" <string> "," <streams> ")" |
                      "System(" <string> "," <literal> "," <streams> ")" |
                      "System(" <streams> ")"

<streams>         ::= <stream> | <streams> "," <stream>
<stream>          ::= "Stream(" <interfaces> ")"
<interfaces>      ::= <interface> | <interfaces> "," <interface>
<interface>       ::= "Interface(" <source> "," <sink> "," <instances> ","
                                   <signals> "," <dynamic> ")"

<source>          ::= <string> | <tuple> | <literal>
<sink>            ::= <string> | <tuple> | <literal>
<instances>       ::= "[" <tuples> "]"
<signals>         ::= "[" <tuples> "]"
<dynamic>         ::= "[" <timeframes> "]"

<tuples>          ::= <spec-tuple> | <tuples> "," <spec-tuple>
<timeframes>      ::= "(" <string> "," <spec-integer> "," <time> ")" |
                      <timeframes> <spec-tuple>
<tuple>           ::= "(" <string> "," <integer> ")"
<spec-tuple>      ::= <tuple> | "(" <string> "," <integer> "," <time> ")"

<time>            ::= <float> | "(" <float> "," <time-string> ")"
<float>           ::= <integer> | <integer> "." <integer>
<time-string>     ::= "'" <time-unit> "'" | '"' <time-unit> '"'
<time-unit>       ::= "fs" | "ps" | "ns" | "us" | "ms" | "s" | "m" | "h" | "d"

<character>       ::= <letter> | <digit> | "_"
<literal>         ::= <letter> | "_" | <literal> <character>
<string>          ::= "'" <literal> "'" | '"' <literal> '"'

<digit>           ::= "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9"
<integer>         ::= <digit> | <integer> <digit>
<spec-integer>    ::= "-1" | <integer>

<letter>          ::= "A" | "B" | "C" | "D" | "E" | "F" | "G" | "H" | "I" | "J" |
                      "K" | "L" | "M" | "N" | "O" | "P" | "Q" | "R" | "S" | "T" |
                      "U" | "V" | "W" | "X" | "Y" | "Z" |
                      "a" | "b" | "c" | "d" | "e" | "f" | "g" | "h" | "i" | "j" |
                      "k" | "l" | "m" | "n" | "o" | "p" | "q" | "r" | "s" | "t" |
                      "u" | "v" | "w" | "x" | "y" | "z"
```

Appendix B: Component functions

This appendix contains a description of every derivable function. It includes an example of a set of interfaces that would result in that function. It also includes the conditions that need to be met for it to classify as that function. If the function is also generatable, the implementation of that function is briefly explained.

Some conditions rely on definitions derived from a SamPy interface, these are presented below. In the function descriptions, the definitions belonging to an input interface are indicated by the subscript *in*. Similarly, the definitions belonging to an output interface are indicated by the subscript *out*.

All the matrices in the definitions below represent the properties of a SamPy interface. The rows of these matrices are of the format [<String>, <Int>, <Time>] just as in the SamPy interface attributes. Please see the description of the *Directly connected* function described in the first subsection of this appendix for an example.

L_X	Number of instances in a SamPy interface.
L_Y	Number of signals in a SamPy interface.
L_S	Number of static properties in a SamPy interface = $L_X + L_Y$
L_D	Number of dynamic properties in a SamPy interface.

n	Integer, total number of ports
X	L_X-by-3 matrix, instances properties of a SamPy interface.
Y	L_Y-by-3 matrix, signals properties of a SamPy interface.
S	L_S -by-3 matrix,all static properties of a SamPy interface = $\begin{bmatrix} X \\ Y \end{bmatrix}$
D	L_D-by-3 matrix, dynamic properties of a SamPy interface.
p	Period in seconds of the largest time frame.
q	Total number of signal lines (bits) for one interface =

$$\left(\sum_{i=1}^{L_Y} Y_{i,2}\right) * \prod_{i=1}^{L_X} X_{i,2}$$

Equation 2: Calculation of the total number of signal lines (q)

r number of elements per second =

$$\frac{\prod_{i=2}^{L_D} D_{i,2}}{p}$$

Equation 3: Calculation of the number of elements per second (r)

Directly connected

The directly connected function will connect the input interface directly to the output interface. This function is used when the input interface is identical to the output interface.

Example

```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dest”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)])
```

As this is the first example, the properties mentioned above are described below according to the example interface.

$L_X = 1$
$L_Y = 1$
$L_S = 2$
$L_D = 2$

$n_{out} = n_{in} = 1$
$X_{out} = X_{in} = \begin{bmatrix} "ch" & 3 & 0 \end{bmatrix}$
$Y_{out} = Y_{in} = \begin{bmatrix} "data" & 8 & 0 \end{bmatrix}$
$S_{out} = S_{in} = \begin{bmatrix} "ch" & 3 & 0 \\ "data" & 8 & 0 \end{bmatrix}$
$D_{out} = D_{in} = \begin{bmatrix} "p" & -1 & 1 \\ "elm" & 80 & 0.0125 \end{bmatrix}$
$p_{out} = p_{in} = 1$
$q_{out} = q_{in} = (8) * 3 = 24$
$r_{out} = r_{in} = \frac{80}{1} = 80$

Conditions

- $n_{out} = n_{in} = 1$
- $S_{out} = S_{in}$
- $D_{out} = D_{in}$

Implementation

The implementation of this function is simply a direct connection between the input interface and the output interface.

Transpose space

This function will transpose two instances. It is described as two instances, which are next to each other, being swapped.

Example

```
Interface(“src”, “comp”, [(“A”, 3), (“B”, 2), (“C”, 6)], [(“data”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dest”, [(“A”, 3), (“C”, 6), (“B”, 2)], [(“data”, 8)],
                          [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $S_{out} \neq S_{in}$
- $D_{out} = D_{in}$
- $Y_{out} = Y_{in}$
- $L_{S-out} = L_{S-in}$
- $q_{out} = q_{in}$
- The instances that are being swapped must be next to each other in the instances list, just like in the example above

Transpose time

This function will transpose two dynamic properties. It is described as two dynamic properties, which are next to each other, being swapped.

Example

```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)],
                         [(“p”, -1, 1), (“A”, 80), (“B”, 100)]),
Interface(“comp”, “dest”, [(“ch”, 3)], [(“data”, 8)],
                          [(“p”, -1, 1), (“B”, 100), (“A”, 80)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $S_{out} = S_{in}$
- $L_{D-out} = L_{D-in}$
- $D_{out} \neq D_{in}$
- $r_{out} = r_{in}$
- The dynamic properties that are being swapped must be next to each other in the dynamic properties list, just like in the example above.

Reframe space

This function will reframe instances.

Example

```
Interface(“src”, “comp”, [(“A”, 3), (“B”, 2), (“C”, 6)], [(“data”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dest”, [(“A”, 3), (“B”, 3), (“C”, 4)], [(“data”, 8)],
                          [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $S_{out} \neq S_{in}$
- $D_{out} = D_{in}$
- $Y_{out} = Y_{in}$
- $L_{S-out} = L_{S-in}$
- $q_{out} = q_{in}$
- It cannot be classified as a transpose.

Reframe time

This function will reframe dynamic properties.

Example

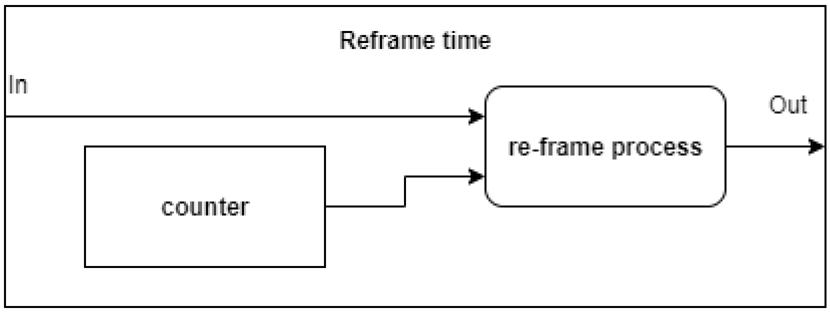
```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)],
                         [(“p”, -1, 1), (“A”, 80), (“B”, 100)]),
Interface(“comp”, “dest”, [(“ch”, 3)], [(“data”, 8)],
                          [(“p”, -1, 1), (“A”, 10), (“B”, 800)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $S_{out} = S_{in}$
- $D_{out} \neq D_{in}$
- $r_{out} = r_{in}$
- It cannot be classified as: a transpose time, a concatenate time, or a de-concatenate time.

Implementation

The implementation of this function is visualized in the figure below. A re-frame process merges the incoming interface with new indices created by the counter. These new indices correspond with the dynamic properties of the output interface.



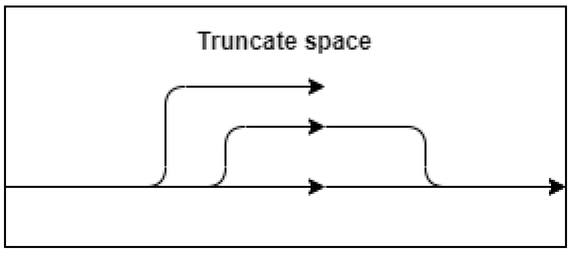
Truncate space

Truncate space will only output part of the incoming data while keeping the dynamic properties the same. This lowers the data rate.

Example
```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dest”, [(“ch”, 2)], [(“data”, 6)], [(“p”, -1, 1), (“elm”, 80)])
```

Conditions
- $n_{out} = n_{in} = 1$
- $D_{out} = D_{in}$
- $q_{out} < q_{in}$

Implementation
The implementation is shown in the figure below. As you can see, only part of the signals in the incoming interface are connected to the outgoing interface. This results in a loss of data in the space domain.



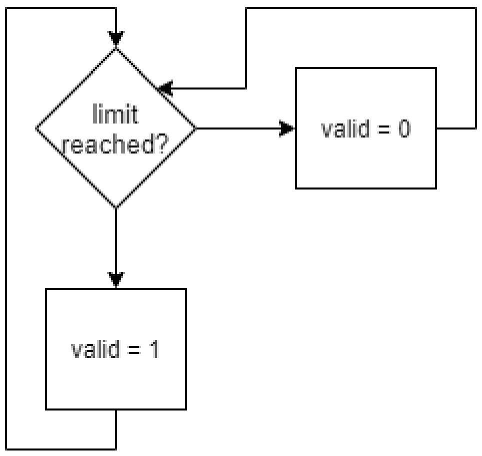
Truncate time

Truncate time will only output part of the incoming data while keeping the static properties the same. This lowers the data rate.

Example
```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dest”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 50)])
```

Conditions
- $n_{out} = n_{in} = 1$
- $S_{out} = S_{in}$
- $r_{out} < r_{in}$

Implementation
To truncate the data in time, this function sets the valid flag of the control record to zero when the limit for that time frame is reached. This limit is derived from the output interface as it is defined in the dynamic properties. In the example above, the first 50 “elm” would be valid but the remaining 30 will be invalid. This process is shown in the figure below.



Data repack

The data repack function repacks the incoming data in either more smaller elements or less larger elements.

Example

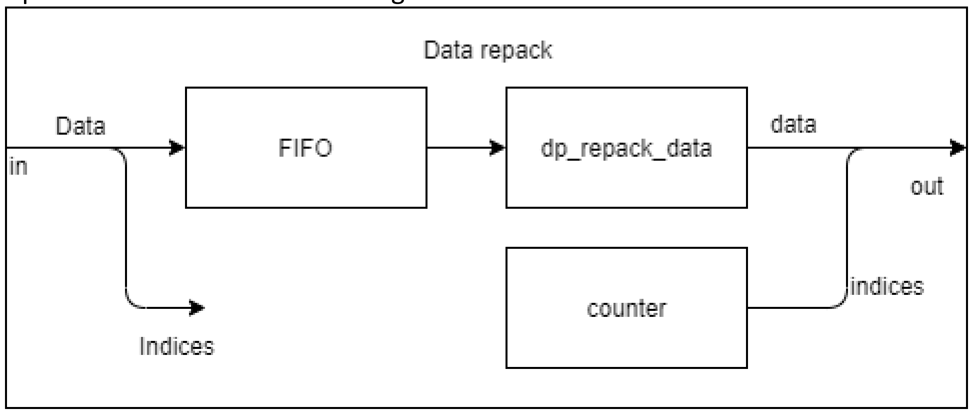
```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dst”, [(“ch”, 3)], [(“data”, 32)], [(“p”, -1, 1), (“elm”, 20)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $X_{out} = X_{in}$
- $L_{Y\_out} = L_{Y-in} = 1$
- $\frac{q_{out}}{q_{in}} = \frac{r_{in}}{r_{out}} \neq 1$
- Only the last dynamic property in the interface has changed, just as in the example above.

Implementation

The data repack implementation makes use of an ASTRON IP called “dp_repack_data”. This IP uses backpressure signals, therefore a FIFO is needed to buffer incoming data. Since the output interface has different dynamic properties, the indices in the interface are re-generated by a counter. This implementation is shown in the figure below.



Generate time

This function generates data in the time domain.

Example

```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dest”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 99)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $S_{out} = S_{in}$
- $r_{out} > r_{in}$
- $L_{D\_out} = L_{D\_in}$

Generate space

This function generates data in the space domain.

Example

```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dest”, [(“ch”, 4)], [(“data”, 9)], [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $D_{out} = D_{in}$
- $q_{out} > q_{in}$
- $L_{X\_out} = L_{X\_in}$

Concatenate signals

This function concatenates multiple signals into 1 larger signal.

Example

```
Interface(“src”, “comp”, [(“ch”, 3)], [(“A”, 8), (“B”, 16), (“C”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dst”, [(“ch”, 3)], [(“data”, 24), (“C”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $D_{out} = D_{in}$
- $q_{out} = q_{in}$
- $X_{out} = X_{in}$
- $L_{Y\_out} < L_{Y\_in}$

Implementation

The implementation of concatenating signals is pretty straightforward. Since all signals are translated to std_logic_vector in VHDL, the signals that are being concatenated simply will be partly assigned to the outgoing signal. So in the example above, “A” will be assigned to “data” 23 down to 16 and “B” will be assigned to “data” 15 down to 0.

Concatenate instances

This function concatenates multiple instances into 1 larger instance.

Example

```
Interface(“src”, “comp”, [(“A”, 2), (“B”, 3), (“C”, 4)], [(“data”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dst”, [(“ch”, 6), (“C”, 4)], [(“data”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $D_{out} = D_{in}$
- $q_{out} = q_{in}$
- $Y_{out} = Y_{in}$
- $L_{X\_out} < L_{X\_in}$

Implementation

Concatenating instances makes use of the way the instances are defined in the VHDL interface package. This makes it possible to simply assign the incoming instances to the outgoing instance. In the example above, we have 2 (“A”) times 3 (“B”) incoming instances. This is then assigned to the “ch ” instance by correct indexing. So, index [A0, B0, C0] to [ch0, C0] all the way to [A1, B2, C3] re-indexed to [ch5,C3].

Concatenate space

This function concatenates all signals and 1 or more instances into one large signal.

Example

```
Interface(“src”, “comp”, [(“A”, 2), (“B”, 3), (“C”, 4)],
                         [(“dataX”, 8), (“dataY”, 16)],
                         [(“p”, -1, 1), (“elm”, 80)] ),
Interface(“comp”, “dst”, [(“A”, 2)],
                         [(“data”, 288)],
                         [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $D_{out} = D_{in}$
- $q_{out} = q_{in}$
- $L_{S_{out}} < L_{S_{in}}$
- $L_{Y_{out}} = 1$
- $X_{out} \neq X_{in}$
- $Y_{out} \neq Y_{in}$
- The instances that are being concatenated must be all at the end of the list in the input interface description, just as in the example above.

Implementation

This function implementation is very similar to the concatenate signals implementation as the signals get concatenated the same way. The difference here is that all the signals that are concatenated for each instance are again concatenated into an even larger signal. So in the example above there are 2 signals which would be 8+16 = 24 bits. The instances “B” and “C” are also concatenated, this means that there are a total of 3 (“B”) * 4 (“C”) * 24 = 288 bits being concatenated into one signal.

De-concatenate signals

This function de-concatenates 1 large signal into multiple smaller signals.

Example

```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 24), (“C”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dst”, [(“ch”, 3)], [ (“A”, 8), (“B”, 16), (“C”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $D_{out} = D_{in}$
- $q_{out} = q_{in}$
- $X_{out} = X_{in}$
- $L_{Y\_out} > L_{Y\_in}$

Implementation

The implementation of de-concatenating signals is pretty straightforward. The signal that is being de-concatenated is simply broken up into the multiple outgoing signals. So in the example above, “data” 23 down to 16 will be assigned to “A” and “data” 15 down to 0 will be assigned to “B”.

De-concatenate instances

This function de-concatenates 1 large instance into multiple smaller instances.

Example

```
Interface(“src”, “comp”, [(“ch”, 6), (“C”, 4)], [(“data”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)])
Interface(“comp”, “dst”, [(“A”, 2), (“B”, 3), (“C”, 4)], [(“data”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)]),
```

Conditions

- $n_{out} = n_{in} = 1$
- $D_{out} = D_{in}$
- $q_{out} = q_{in}$
- $Y_{out} = Y_{in}$
- $L_{X\_out} > L_{X\_in}$

Implementation

De-concatenating instances makes use of the way the instances are defined in the VHDL interface package. This makes it possible to simply connect the incoming interface to the outgoing interface by indexing correctly. In the example above, (“ch”, 6) is re-indexed to (“A”, 2) and (“B”, 3). This is going from a one dimensional array to a two dimensional array. So index [ch0] becomes index [A0, B0] all the way to [ch5] becomes index [A1, B2]

De-concatenate space

This function de-concatenates 1 large signal into multiple instances and signals.

Example

```
Interface(“src”, “comp”, [(“A”, 2)],
                         [(“data”, 288)],
                         [(“p”, -1, 1), (“elm”, 80)])
Interface(“comp”, “dst”, [(“A”, 2), (“B”, 3), (“C”, 4)],
                         [(“dataX”, 8), (“dataY”, 16)],
                         [(“p”, -1, 1), (“elm”, 80)] ),
```

Conditions

- $n_{out} = n_{in} = 1$
- $D_{out} = D_{in}$
- $q_{out} = q_{in}$
- $L_{S_{out}} > L_{S_{in}}$
- $L_{Y_{in}} = 1$
- $X_{out} \neq X_{in}$
- $Y_{out} \neq Y_{in}$
- The new instances at the output interface must all be at the end of the list, just as in the example above.

Implementation

This function implementation is very similar to the de-concatenate signals implementation as the signals get de-concatenated the same way. This function simply assigns part of the large incoming signal to the outgoing signal at the correct new instance. In the example above, “data” 23 down to 16 is connected to “dataX” of instance [B0, C0], similarly “data” 15 down to 0 is connected to “dataY” of instance [B0, C0]. This goes on until the last “data” 287 down to 280 is connected to “dataX” of instance [B2, C3].

Concatenate time

This function concatenates multiple time frames into 1 larger time frame.

Example

```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)],
                         [(“p”, -1, 1), (“packet”, 10), (“elm”, 8)]),
Interface(“comp”, “dst”, [(“ch”, 3)], [(“data”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $S_{out} = S_{in}$
- $r_{out} = r_{in}$
- $L_{D\_out} < L_{D\_in}$
- The dynamic properties that are being concatenated must result in a time frame with the correct size just as in the example above.

De-concatenate time

This function de-concatenates one time frame into multiple smaller time frames.

Example

```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dst”, [(“ch”, 3)], [(“data”, 8)],
                         [(“p”, -1, 1), (“packet”, 10), (“elm”, 8)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $S_{out} = S_{in}$
- $r_{out} = r_{in}$
- $L_{D\_out} < L_{D\_in}$
- The dynamic property that is being de-concatenated must result in time frames with the correct sizes just as in the example above.

Align instances

This function aligns the instances of an incoming interface such that the misalignment is removed.

Example

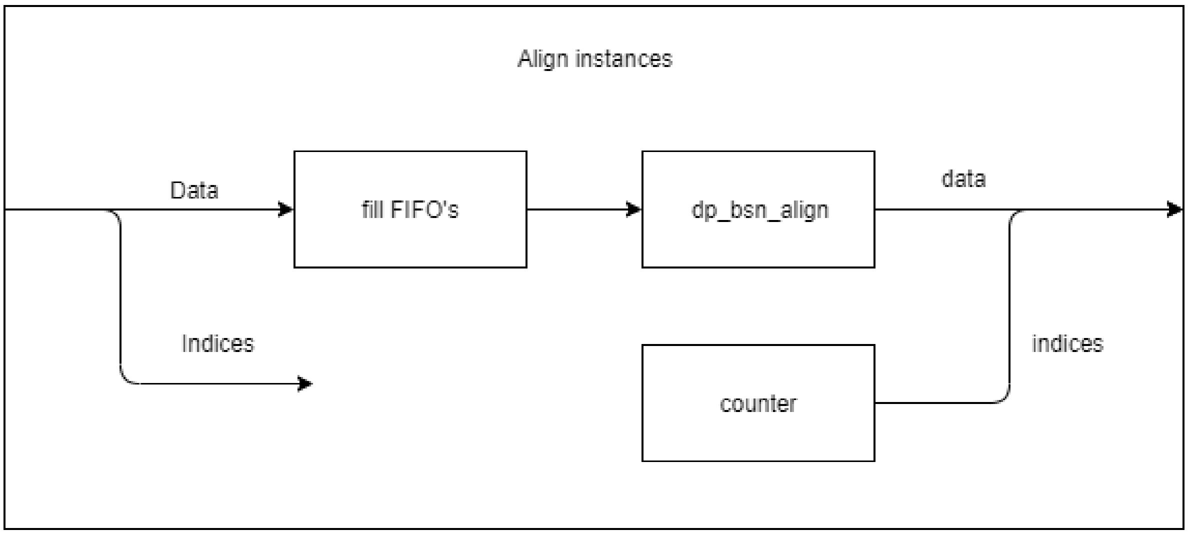
```
Interface(“src”, “comp”, [( “ch”, 3, (1.1, “us”) )], [(“data”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dest”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{out} = n_{in} = 1$
- $D_{out} = D_{in}$
- $Y_{out} = Y_{in}$
- $L_{S\_out} = L_{S\_in}$
- The only difference between the instances of the input and output interface must be that one or more instances have a misalignment defined in the input interface while those same instances have 0 misalignment in the output interface.

Implementation

Aligning instances is performed by the ASTRON IP dp_bsn_align. This template makes use of another ASTRON IP which is the fill FIFO. The fill FIFOs are needed to buffer the input in order to enable the dp_bsn_aligner to perform the actual alignment. The implementation is shown in the figure below. As you can see, the indices are not used as it makes use of the ASTRON control signals. The instances are re-generated at the output by a counter.



Split streams time

This function splits one stream into multiple streams in the time domain.

Example

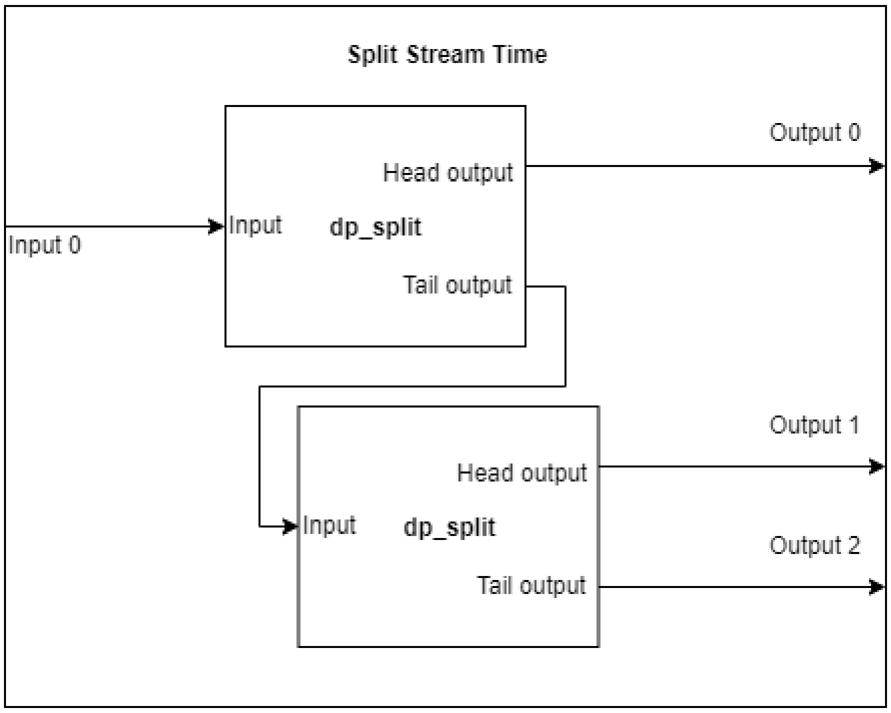
```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)]),
Interface((“comp”,0), “destA”, [(“ch”, 3)], [(“data”, 8)],
                               [(“p”, -1, 1), (“elm”, 10)]),
Interface((“comp”,1), “destB”, [(“ch”, 3)], [(“data”, 8)],
                               [(“p”, -1, 1), (“elm”, 70)])
```

Conditions

- $n_{in} = 1$
- $n_{out} > 1$
- $S_{out} = S_{in}$
- Only the last dynamic property in the list is being split, just as in the example.
- The sizes of the time frames at the output interfaces should add up to the size of the time frame in the input interface that is being split.

Implementation

The split stream time template makes use of the ASTRON dp_split IP. This IP splits one data stream into two according to the configured generics. These generics are first derived from the SamPy interfaces and then applied to the template. Cascading these IPs makes it possible to split one data stream into multiple. This is visualized in the figure below. In the figure it is shown how one input interface is split into three output interfaces. More split IPs are used if more output interfaces are defined. Note that the clock and reset signals are not drawn.



Split streams instances

This function splits one stream into multiple streams in the instances domain.

Example

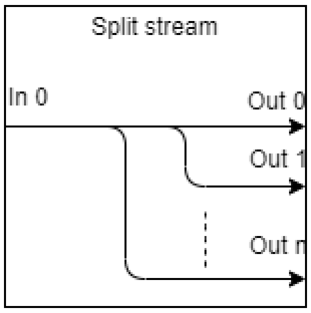
```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)]),
Interface((“comp”,0), “destA”, [(“ch”, 2)], [(“data”, 8)],
                               [(“p”, -1, 1), (“elm”, 80)]),
Interface((“comp”,1), “destB”, [(“ch”, 1)], [(“data”, 8)],
                               [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{in} = 1$
- $n_{out} > 1$
- $D_{out} = D_{in}$
- $Y_{out} = Y_{in}$
- Only one instance is being split, the rest of the instances should remain the same.
- the sizes of the instances at the output interfaces should add up to the size of the instance at the input interface that is being split.

Implementation

Splitting streams in the instances domain is very straightforward as the incoming interface can simply be partly connected to the outgoing interfaces by means of correct indexing. This is visualized in the figure below.



Split streams signals

This function splits one stream into multiple streams in the signals domain.

Example

```
Interface(“src”, “comp”, [(“ch”, 3)], [(“A”, 8), (“B”, 8), (“C”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)]),
Interface((“comp”,0), “destA”, [(“ch”, 3)], [(“A”, 8), (“C”, 8)],
                               [(“p”, -1, 1), (“elm”, 80)]),
Interface((“comp”,1), “destB”, [(“ch”, 3)], [(“B”, 8)],
                               [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{in} = 1$
- $n_{out} > 1$
- $D_{out} = D_{in}$
- $X_{out} = X_{in}$
- all signals of the output interfaces combined are identical to the signals in the input interface, just as shown in the example above.

Implementation

Splitting streams in the signals domain is similar to splitting streams in the instances domain as the incoming interface can simply be partly connected to the outgoing interfaces.

Merge streams time

This function merges multiple streams into 1 stream in the time domain.

Example

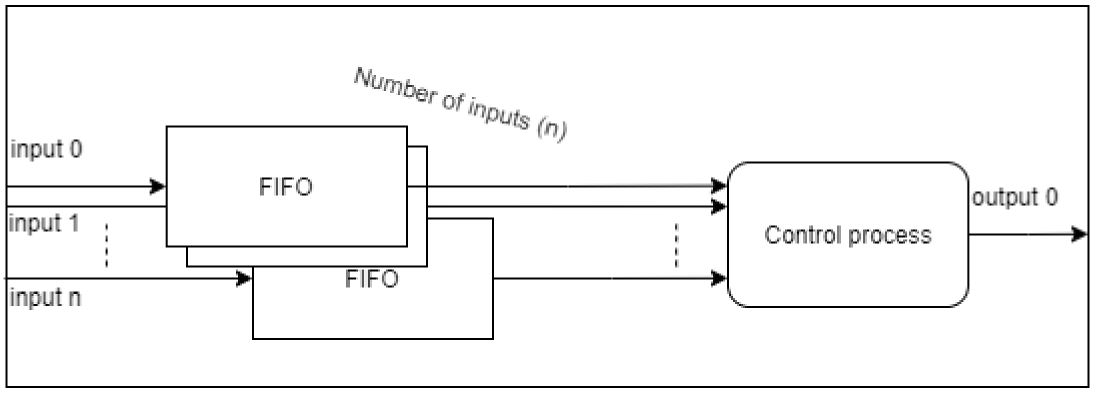
```
Interface(“srcA”, (“comp”,0), [(“ch”, 3)], [(“data”, 8)],
                              [(“p”, -1, 1), (“elm”, 10)]),
Interface(“srcB”, (“comp”,1), [(“ch”, 3)], [(“data”, 8)],
                              [(“p”, -1, 1), (“elm”, 70)]),
Interface(“comp”, “dst”, [(“ch”, 3)], [(“data”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{in} > 1$
- $n_{out} = 1$
- $S_{out} = S_{in}$
- Only the last dynamic property in the list is being merged, just as in the example.
- The sizes of the time frames that are being merged should add up to the size of the time frame in the output interface.

Implementation
Merging streams requires buffering of the incoming data as the multiple incoming streams may not be perfectly aligned due to latency of previous components in the system. This is done by means of FIFOs. These are available as an ASTRON IP and are used in this implementation. The architecture is shown in the figure below. As you can see, every input first gets buffered by a FIFO. The control process actively reads the data from the FIFOs according to definitions in the input interfaces. For example, if the SamPy description of input interface 0 describes that there are 8 elements in a time frame. The control process will read those 8 elements from FIFO 0 and transfers it to the output. After those 8 elements it outputs the number of elements defined in input interface 1 of FIFO 1, etc. A constraint with key "fifoSize" is used to overwrite the default minimum FIFO size of the FIFO on the corresponding input port. For example,

```
{"fifoSize" : {0 : 3,  # FIFO size of input port 0 is 3 element
               1 : 7 }}  # FIFO size of input port 1 is 7 elements
```



Merge streams instances

This function merges multiple streams into 1 stream in the instances domain.

Example

```
Interface(“srcA”, (“comp”,0), [(“ch”, 2)], [(“data”, 8)],
                              [(“p”, -1, 1), (“elm”, 80)])
Interface(“srcB”, (“comp”,1), [(“ch”, 1)], [(“data”, 8)],
                              [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, dst, [(“ch”, 3)], [(“data”, 8)],
                       [(“p”, -1, 1), (“elm”, 80)])
```

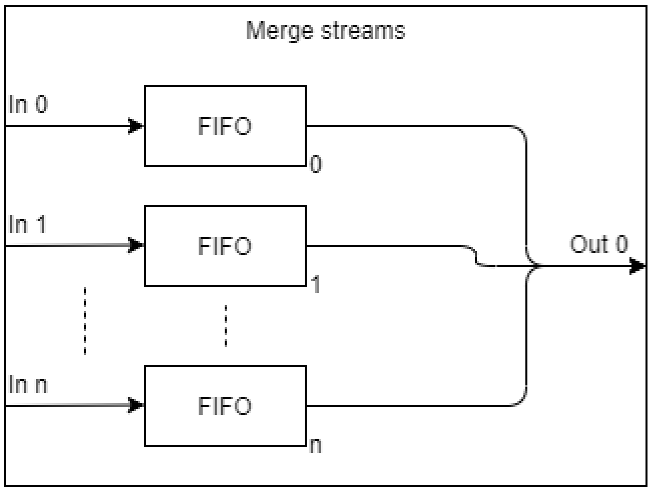
Conditions

- $n_{in} > 1$
- $n_{out} = 1$
- $D_{out} = D_{in}$
- $Y_{out} = Y_{in}$
- Only one instance is being merged, the rest of the instances should remain the same.
- the sizes of the instances that are being merged should add up to the size of the instance at the output interface.

Implementation

Merging streams in the instances domain is done by first buffering all the inputs in case of a misalignment. When all FIFOs contain data, the FIFOs will output their data simultaneously to form the merged output interface. This is shown in the figure below. A constraint with key "fifoSize" is used to overwrite the default minimum FIFO size of the FIFO on the corresponding input port. For example,

```
{"fifoSize" : {0 : 3,  # FIFO size of input port 0 is 3 element
               1 : 7 }}  # FIFO size of input port 1 is 7 elements
```



Merge streams signals

This function merges multiple streams into 1 stream in the signals domain.

Example

```
Interface(“srcA”, (“comp”,0), [(“ch”, 3)], [(“A”, 8), (“C”, 8)],
                              [(“p”, -1, 1), (“elm”, 80)]),
Interface(“srcB”, (“comp”,1), [(“ch”, 3)], [(“B”, 8)],
                              [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”,”dst”, [(“ch”, 3)], [(“A”, 8), (“B”, 8), (“C”, 8)],
                        [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{in} > 1$
- $n_{out} = 1$
- $D_{out} = D_{in}$
- $X_{out} = X_{in}$
- all signals of the input interfaces combined are identical to the signals in the output interface, just as shown in the example above.

Implementation

Merging streams in the signals domain is similar to merging streams in the instances domain. The inputs are first buffered by FIFOs in case of a misalignment. When all FIFOs contain data, the FIFOs will output their data simultaneously to form the merged output interface. A constraint with key "fifoSize" is used to overwrite the default minimum FIFO size of the FIFO on the corresponding input port. For example,

```
 {"fifoSize" : {0 : 3,  # FIFO size of input port 0 is 3 element
                1 : 7 }}  # FIFO size of input port 1 is 7 elements
```

Duplicate stream

This function duplicates the input interface to multiple output interfaces.

Example

```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)],
                         [(“p”, -1, 1), (“elm”, 80)]),
Interface((“comp”,0), “destA”, [(“ch”, 3)], [(“data”, 8)],
                               [(“p”, -1, 1), (“elm”, 80)]),
Interface((“comp”,1), “destB”, [(“ch”, 3)], [(“data”, 8)],
                               [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- $n_{in} = 1$
- $n_{out} > 1$
- $D_{out} = D_{in}$
- $S_{out} = S_{in}$

Implementation

The implementation of duplicating a stream is simply connecting the input interface directly to all output interfaces.

Sub-system

A sub-system component is a wrapper that instantiates a previously generated system.

Example

Note: comp is not a string as it is not between quotes. In this example, comp is a previously defined SamPy system.

```
Interface(“src”, comp, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)]),
Interface(comp, “dest”, [(“A”, 8)], [(“X”, 32)], [(“p”, -1, 1), (“pck”, 10)])
```

Conditions

- A component is said to be a SamPy subsystem if the sink and source of the input and output interfaces contain a SamPy system object.
- The defined input and output interfaces should be identical to the system input and output interfaces of the SamPy sub-system.

Implementation

The implementation is a simple wrapper that directly connects the inputs of this component to the inputs of the sub-system and the outputs of the sub-system to the outputs of this component. This is needed as the sub-system uses a different type definition for its interfaces.

Generator

This function generates data according to the definitions in the output interface.

Example

```
Interface("Gen", "dst", [("ch", 3)], [("data", 8)], [("p", -1, 1), ("elm", 80)])
```

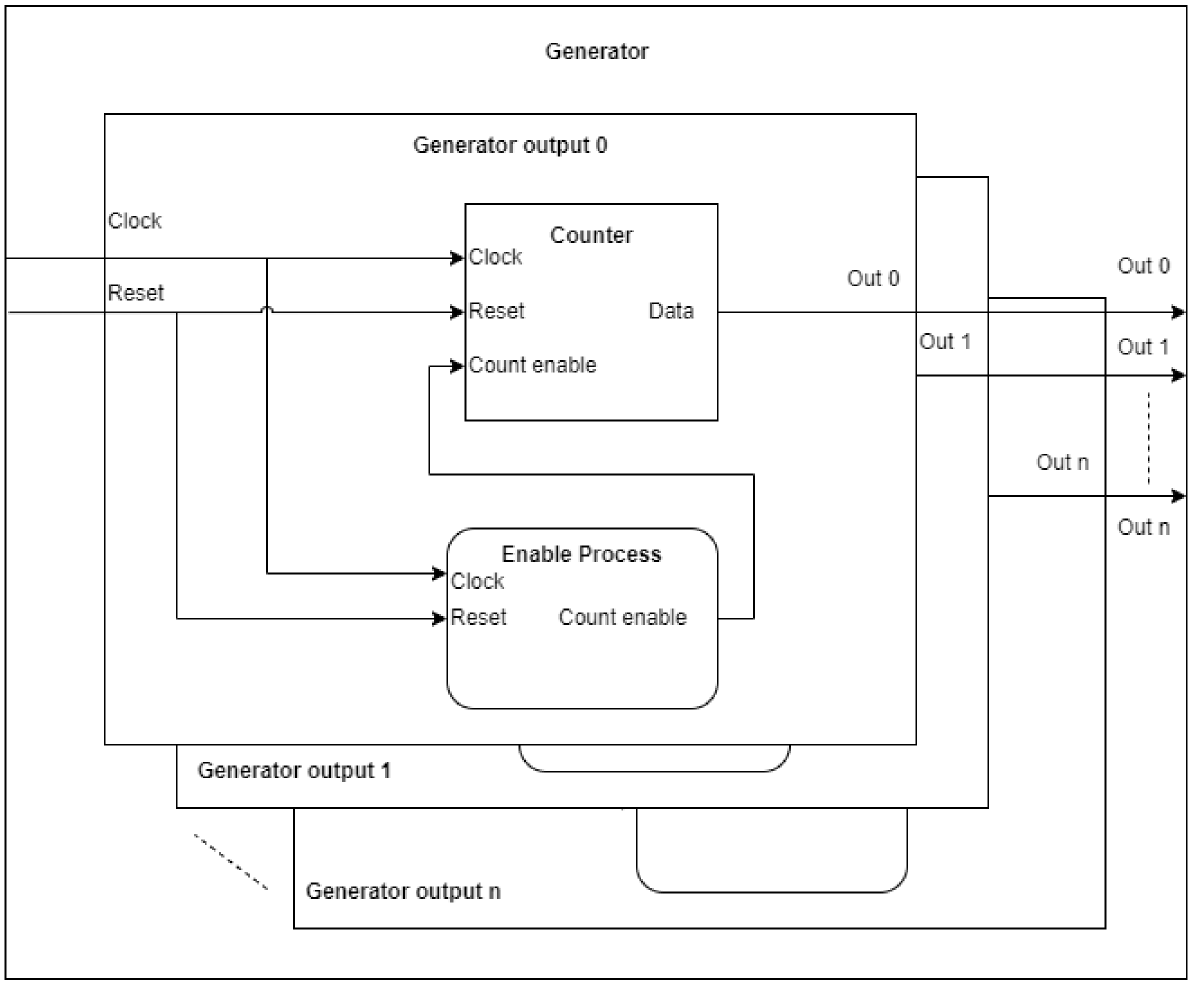
Conditions

- A component is said to be a generator if: it has no input interfaces, it is not a SamPy system and the name is not "input"

Implementation

The generator is made to generate a data stream that fits the SamPy interface description. This is done for every output of the generator that is defined. This is shown in the figure below. The counter is used to generate the indices that correspond with the dynamic properties, it also generates counter values for the signal fields. To ensure the timing of the data stream is correct, the enable process is used to enable or disable the counter at the right time. The dynamic dimension at which the ASTRON control signals are generated is determined by a constrained. The constrained has the key "controlDimension" the value is a Python dictionary where its keys are the port numbers of the generator and its values are the name of the desired dynamic property.

For example, {"controlDimension" : {0 : "Word", 1 : "Packet", 2 : "CB"}}
Where 0, 1, and 2 correspond with the output port number of the generator,
and "Word", "Packet", and "CB" correspond with the desired dynamic dimension of that output.



Sink

A sink is used to send data to which can be discarded.

Example

```
Interface(“src”, “Sink”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)])
```

Conditions

- A component is said to be a sink component if: it has no outputs, it is not a SamPy system and the name is not “output”

Implementation

A sink is simply an open connection.

Custom function

A custom component only has its interfaces defined, the implementation has to be done manually.

Example

```
Interface(“src”, “comp”, [(“ch”, 3)], [(“data”, 8)], [(“p”, -1, 1), (“elm”, 80)]),
Interface(“comp”, “dest”, [(“A”, 8)], [(“X”, 32)], [(“p”, -1, 1), (“pck”, 10)])
```

Conditions

- A custom function is assigned when no other function could be derived or if the user defined the “customFunction” constraint to be true.

Implementation

When a custom function is generated, a VHDL file is generated that contains the entity with the correct interface definitions and an empty architecture. A custom function will not be overwritten when re-generating a system. This keeps the users implementation intact.

Appendix C: SplitMergeTime VHDL description

```
LIBRARY IEEE, dp_lib;
USE IEEE.STD_LOGIC_1164.ALL;
USE IEEE.NUMERIC_STD.ALL;
USE dp_lib.dp_stream_pkg.ALL;
USE WORK.SamPy_records_pkg.ALL;


ENTITY SamPy_SplitMergeTime IS
  PORT (
        clk : IN STD_LOGIC;
        rst : IN STD_LOGIC;
        sink_in_0       : IN  SamPy_sosi_input_0;
        source_out_0 : OUT SamPy_sosi_Merge_0
  );
END SamPy_SplitMergeTime;


ARCHITECTURE str OF SamPy_SplitMergeTime IS
  SIGNAL input_sosi_0 : SamPy_sosi_input_0;
  SIGNAL Split_sosi_2 : SamPy_sosi_Split_2;
  SIGNAL Y_sosi_0 : SamPy_sosi_Y_0;
  SIGNAL Merge_sosi_0 : SamPy_sosi_Merge_0;
  SIGNAL Split_sosi_0 : SamPy_sosi_Split_0;
  SIGNAL X_sosi_0 : SamPy_sosi_X_0;
  SIGNAL Split_sosi_1 : SamPy_sosi_Split_1;
BEGIN

  input_sosi_0 <= sink_in_0;
  source_out_0 <= Merge_sosi_0;

  u_SamPy_Merge : ENTITY work.SamPy_Merge
  PORT MAP(
        clk => clk,
        rst => rst,
        sink_in_0       => X_sosi_0,
        sink_in_1       => Split_sosi_1,
        sink_in_2       => Y_sosi_0,
        source_out_0 => Merge_sosi_0
  );

  u_SamPy_Y : ENTITY work.SamPy_Y
  PORT MAP(
        clk => clk,
        rst => rst,
        sink_in_0       => Split_sosi_2,
        source_out_0 => Y_sosi_0
  );

  u_SamPy_X : ENTITY work.SamPy_X
  PORT MAP(
        clk => clk,
        rst => rst,
        sink_in_0       => Split_sosi_0,
        source_out_0 => X_sosi_0
  );

  u_SamPy_Split : ENTITY work.SamPy_Split
  PORT MAP(
        clk => clk,
        rst => rst,
        sink_in_0       => input_sosi_0,
        source_out_0 => Split_sosi_0,
        source_out_1 => Split_sosi_1,
        source_out_2 => Split_sosi_2
  );

END str;
```

Appendix D: ASTRON Science Case 3 Input Stage SamPy Script

Please note the constants defined at the top of the script which are used in the interface definitions.

```
# Constants
header = [
("eth_dst_mac"       , 48),
("eth_src_mac"       , 48),
("eth_type"          , 16),
("ip_version"        ,  4),
("ip_header_length"  ,  4),
("ip_services"       ,  8),
("ip_total_length"   , 16),
("ip_identification" , 16),
("ip_flags"          ,  3),
("ip_fragment_offset", 13),
("ip_time_to_live"   ,  8),
("ip_protocol"       ,  8),
("ip_header_checksum", 16),
("ip_src_addr"       , 32),
("ip_dst_addr"       , 32),
("udp_src_port"      , 16),
("udp_dst_port"      , 16),
("udp_total_length"  , 16),
("udp_checksum"      , 16),
("dp_reserved"       , 47),
("dp_sync"           ,  1),
("dp_bsn"            , 64)
]




t = (5, 'ns') # Clock period
p = ("Period", -1, 1.024) # Time frame period
m = (3.84, 'us') # Misalignment


# Streams
SC3Stream = Stream(
  Interface( 'input', 'dp_offload_rx',
               [ ('Stream', 24, m) ], [ ('Data', 64) ],
               [ p, ('Packet', 800000), ('Word', 187, t) ]
  ),
  Interface( ('dp_offload_rx',1), ('merge_custom',0),
               [ ('Stream', 24, m) ], [ ('Data', 64) ],
               [ p, ('Packet', 800000), ('Word', 180, t) ]
  ),
  Interface( 'merge_custom', 'reformat',
               [ ('Stream', 24, m) ], [ ('Data', 64) ],
               [ p, ('Packet', 800000), ('Word', 180, t) ]
  ),
  Interface( 'reformat', 'BSN_aligner',
               [ ('CB_set', 8), ('Stream', 3, m) ], [ ('Data', 64) ],
               [ p, ('Packet', 800000), ('Word', 180, t) ]
  ),
  Interface( 'BSN_aligner', 'dp_repack',
               [ ('CB_set', 8, m), ('Stream', 3) ], [ ('Data', 64) ],
               [ p, ('Packet', 800000), ('Word', 180, t) ]
  ),
  Interface( 'dp_repack', 'concat',
               [ ('CB_set', 8, m), ('Stream', 3) ], [ ('Data', 48) ],
               [ p, ('Packet', 800000), ('Word', 240, t) ]
  ),
  Interface( 'concat', 'deconcat',
               [ ('CB_set', 8, m) ], [ ('Data',144) ],
               [ p, ('Packet', 800000), ('Word', 240, t) ]
  ),
  Interface( 'deconcat', 'reframe',
               [ ('CB_set', 8, m), ('Dishes', 8) ], [ ('re', 9), ('im', 9) ],
               [ p, ('Packet', 800000), ('Word', 240, t) ]
  ),
  Interface( 'reframe', ('output',0),
              [ ('CB_set', 8, m), ('Dishes', 8) ], [ ('re', 9), ('im', 9) ],
              [ p, ('CB', 5), ('sb', 24), ('samples', 12500), ('channels',64), ('pols', 2, t) ]
  )
)



SC3HeaderStream = Stream(
  Interface( ('dp_offload_rx',0), 'repack_header',
              [ ('Stream', 24, m) ], [ ('Data', 64) ],
              [ p, ('Packet', 800000), ('Word', 7, t) ]
  ),
  Interface( 'repack_header', 'deconcat_header',
              [ ('Stream', 24, m) ], [ ('Data', 7*64) ],
              [ p, ('Packet', 800000), ('Word', 1, t) ]
  ),
  Interface( 'deconcat_header', 'copy_stream',
              [ ('Stream', 24, m) ], header,
              [ p, ('Packet', 800000), ('Word', 1, t) ]
  ),
  Interface( ('copy_stream',0), ('output',1),
              [ ('Stream', 24, m) ], header,
              [ p, ('Packet', 800000), ('Word', 1, t) ]
  )
)



SC3RestoredStream = Stream(
  Interface( ('copy_stream',1), 'truncate_stream',
              [ ('Stream', 24, m) ], header,
              [p, ('Packet', 800000), ('Word', 1, t)]
  ),
  Interface( 'truncate_stream', ('merge_custom',1),
              [ ('Stream', 24, m) ], [ ('dp_sync', 1), ('dp_bsn', 64) ],
              [ p, ('Packet', 800000), ('Word', 1, t) ]
  )
)

# System
SC3Sys = System("unb2b_SC3_input_subSystem", SC3Stream, SC3HeaderStream, SC3RestoredStream)

genRadioHDLProject(SC3Sys)
```