FRAMEWORK USER GUIDE

Carlos Gabriel de Araujo Gewehr

carlos.gewehr@ecomp.ufsm.br

1. Introduction and Overview:

This document describes the functionalities of the hybrid topology interconnection generation framework. After a brief introduction exposing the overall principles of its operation, a step-by-step execution of a standard use case is performed, where functionalities are covered in-depth as needed in the use case being explored.

The framework is implemented in VHDL for its hardware components and Python 3 for its software components. The following Python libraries are required, all of them included in most popular distributions:

- argparse,
- copy,
- json,
- math,
- os,
- random,
- sys

The software component consists of two modules: *AppComposer* and *PlatformComposer*.

AppComposer can be used to emulate an existing application observed communication pattern, as CBR (Constant Bit Rate) bandwidths between its threads. In this manner, the emulated application, or group of applications, communication pattern can serve as stimulus to any arbitrary network topology describable through the framework, without having the need to factually execute it to produce accurate "real-world" traffic to measure a certain topology or test-case's capabilities.

PlatformComposer can be used to describe a hybrid topology interconnection network, composed of a base NoC, with Buses or Crossbar connected to any router's local port. Beyond topology related parameters, other parameters can be manipulated as well, such as routers buffer sizes and Bus arbiters.

The hardware component consists of synthesizable VHDL, for implementing network topologies, and non-synthesizable VHDL, for generating stimulus. For both cases, the VHDL implementations are deeply parametric, as to make possible simulating arbitrary topologies with arbitrary emulated real-world applications as stimulus.

The interfacing of the hardware component with the software component happens through several JSON files, each containing the parameters expected by its associated VHDL implementation (Topology parameters for describing the interconnection network, and thread communication patterns for application emulation).

A complete execution of the common use case is done in two steps: Description and Execution.

In the Description step, 4 definitions must be elaborated: Topology, Workload, Allocation Map and Cluster Clocks (for non-DVFS executions). Topology definitions are done through the

use of *PlatformComposer*, Workloads with *AppComposer*, Allocation Maps and Cluster Clocks with plain Python.

In the Execution step, the definitions made in the Description step are used to generate the JSON files containing the parameters expected by the hardware. As per the defined parameters, the desired topology is simulated in RTL, with the emulated application as stimulus.

The example use case to be explored is exposed in the two figures below, with 3 PIP applications mapped in the following topology:

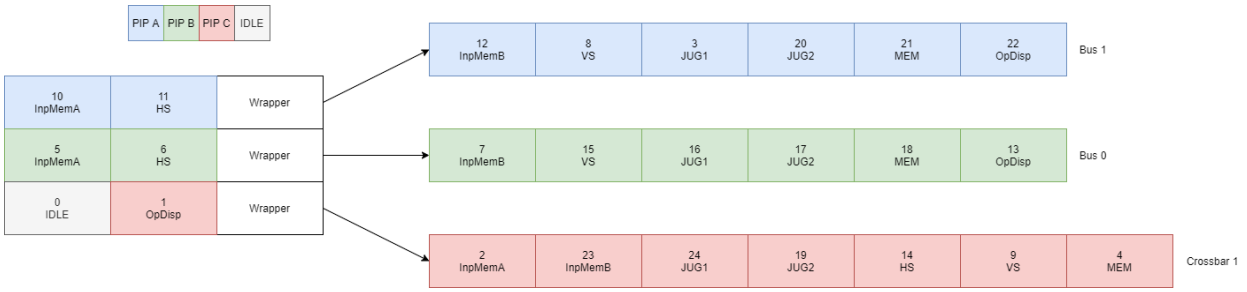

Figure 1 - Example network topology with 3 PIP applications mapped.

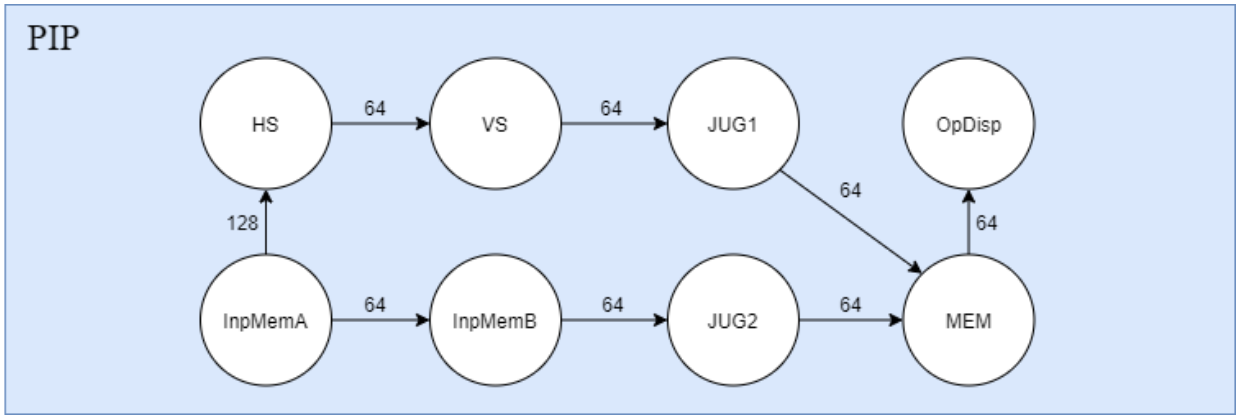

Figure 2 - PIP communication graph, with vertexes as threads, and edges as communication bandwidth, in MBps (For example, InpMemA sends 128 Megabytes/second to HS).

A "-h" argument for the commands and scripts mentioned throughout this guide is always available. Executing a command/script with this argument describes the required and optional arguments for the command/script it is executed with, which might be useful if this guide is ever unclear or inadequate for a specific functionality.

2. Setup

The first step is to execute the *setup.py* script, located at "setup/setup.py". This script generates two other scripts, which, when executed, define necessary environment variables. One of these scripts, meant to be executed in Linux, is written to a ".source" file, while the other, meant for Windows, is written to a ".bat" file. (MacOS or any other OS is incompatible). These generated scripts must be executed (with the "source" command, in Linux, or "call", in Windows) before any attempt to use the framework in a new *shell* instance. It is recommended that, for Linux, the "source" call to the generated script is added to ".bash_aliases", removing the need to manually execute it at every new shell instance.

The *setup.py* script requires the following arguments:

- *InstallName:* Main command name to be executed from shell e.g. (*<InstallName> compile [project]*). If not given, "hibrida" will be taken as default.
- *InstallPath:* Framework's main directory (Where "doc/", "data/", "scripts/", "setup/" and "src/" are contained). If not given, *setup.py*'s parent directory will be taken as default.
- *DefaultProjDir*: Default directory for new projects. If not given, "Desktop/HibridaProjects" will be taken as default.

For a main command name "hibrida" and an install directory "/home/usr/ExUser/hibrida/", *setup.py* should be executed as:

python setup.py -InstallName hibrida -InstallPath /home/usr/ExUser/hibrida/

Executing the above command, "hibrida.source" and "hibrida.bat" will be created in the same directory as *setup.py*.

Additionally, the script creates a *config.json* file, located at "data/config.json". This file contains default directories info, as well as project info, to be filled out later, when using the *projgen* and *setConfig* commands.


```
{
    "HibridaConfigFile": "/home/usr/cgewehr/Desktop/FrameworkHibrida/data/config.json",
    "AllocationMapsPaths": [
        "/home/usr/cgewehr/Desktop/FrameworkHibrida/data/flowgen/allocationMaps"
    ],
    "Projects": {
    },
    "WorkloadsPaths": [
        "/home/usr/cgewehr/Desktop/FrameworkHibrida/data/flowgen/workloads"
    ],
    "DefaultProjDir": "/home/usr/cgewehr/Desktop/HibridaProjects",
    "HibridaName": "hibrida",
    "HibridaPath": "/home/usr/cgewehr/Desktop/FrameworkHibrida",
    "ApplicationsPaths": [
        "/home/usr/cgewehr/Desktop/FrameworkHibrida/data/flowgen/applications"
    ],
    "TopologiesPaths": [
        "/home/usr/cgewehr/Desktop/FrameworkHibrida/data/flowgen/topologies"
    ],
    "ClusterClocksPaths": [
        "/home/usr/cgewehr/Desktop/FrameworkHibrida/data/flowgen/clusterClocks"
    ],
    "MostRecentProject": null
}
```


Figure 3 - config.json file, where information is stored as framework command are executed

3. Description Step

a. Topology Description

Topology descriptions are done through the *Platform* class in the *PlatformComposer* module and the *Bus* and *Crossbar* classes in the *Structures* module. Starting with a Platform object, the *Platform.addStructure()* method can be called to insert a new Bus/Crossbar at a given position in the base NoC.

For the topology from the use case example being explored, the required *PlatformComposer* manipulations necessary are:

```
import PlatformComposer

# Creates base 3x3 NoC
Setup = PlatformComposer.Platform(BaseNoCDimensions=(3, 3))

# Adds crossbar containing 7 PEs @ base NoC position (2, 0)
CrossbarA = PlatformComposer.Crossbar(AmountOfPEs = 7)
Setup.addStructure(NewStructure=CrossbarA, WrapperLocationInBaseNoC=(2, 0))

# Adds bus containing 6 PEs @ base NoC position (2, 1)
BusA = PlatformComposer.Bus(AmountOfPEs = 6)
Setup.addStructure(NewStructure=BusA, WrapperLocationInBaseNoC=(2, 1))

# Adds bus containing 6 PEs @ base NoC position (2, 2)
BusB = PlatformComposer.Bus(AmountOfPEs = 6)
Setup.addStructure(NewStructure=BusB, WrapperLocationInBaseNoC=(2, 2))

Setup.toJSON(SaveToFile = True, FileName = "ExampleTopology")
```

Figure 4 - Using PlatformComposer to describe the topology from Figure 1

Executing the script above creates a “ExampleTopology.json” file in its directory, containing the topology parameters which describe the desired network topology for the example being explored, such as Bus/Crossbar positions in base NoC, their sizes, and PEPos values for its associated PEs (file is too big to be included as a figure in this document). This file will be used later on in the Execution step.

b. Application Description

In a similar fashion, application descriptions are done through the *AppComposer* module. The Application, Thread and Flow classes are hierarchically used, with Application at the top. A Flow class represents an edge in an application 's communication graph (associating two vertices with a quantified value, in this case, communication bandwidth, in MBps), and a Thread its vertices. Thread classes are a collection of Flow objects, and Application classes a collection of Thread objects.

For the application (PIP) in the example being explored:

```
import AppComposer

# Make Application
PIP = AppComposer.Application(AppName = "PIP")

# Make Threads
InpMemA = AppComposer.Thread(ThreadName = "InpMemA")
HS = AppComposer.Thread(ThreadName = "HS")
VS = AppComposer.Thread(ThreadName = "VS")
JUG1 = AppComposer.Thread(ThreadName = "JUG1")
InpMemB = AppComposer.Thread(ThreadName = "InpMemB")
JUG2 = AppComposer.Thread(ThreadName = "JUG2")
MEM = AppComposer.Thread(ThreadName = "MEM")
OpDisp = AppComposer.Thread(ThreadName = "OpDisp")

# Add Threads to applications
PIP.addThread(InpMemA)
PIP.addThread(HS)
PIP.addThread(VS)
PIP.addThread(JUG1)
PIP.addThread(InpMemB)
PIP.addThread(JUG2)
PIP.addThread(MEM)
PIP.addThread(OpDisp)

# Add Flows to Threads (Bandwidth parameter must be in Megabytes/second)
InpMemA.addFlow(AppComposer.Flow(TargetThread = HS, Bandwidth = 128))       # InpMemA -- 128 -> HS
InpMemA.addFlow(AppComposer.Flow(TargetThread = InpMemB, Bandwidth = 64))   # InpMemA -- 64 -> InpMemB
HS.addFlow(AppComposer.Flow(TargetThread = VS, Bandwidth = 64))             # HS -- 64 -> VS
VS.addFlow(AppComposer.Flow(TargetThread = JUG1, Bandwidth = 64))           # VS -- 64 -> JUG1
JUG1.addFlow(AppComposer.Flow(TargetThread = MEM, Bandwidth = 64))          # JUG1 -- 64 -> MEM
InpMemB.addFlow(AppComposer.Flow(TargetThread = JUG2, Bandwidth = 64))      # InpMemB -- 64 -> JUG2
JUG2.addFlow(AppComposer.Flow(TargetThread = MEM, Bandwidth = 64))          # JUG2 -- 64 -> MEM
MEM.addFlow(AppComposer.Flow(TargetThread = OpDisp, Bandwidth = 64))        # MEM -- 64 -> OpDisp

# Save App to JSON
PIP.toJSON(SaveToFile = True, FileName = "PIP")

```

Figure 5 - Using AppComposer to describe the application from Figure 2

Hierarchically, Application, Thread and Flow objects are defined, and linked to their parent object with the *Application.addThread()* and *Thread.addFlow()* methods.

Like previously done for the topology description, the Application class is exported in JSON format to “PIP.json”. This file will be used for defining a Workload.

c. Workload Description

In the same way that Application objects are a collection of Thread objects, Workload objects are a collection of Application objects. A Workload can be composed of any number of instances of any Application object, as made such as in Figure 4. This can be leveraged as to easily reuse previously made Application objects in different Workloads.

For the example use case being explored, 3 PIP applications are instantiated. Such a workload can be described, using *AppComposer*, like:

```
1  import os
2  import AppComposer
3
4  # Makes Workload object
5  PIP_WL = AppComposer.Workload(WorkloadName = "PIP_WL")
6
7  # Opens PIP App json file
8  with open(os.getenv(HIBRIDA_PATH) + "/data/flowgen/applications/PIP.json") as PIP_JSON:
9
10     # Builds 3 PIP Apps from JSON and adds them to PIP_WL Workload
11     for i in range(3):
12
13         PIPApp = AppComposer.Application()
14         PIPApp.fromJSON(PIP_JSON.read())
15         PIPApp.AppName = "PIP_" + str(i)
16         PIP_WL.addApplication(PIPApp)
17         PIP_JSON.seek(0)
18
19 # Exports Workload to json format
20 PIP_WL.toJSON(SaveToFile = True, FileName = "PIP_WL")
```

Figure 6 - Using "PIP.json" to describe a Workload

Executing the script in Figure 5 creates "PIP_WL.json", describing the 3 PIP instances. This file will be used in the Execution step.

d. Allocation Map Description

Allocation Maps are more simply described than Workloads or Topologies. It is a list of Application and Thread name strings, associating a Thread to a location in the network (PEPos).

Threads should be identified as "<AppName>.<ThreadName>", or a list of such, if multiple Threads are to be allocated to the same PEPos.

For the example being explored:

```
import json

# AllocMap[PEPos] = $App.$Thread
AllocArray = [None] * 25

AllocArray[0] = None
AllocArray[1] = "PIP_3.OpDisp"
AllocArray[2] = "PIP_3.InpMemA"
AllocArray[3] = "PIP_1.JUG1"
AllocArray[4] = "PIP_3.MEM"
AllocArray[5] = "PIP_2.InpMemA"
AllocArray[6] = "PIP_2.HS"
AllocArray[7] = "PIP_2.InpMemB"
AllocArray[8] = "PIP_1.VS"
AllocArray[9] = "PIP_3.VS"
AllocArray[10] = "PIP_1.InpMemA"
AllocArray[11] = "PIP_1.HS"
AllocArray[12] = "PIP_1.InpMemB"
AllocArray[13] = "PIP_2.OpDisp"
AllocArray[14] = "PIP_3.HS"
AllocArray[15] = "PIP_2.InpMemB"
AllocArray[16] = "PIP_2.JUG1"
AllocArray[17] = "PIP_2.JUG2"
AllocArray[18] = "PIP_2.MEM"
AllocArray[19] = "PIP_3.JUG2"
AllocArray[20] = "PIP_1.JUG2"
AllocArray[21] = "PIP_1.MEM"
AllocArray[22] = "PIP_1.OpDisp"
AllocArray[23] = "PIP_3.InpMemB"
AllocArray[24] = "PIP_3.JUG1"

AllocJSONString = json.dumps(AllocArray, sort_keys = False, indent = 4)

with open("ExampleAllocMap.json", "w") as JSONFile:
    JSONFile.write(AllocJSONString)
```

Figure 7 - Making an Allocation Map file

This script creates "ExampleAllocMap.json", which contains a mapping of Threads to PEPos values. This file will be used in the Execution step.

e. Cluster Clocks Description

A Cluster Clocks file establishes clock periods (in nanoseconds) for every cluster in a described topology. A cluster is defined as a single router in the base NoC plus its associated Bus/Crossbar, if any. As with Allocation Maps, Cluster Clocks are described using plain Python.

For the example being explored, defining 100 MHz clocks (10 ns period) for every cluster:

```
1   import json
2
3   # ClusterClocks[BaseNoCPos] = Clock Period (in ns)
4   ClusterClocks = [None] * 9
5
6   ClusterClocks[0] = 10
7   ClusterClocks[1] = 10
8   ClusterClocks[2] = 10
9   ClusterClocks[3] = 10
10  ClusterClocks[4] = 10
11  ClusterClocks[5] = 10
12  ClusterClocks[6] = 10
13  ClusterClocks[7] = 10
14  ClusterClocks[8] = 10
15
16  ClocksJSONString = json.dumps(ClusterClocks, sort_keys = False, indent = 4)
17
18  with open("ExampleClusterClocks.json", "w") as JSONFile:
19      JSONFile.write(ClocksJSONString)
```

Figure 8 - Making a Cluster Clocks file

4. Execution Step

In the Execution step, the 4 required descriptions made in the Description step are used to create JSON configuration files to be read by the VHDL implementations, and simulate the intended network topology, with stimulus emulating the desired real-world application.

Each sub-step in the Execution step is implemented as a sub-command to the main command defined by the setup script, in Chapter 2. For the example being explored, it was previously defined as "hibrida".

a. projgen

In this command, the required directory structure for a framework project is established. 4 main directories are created: "platform/", where topology and cluster clock parameters will be stored; "flow/", for application emulation parameters; "log/", for run-time logs; "src_json/" for the original JSON files, from the Description step.

projgen expects two arguments:

- *ProjectDirectory, pd*: Directory where "flow/", "log/", "flow/" and "src_json" will be contained. If not given, the default project dir as defined in *config.json* will be taken as default;
- *ProjectName, pn*: Name of new project, to be indexed in projects list in *config.json*. If not given, "HibridaProject" will be taken as default.

In the example being explored, from the main command name defined as "hibrida", to create a new project "ExampleProject" at "C:/HibridaProjects/ExampleProjects", *projgen* should be executed as:

hibrida projgen -pd C:/HibridaProjects/ExampleProject -pn ExampleProject

Which creates the directory structure:

```
PS C:\HibridaProjects\ExampleProject> ls


    Directory: C:\HibridaProjects\ExampleProject


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----        21-Aug-20  02:25 AM                flow
d-----        21-Aug-20  02:25 AM                log
d-----        21-Aug-20  02:25 AM                platform
d-----        21-Aug-20  02:25 AM                src_json
```

Figure 9- Directories created by projgen

projgen also creates a *makefile*, which can be used to interface with the EDA tool used to compile/elaborate/simulate the VHDL implementations (Cadence tools are assumed as default). It can be used directly, through the *make* Linux command (*make compile, make elab, ...*), or through the framework's common frontend, as *hibrida compile*, *hibrida elab, ...*

b. setConfig

The files created in the Description step are linked to a project by the use of *setConfig*. Files can be linked individually, with multiple calls to *setConfig*, or all at once, with a single call to *setConfig*. Each required file has an associated argument in *setConfig*:

- *ProjectDir, p*: Project created by *projgen*;
- *TopologyFile, t*: Topology file, created in the Description step;
- *WorkloadFile, w*: Workload file, created in the Description step;
- *AllocationMapFile, a*: Allocation Map file, created in the Description step;
- *ClusterClocksFile, c*: Cluster Clocks file, created in the Description step;
- *State, s*: Print out status of required files

Files can be given either as an absolute path or as a relative path, in which case the default directory for its file type, as defined in config.json, is concatenated to the given relative path. Additional directories for files to be searched for if given as relative paths can be added through the *addSearchPath* command (not covered in this guide).

Setting the files created in this guide at the Description step, using relative paths and multiple calls to *setConfig*, should be executed as:

hibrida setConfig -p ExampleProject -t ExampleTopology.json
hibrida setConfig -p ExampleProject -w PIP_WL.json
hibrida setConfig -p ExampleProject -a ExampleAllocMap.json
hibrida setConfig -p ExampleProject -c ExampleClusterClocks.json

Checking the status of the project's files can be done by executing *setConfig* with the "-s" argument:

hibrida setConfig -p ExampleProject -s

```
[cgewehr@gmicroll ExampleProject]$ hibrida setConfig -p ExampleProject -s

Allocation Map file: /home/usr/cgewehr/Desktop/FrameworkHibrida/data/flowgen/allocationMaps/ExampleAllocMap.json
ClusterClocks file: /home/usr/cgewehr/Desktop/FrameworkHibrida/data/flowgen/clusterClocks/ExampleClusterClocks.json
Topology file: /home/usr/cgewehr/Desktop/FrameworkHibrida/data/flowgen/topologies/ExampleTopology.json
Workload file: /home/usr/cgewehr/Desktop/FrameworkHibrida/data/flowgen/workloads/PIP_WL.json
```

.

c. flowgen

With *flowgen*, the Description files previously defined with *setConfig* will be used to generate the parameter files expected by the hardware component, implementing the intended network topology and stimulus.

For the example being explored, *flowgen* should be executed as:

hibrida flowgen -p ExampleProject

After its execution, “flow/” will be populated by the JSON config files to be read by the Injectors and Triggers and “platform/” by the JSON files to be read by the top level entity, which instantiates the base NoC and Bus/Crossbars.

d. run

The *run* command is used to interface with the makefile generated with *projgen* through the framework’s common frontend, invoking the *all* makefile recipe. With it, VHDL files are compiled, the top level entity elaborated (with the configuration files generated by *flowgen*) and the HDL simulator opened with a waveform viewer.

For the example being explored:

hibrida run -p ExampleProject