GALAXY Tools Manual

1. Introduction

2. Administrator Guide – How to install the tools

2.1 Tools Installation and Setup – GUI part

The tools are provided as a source code package, which can be compiled on any UNIX system.

2.1.1 Dependencies

The dependencies are:

- GTKmm 2.4 (usual package name: gtkmm24-devel or libgtkmm-2.4-dev)
- Xerces-c (usual package name: xerces-c-devel or libxerces-c2-dev)
- Expat (usual package name: expat-devel or libexpat1-dev)
- Boost (usual package name: boost-devel or libboost-dev)
- Graphviz (usual package name: graphviz-devel or libgraphviz-dev)
- Libgvc (sometimes included in Graphviz, or in package libgvc)
- XSD from CodeSynthesis (usual package name: xsd)
- Gawk (usually included with OS, may require installation on MacOS)
- If co-simulation is wanted: Portico (see below)

On recent linux/unix systems, most dependencies can be installed by using installation tools such as yum or apt.

```
yum install gtkmm24-devel xerces-c-devel expat-devel
graphviz-devel boost-devel xsd

OR

yum install libgtkmm-2.4-dev libxerces-c2-dev
libexpat1-dev libgraphviz-dev libboost-dev libgvc
xsd
```

Two dependencies, XSD and Portico, may need to be installed manually.
XSD's download page is: http://codesynthesis.com/products/xsd/download.xhtml
Portico's download page is: http://porticoproject.org/index.php?title=Download

2.1.2 Compilation and Installation

The *galaxy-tools-<version>.tar.bz2* file can be decompressed anywhere. It creates a *galaxy-tools-<version>* directory, from which one can run the usual configure, make, make install, possibly as root for the last step.
Two arguments may be passed to the *configure* script:

--with-xsd=<path-to-xsd> : indicates where XSD was installed
--prefix=<path> : indicates where to install the tools after compilation (/usr/local/galaxy recommended)

The final step is to adjust the PATH and GALAXY_PATH environment variables:

```
tar xjf galaxy-tools-1.0.tar.gz

cd galaxy-tools-1.0

./configure ––prefix=/usr/local/galaxy

make

sudo make install

export GALAXY_PATH=/usr/local/galaxy

export PATH=${PATH}:${GALAXY_PATH}/bin
```

Make sure every user has GALAXY_PATH and PATH set up.

2.1.3 Testing

Correct operation can be checked by running *galaxy-ide* on an example:

```
cd test/g3card

galaxy-ide –i g3card.asip.xml
```

2.2 Back-end configuration (Compilation&simulation scripts)

The compilation and simulation scripts can be found in the source code sub-directory: *galaxy-tools-<version>/scripts*.
Three classes of scripts are there:

- ***galaxy-wrapper-****: wrapper scripts around compilers or tools able to convert files from one format to another
- ***galaxy-launcher-****: scripts launching a simulator and generating a trace file
- ***galaxy-loader-****: scripts transfering a file on an fpga board

For each script, 2 versions are available: *<script name>-local and <script name>-remote*.
During installation (*make install*), each *<script>-local* is run with the *–t* option, which tests whether the corresponding tool is installed locally (for example, "*galaxy-wrapper-xst-local –t"* tests if *xst* is in the current PATH. If yes, then *<script>-local* will be installed. Otherwise, *<script>-remote* will be installed. In both cases, the *-local* or *-remote* extension is removed in the installed version of the script.

Using the remote versions of the scripts is really not recommended at the moment. Too many things are specific to our own system, and it needs a big update.

The local scripts may all work directly.
The easiest way to test them is to describe a project using the desired simulators, as described in Section 3.3 "Starting a simulation", and check any error message.

3. User Guide – How to use the tools

3.1 XPipes NoC

3.1.1 Lib_xpipes and its NoC building blocks

Step 1: Adding the component library

Using the '+' icon button located at the bottom of the Component Library View, add the library *lib_xpipes.asip.xml*.
The library should be present in the following directory:
<galaxy_installation_directory>/share/asip_libraries/lib_xpipes.
By default the Galaxy installation directory is */usr/local/galaxy*. If it is not present, ask your administrator.

Step 2: Instantiating building blocks

Three building blocks are available:

- Switches
- Network interface Initiators
- Network interface Targets

To instantiate one of these components: Drag&drop it from the Component Library View to the main design view (called Graph View).
These are parameterised components. A window should appear to make you customise those parameters.
The environment is automatically switched to "Edit mode", where you can move and resize components with the mouse in the design view.
Note 1: Once a parameterised component has been instantiated with a set of parameters, it is also shown as a new component in the Components Library View (with a name of the form "*main_name_param1_param2_param3…*"). You can create more instances of the same component without having to re-enter the parameters, by drag&dropping this new item.
Note 2: If you just intend to generate a *noc* file to use with XPipes, you only need these 3 building blocks. There is no need to connect real OCP components (processors, memories, etc.) to the network interfaces, as these will be inferred automatically by the XPipes tools.

Step 3: Creation of NoC interconnect

Select 2 components (click the 1st one, then CTRL+click the 2nd one) then select "Create Connection" from the Design menu.
A new window appears, where you can drag&drop ports (or groups thereof) from one side to the other side to create connections.
For example, select a NI_Initiator and a Switch, and connect the "noc input port" of the 1st component to the "noc output port 1" of the 2nd component.
Note: In the design view, the automatic layout of the new connections is not implemented yet. It is normal that they look rather "unorganised".

Step 4: Creation of NoC routes

To create each route: Select a series of components

- Starting by clicking on an initiator network interface, then
- CTRL+clicking successive switches, and finally
- CTRL+clicking the final target network interface)

Finalise the route creation by selecting “Create NoC connection group” from the Design menu.
Repeat for each route.
Created routes can be inspected or deleted by selecting a source component (usually an initiator network interface) and selecting “Edit connection groups” from the Design menu.

Step 5: Clock domains
Clock domains are created by selecting “Edit clock domains” from the Design menu.
In order to work with the XPipes tools, the names of the clock domains must follow the XPipes conventions: “clk_x_y”, where x and y are integers starting from 1, and where all mesochronous clock domains should share the same x numbers.
Enter the clock names, and indicate which clock domains are mesochronous to each other.
Note: The clock domains will be created as invisible ports for the top-level component.

Step 6: Assigning clock domains to the NoC building blocks
For each component (network interfaces and switches):
Select the component, and click “Connect global signals” from the Design menu.
Connect the noc_Clock of the component to the desired clock domain by using the drag&drop technique.
Note: You can leave some component clocks unconnected. They will automatically be assigned to the first clock domain (clk_1_1) by the next tools.
Note 2: The clock connections will be created as invisible connections to the (invisible) top-level clock ports. If you wish to make all those visible, you can do so by right-clicking on the design view, and selecting “Global connections On”.

Step 7: Generating the XPipes .noc file
- Make sure you save your project first.
- Select “Generate XPipes” from the Design menu.
- The Execution window should come up with a lot of red messages. Do not worry, these are only warnings.
- Check your current directory. A *.noc* file should have been generated.
- Use your XPipes tools to process this *noc* file.

3.2 Parameterised components (via Generators)

Parameterised components have to be described by hand for the moment, as there is no GUI wizard to create them.
A good starting point is to look at the XPipes library ***libxpipes.asip.xml*** located in *<galaxy_install_directory>/share/asip_libraries/lib_xpipes*.
After a few headers, the first ASIP component description looks like:

```
<ASIP-component generator="xpipes_asip_generator_initiator"
name="NI_Initiator">
    <version>
      <version-number>1.0</version-number>
      <description></description>
    </version>
    <parameters>
      <parameter default-value="6" name="NB_BUFFERS"/>
    </parameters>
  </ASIP-component>
```

The items that are specific to parameterised components are:

- The ***parameters*** section, lines 6-8
- The ***generator*** attribute, line 1

3.2.1 *Parameters* section

The <*parameters*> section contains <*parameter*> (without '*s*') elements with these attributes:

- *Name* (required): the name of the parameter
- *default-value* (optional(?)): a default value

We may add a *type* attribute in the future, but it is not in there at the moment, so no type checking is done (i.e. the user can enter a string when a number is expected. The generator is in charge of checking the validity).

3.2.2 *Generator* attribute

A *Generator* is an executable program or script, which will generate an ASIP-component description based on the provided parameters.
In our XPipes example, the generator script's name is *xpipes_asip_generator_initiator*. You should be able to find it installed in *<galaxy_install_directory>/bin*.

If you look at this generator script, the important lines are:

```
while getopts "o:p:vth" OPTION
do
     case $OPTION in
         o)
             OUTPUT_FILE=$OPTARG
             ;;
         p)
             PARAMS="$PARAMS -e s=$OPTARG=g"
             ;;
[…]
sed ${PARAMS}
${SCRIPT_DIR}/xpipes_asip_generator_initiator.template >
${OUTPUT_FILE}
```

The *getopts+case* part is parsing the command line arguments, which are:

-o <output file name>
-p=<parameter name>=<user value>

The *sed* command (over the last 3 lines) is reading the template file *xpipes_asip_generator_initiator.template* from the same directory, and is doing a *Replace* operation to replace in the template all the occurrences of parameters to the values specified by the user.

If you look at the template file, 5 lines before the end, this line:

```
<procedure>NI_Initiator_NB_BUFFERS</procedure>
```

will become, if the user specified *NB_BUFFERS=6*:

```
    <procedure>NI_Initiator_6</procedure>
```

You can copy these files (asip+generator+template) to define your own parameterised components.

3.3 Starting a simulation

This section relies on proper configuration of the compilation and simulation scripts, as described in Section 2.2 "Back-end configuration (Compilation&simulation scripts)".
The following sequence of steps usually gets your simulation running:

- Select each/all the components linked to source code files (i.e. the 'leaf' components, which do not contain any sub-component)
- In the property view, select the desired simulator on the ***Target*** row.
- Check inside the Simulation targets view that only 1 item is shown (the one corresponding to your selected simulator). Especially avoid having an empty rectangle: it means that some components don't have assigned targets (and they will just get ignored during the simulation, resulting in undefined components). If you see an empty rectangle, click on it, and select ***Select simulated components*** from the ***Property view*** in order to select all the attached components. Then select a proper simulation target for them.
- Click the ***Play*** icon, or select ***Run Simulation*** from the ***Run menu***.
- Two windows should come up: Tool flow window and Execution window.
- Your simulation results/errors should appear in the Execution window.
- The Tool flow window shows the tools that are being run, and can be used to run the tools independently if required. It is useful in order to control co-simulations.

3.4 Importing existing Verilog or VHDL projects

3.4.1 VHDL→ASIP

vhdl2asip.sh –d <vhdl project root directory> -t <vhdl top level name>

4. User Config Guide – How to customise the tools

4.1 Galaxy IDE

4.1.1 Adding a new View

TODO: <Edit galaxy-ide.ui>

5. Developer Guide – How to make changes to the source code

5.1.1 Adding a new View

Views (sometimes also called Plugin Views) must derive from *class view* defined in *view.h*.
They must override the virtual method *Gtk::Widget* view::get_gtk_widget()*, which should return the gtk widget of the view. The actual widget construction can be done in this method or somewhere else if the programmer prefers to (sometimes it can be desired to construct the widget in the new view's constructor).

The view will also be able to receive events via the callback method *event_from_event_manager(class event e)*.

5.2 How to properly change the ASIP schema

5.2.1 Updating the version number

The current asip schema is a symbolic link in trunk/xml_schemas from asip.xsd to asip_<version_number>.xsd.

- Create a new asip_<next_version_number>.xsd corresponding to the new version number
- Update the symbolic link to point to the new file
- Each of the following files contain 1 reference each to the asip version number. Search for the old version number between quotes and replace with the new version number:
    - src/asip-generators/noc2asip/noc2asip.cpp
    - src/asip-generators/stg2asip/stg2asip.cpp
    - src/asip-split-sim/main.cpp
    - src/libasip/asip_doc.cpp
- Update every .asip.xml demo file to conform to the new asip version

5.3 How to add a new language to the framework

We will go through a real life example: adding Petri nets to the Galaxy framework.

The main use of our framework being debugging through co-simulation, our first task is to find a Petri net simulator able to handle co-simulation. We will also need to figure out how we define the interface between the Petri net and the other languages.

We actually decided to use a different route, as Petri nets are very simple descriptions which can be converted entirely to ASIP. We therefore simply need a Petri net to ASIP converter.

We decided to use the Petri Net Markup Language (PNML) format as an input format, and we just need to write a pnml2asip tool.

6. Unsorted items

6.1 Global ports and global connections

6.1.1 To reduce clutter

Clock and reset wires are usually spreading across the whole design. This clutters the design view as they are often useless to watch during the debugging process.

In ASIP, we use global ports and global connections to connect modules to these ports without going through the whole hierarchy (a module's port can be connected to the clock signal without going through his parent's interface, so the parent doesn't need any clock port).

We use the key symbol '^' to access those global ports.

Declaration is ASIP:

- The global port is a port defined at the top-level prefixed with '^'.
- TO BE CHANGED TO: The global port is simply a port defined at the top-level, without prefixed '^'.
- When a connection refers to a global port, the referred component is "", and the port name is the global port name, with its prefix.
- TO BE CHANGED TO: When a connection refers to a global port, the referred component is "^", and the port name is the global port name, without prefix.

6.1.2 To handle clocks on multiple simulation&hardware targets

When a component is moved from one simulation target to another, its associated clock generator usually needs to follow the same change, as it is not desirable to transfer clock signals across a co-simulation interface. When multiple components simulated on the same target are moved to different targets, too many adjustements of clock generators become needed.

In order to let the user move components between simulation targets without having to adjust the attached clock generator component, we make special use of the global ports:

- Global ports are allowed to have multiple input connections (i.e. multiple clock generators)
- The only active connection between a clock generator and a component is the one where both ends share the same simulation target.
- If a clock generator doesn't exists for this simulation target, then no connection is made.

6.2 Working with hardware

6.2.1 The problem(s)

6.2.1.1 Problem 0

Can the hardware pins be accessed using our traditional cosimulation components, as inserted by asip-add-cosim-comps?

6.2.1.2 Problem 1

When placing one module on an FPGA, and another module on a connected FPGA, we need a way to indicate to the back-end tools which of the hardware connections we want to use.

6.2.1.3 Problem 2

How do we specify when a module placed on an FPGA needs access to specific FPGA pins in order to access a hardware component?

6.2.1.4 Problem 3

How do we use the FPGA clock?

6.2.1.5 Problem 4

When we plan on using a bus, how do we specify and implement the merging of all signal to a “bus controller”?

This problem needs to be considered either if the user wants to use the bus, or if the ASIP routers need the bus. What if both want to use it together?

6.2.2 Some answers

6.2.2.1 Ideas for problem 1

The hardware wire name (the connection’s name in the ASIP description of the hardware target) is specified as a property of the connection in the user ASIP.

6.2.2.2 Ideas for problem 2

For each desired pin, the user will need to create a connection pointing to the hardware wire.

We need a second component to connect the connection to, and this component needs NOT to be simulated. For the moment we achieve this by giving the component the keyword HARDWARE as its implementation and simulation target names.

6.2.2.3 Ideas for problem 3

Same as Problem 2, with the CLK pin.

6.2.3 Handling from a user’s point of view

6.2.4 Handling from an admin’s point of view

Asip2v will be called with a –t <target name> option, in order to identify and load the proper hardware model.

The hardware model, an ASIP description, is made of ASIP-components representing the hardware modules. ASIP-ports are named after their LOC, so that asip2v can write the proper “//synthesis LOC=”…””. Connections’ names are those used in the user design to identify hardware wires.

6.3 Co-simulation with HLA

For speed and simplicity, we often co-simulate 2 targets just by using 2 files: sim1to2 and sim2to1. Unfortunately, this has the problem of 1. Requiring its own syntax and 2. Not being extendable to more simulators.

It is now time to move all co-simulations to HLA.

A HLA co-simulation goes through the following stages:

- Connection to HLA server
- Publication of all the output wires
- Waiting for all other simulators to connect
- Association of all the input wires to the corresponding output wires published by other simulators
- Simulation run
    - Synchronised by time step (or not?)
    - Changes in inputs should be propagated to the main design
    - Changes to outputs need to be sent over HLA
    - Series of inputs&outputs changes during the same timestep need to be handled correctly
- End of simulation

6.3.1 Waiting for all other simulators to connect

One problem faced is that we don’t know how many other simulators there are.

6.3.2 End of simulation

This is actually very tricky to detect.