spyce Documentation

Release 0.1

Imec NL

May 27, 2019

CONTENTS

Contents:

CHAPTER

ONE

WELCOME TO SPYCE'S DOCUMENTATION!

Overview

The goal of the spyce project is to empower hardware designers with a tool that simplifies the communication between developers, and to take away the tedious work of manually connecting modules and block.

Spyce (**S**imple **py**thon **c**ircuit **e**ditor) is a free, open-source package using Python and Qt to create a schematic, hierarchical view of a hardware design. Spyce also provides a simple, extensible netlist generation to be able to simulate your design. For now the project is mainly targeting digital signal processing (simulink like) and using MyHDL as the main netlisting language, with systemverilog netlisting in the planning.

This project is based on the work of Robert Bucher (http://robertobucher.dti.supsi.ch/python/), and follows his naming conventions that are a bit ununsual in the hardware world. A 'diagram' is a collection of basic elements (blocks, connections and ports) that constitutes a design. This diagram is in fact a schematic drawing. A 'Port' is anything that can be connected to. These can be pins on a block, pins to upper hierarchical levels, or nodes; that connect wires.

prerequisites

Spyce requires: - python 2.7 or 3.5+ - Qt: wrapper around PySide, PySide2, PyQt4 and PyQt5, see https://github.com/mottosso/Qt.py - any of PySide, PySide2, PyQt4 and PyQt5 - myhdl + fixbv extension, see https://github.com/imec-myhdl/myhdl - inkscape to edit icons, and convert to png - openoffice is optional, but default configured as editor for documentation - spyder3 is optional, but default configured as python editor. - spinx and latex if you want to build the documentation

Spyce is developed on python 2.7 and PyQt4

Installing

Install source code:

```
git clone https://github.com/imec-myhdl/pycontrol-gui.git
```

Build documentation:

```
cd pycontrol-gui/doc
make latexpdf # or html
```

Start Spyce:

```
cd pycontrol-gui/BlockEditor
python spyce.py
```

Building components

Blocks

Blocks are symbolic representations of circuit modules. A block must contain a number of standard elements like pins, but can also include optional elements like tooltip or netlisting functions.

Parameters and properties

Blocks can have both parameters and properties. They control the behaviour of the block, The difference is that a parameter can change the *interface* of the blocks: pin names, or number of pins. Properties *can* change the symbol, but *not* the pins. Both properties and parameters are passed to the netlist. A paramerized Block is a block containing a non empty *parameters* variable

Creating Blocks with the GUI

There are two ways to create a Block: either via the *import MyHdl* function or manually via the *Create Block* function.

If you already have a MyHdl file, you can create a Block (symbolic view) by importing it. In the library viewer select the destination library, and then click on the *import Myhdl* icon. The import routine will guess in and outputs, but might get it wrong, so you might want to edit the pins or their positions. Note that any keyword arguments are treated as properties. You can check the created block by right-clicking on the new block and selecting views->blk_source, which will pop up the prefered editor

In the library viewer right-click somewhere in empty space and choose *Create block*. You can choose an icon if you can re-use an existing icon (but most of the time you will create a new one later). You add inputs, outputs, parameters and properties. After clicking OK the block is created in the current library.

When you right click on a block you can choose:

edit icon to edit the icon, or generate an empty svg drawing when it is not yet present.

Edit pins to edit pin-names and pin-positions, or add/delete pins

Change BBox to edit the bounding box

Views will allow you to edit any of the other views

Creating Blocks from code

Blocks are simple python files that are inside a 'library'. The blocks should *not* import Qt as the netlister must be able to import them, without a gui. If Qt is needed or modules that import Qt the easiest solution is to import them in the function that needs them (but never in the main, or netlist routines). An Example:

```
# cell definition
# name = 'myblock'
# libname = 'mylibrary'

tooltip = 'This is an empty block with inputs a and b and output z'

inp  = [('a', -40, -20), ('b', -40, 20)]
outp = [('z', 40, 0)]
io   = []
bbox = None

parameters = {} # pcell if not empty
properties = {} # netlist properties
```

```
#view variables:
views = {'icon':'myblock.svg'}
```

inp

outp

inout

inp, outp, inout are lists of tuples (pinname, x, y). The pinname can start with a '.' to indicate that the label should not be displayed. Alternatively *inp, outp, inout* can be an integer, being the number if inputs/outputs or inouts. The actual pins will be named '.i_0', '.i_1' etc. for inputs, '.o_0', '.o_1' etc. for outputs, or '.io_0', '.io_1' etc. for inouts. Note: inouts are optional and not yet properly implemented.

tooltip

tooltip is an optional string that will be displayed when the mouse hoovers on the block.

views

views is a dictionary that contains all (other) views. If *views['icon']* is defined it looks for an svg file in either the *resources/blocks* directory (when no extension is specified) or in the same directory (library) as the block code otherwise.

bbox

bbox is either *None*, or a 4-tuple: *(left, top, width, height)*

ports (*param*)

This (optional, but highly recommended) function must return a tuple (inp, outp, inout), based on the parameters in the dictionary 'param'. Each of inp, outp, inout is a list of tuples (pinname, x, y). The pinname can start with a '.' to indicate that the label should not be displayed

getSymbol (*param*, *properties*, *parent=None*, *scene=None*)

This function returns a `Block` object. It is mandatory for parametrized blocks. The getSymbol function will probably start with importing the block class, and Qt

toMyhdlInstance (*instname*, *connectdict*, *param*)

This function should return a properly indented string (4 leading space) containing the MyHDL code. It is required for myhdl netlisting a parametrized blocks. The instance name is the name of the block in the diagram. Connectdict is a dictionary with connections and properties (connectdict[pinname] = nettname or connectdict[property_name] = property_value). This choice was made since they are both elements of an instantiation e.g.:

```
block1_instance = myBlock(signal_1, signal_2, property_1=42)
```

note: spyce always netlists with the *pin_name = connected_signal_name* syntax to remove all ambiguity

toSystemVerilogInstance (*instname*, *connectdict*, *param*)

This function should return a properly indented string (4 leading space) containing the SystemVerilog code. It is required for SystemVerilog netlisting a parametrized block. The instance name is the name of the block in the diagram. Connectdict is a dictionary with connections and properties (connectdict[pinname] = nettname or connectdict[property_name] = property_value)

After importing a block the following defaults are added:

blockname

This is the name of the module (without the .py extension)

libname

This is the name of the directory of the module (without the *'library_'* prefix)

views

views will be extended with all views that are found (including the block-source itself)

Drawing

Adding blocks

Blocks can be dragged from the library viewer and dropped on the editor. Right-click on a block in the Editor will show all options. (note that adding parameters is not supported, only changing). Input pins are shown as little arrow heads, and outputs as little squares.

Changing the name can be done directly by clicking in the text-label below a block, or via the right-click menu. The block can be flipped in the x direction, to enhance the readability.

Double clicking on a block will open the diagram view (if there is one), effectively descending one level

Adding connections

To draw a connection you have to click on a block output, a node or an input pin. Double click will end the connection with the insertion of a node, and will pop up the edit-node menu to add a label or other properties. A connection can also be finished by clicking on a block-input, node or output pin

Adding pins

To add a pin click on the pin symbol in the tool bar at the top of the editor. When the mouse is moved to the drawing area and clicked the pin will be 'dropped' on the place of the cursor. Right click on the pin will show an edit menu (to add a pin label, or change the pin-type) and an entry to add a signalType.

signalType can be any myhdl type, like *intbv(0,min=-12,max=12)* or *bool(1)*. It is used to initialize the Signal() of the net for the netlisting.

pin-label should either be a valid identifier (starting with a letter or underscore) or a number. In the latter case the net is take to be a constant rather than a Signal during netlisting.

Adding comments

comments can be added by clicking on the text-balloon symbol in the tool bar at the top of the editor. When the mouse is moved to the drawing area and clicked the comment will be 'dropped' on the place of the cursor.

Double cicking on any editable label will pop up a font configuration diagram.

TODO

The drawing tool works, but is rather unpolished. Some features that I would like to see:

- proper Undo: There has been some work, but when moving a component, Qt constantly emits 'changed' signals and that will spam the redrawing. Probably it is possible to suppress these whith button down/up
- Git support: it should be possible to checkin/revert any block or diagram
- connections: Connections as line-segments, but moving a segment should also move the vertices of the attached segments. This is not so straight-forward as it looks, especially when a segment is connected to a node or block-port
- blocks: the interface to a block is rather clumsy, passing attributes as a dictionary, where python **kwargs approch already provides the same functionality. Changing however is quite some work due to all the libraries

- inout ports: There are some hooks in the code to implement inout, but for now they are low priority.
- rotation: flipping is supported, it should not be too hard to add rotation (block, label, pin)
- Probes: It is simple to hook-up any simulator, but the postprocessing requires ‘software’ measurement devices. These can be ‘volt’ meters, oscilloscopes, spectrum-analyzers, communications analyzer (i.e. a software receiver) etc.
- Sources: like probes there is also a need for software clock-sources, vector/data-generators etc.

CHAPTER

TWO

INDICES AND TABLES

- genindex
- modindex
- search

CHAPTER

THREE

INDICES AND TABLES

- genindex
- modindex
- search