ARROWS

USER MANUAL

Version 1.0

AUTHOR: Michael Wieckowski

CONTACT: wieckows@umich.edu

Introduction

What is Arrows?

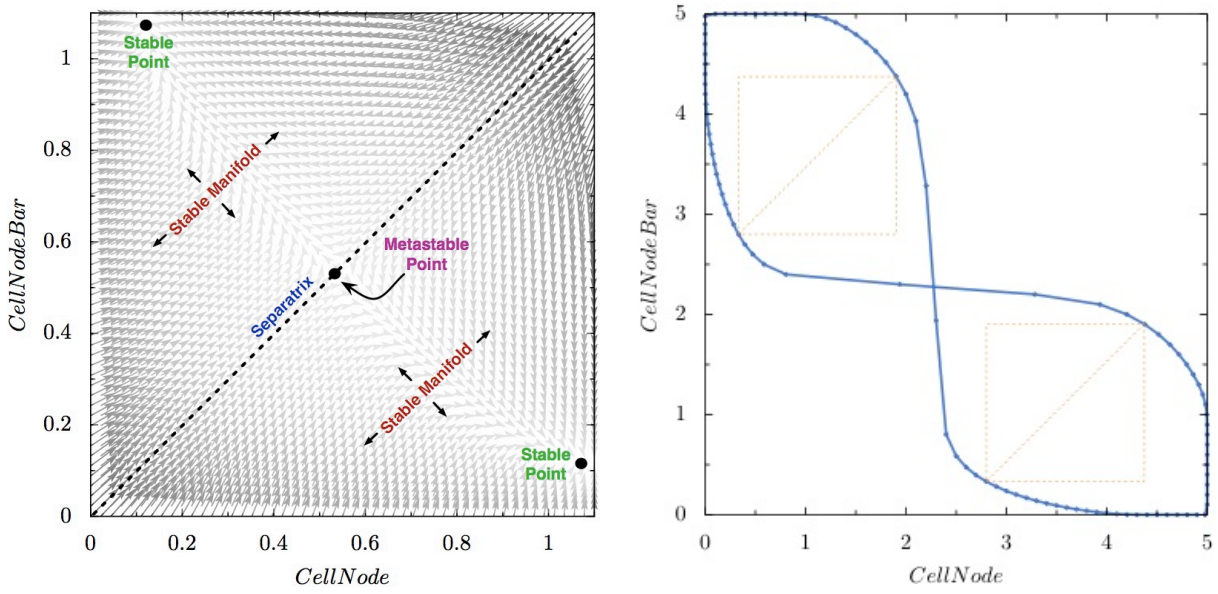
Arrows is a Python scripting package for performing vector field (dynamical) analysis on bistable circuits using the Spectre simulator. Arrows is highly object oriented and abstracted allowing for complete independence of circuit structure, technology node, environment, and analysis type. This allows Arrows to perform "black box" analysis of any bistable circuit (design for SRAM cells) to:

- Generate butterfly curves
- Calculate noise margins
- Trace separatrix lines
- Perform dynamic stability analysis
- Calculate the Separatrix Affinity metric
- Visualize "quasi-transient" simulations

Two examples are shown in the figures below.

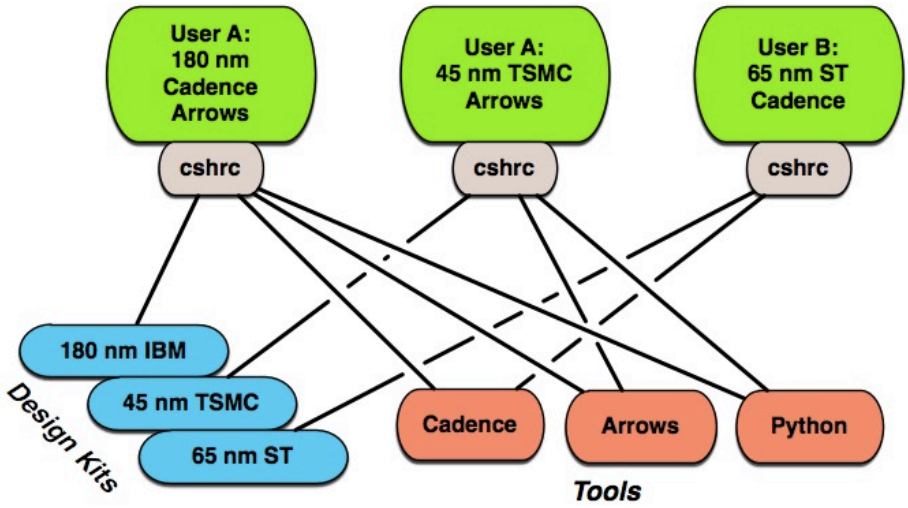
One of the primary goals of Arrows is to be highly extensible while maintaining independence of technology and circuit topology. To that end, circuit structures and technology nodes (design kits) are modeled and wrapped in customizable XML files. The resulting design framework makes Arrows highly readable, easy to learn and implement, and well suited to large multi-user projects.

Arrows is being developed rapidly and new features are being added daily. In addition, I welcome any feedback and feature requests from the VLSI community, so don't hesitate to ask if you think function X is a must have!



Setup

Process technology independence and user independence is a MUST for any successful VLSI project. To that end, I recommend setting up your design environment as shown in the following figure. In essence, all design kits and tools are installed in isolated locations to be shared among all users. Each user maintains separate directories for each of their projects, where each directory has a shell script to link the required tools for that project only. The following setup instructions for Arrows are based on this model.



Installing Python

Since Arrows is a tool for layout scripting in Python, it requires a recent version of Python to be installed (2.6 or higher). Most linux distributions will have a version of Python installed for you to use, but the version may not be recent enough. Luckily, you can install a local copy of Python on top of any existing installations if the machine you want to run on doesn't meet the requirements. I recommend installing a copy of Python 2.6 locally as follows:

1. Download Python via SVN and decompress it as follows:

    1.1. cd /some/tempPath/

    1.2. wget http://www.python.org/ftp/python/2.6.1/Python-2.6.1.tgz

    1.3. tar xvzf Python-2.6.1.tgz

2. Install Python into your home directory (or a project directory):

    2.1. cd /some/tempPath/Python-2.6.1

    2.2. ./configure –prefix /some/localPath/forPython/toReside

2.3. make

2.4. make install

Installing Arrows

These instructions will install Arrows into a central location.

1. Download the Arrows zip from http://www.vlsitools.com
2. mkdir /design/common/Arrows
3. mv /download/location/Arrows_1.0.zip /design/common/Arrows
4. cd /design/common/Arrows
5. unzip Arrows_1.0.zip
6. rm Arrows_1.0.zip

The remainder of the "installation" involves copying and modifying the shell script in /design/common/Arrows/exampleUserDir. This is covered in the section "Setup a directory to work from" below.

Installing the NCSU CDK

All the examples in this manual are based on the NCSU Cadence Design Kit. Since it is freely available, it is a good platform to introducing Arrows while avoiding any IP issues. Arrows will work with ANY technology and is designed specifically to do so. The following steps were used to install the NCSU CDK on my system (yours might be different, so read carefully.)

1. Register and download the CDK from:
   http://www.eda.ncsu.edu/wiki/NCSU_CDK_download
2. I consider it good practice to keep the CDK itself in a central location that is referenced by each user's working directory. Untar / Unzip the downloaded CDK into your central location. For my case:

   2.1. cp ncsu-cdk-1.5.1.tar.gz /design/common

   2.2. cd /design/common

   2.3. tar -xvvzf ncsu-cdk-1.5.1.tar.gz

Configuring tech.xml

To isolate technology specific configuration from analysis and circuit design, Arrows makes use of a tech.xml file. A global version of this file lives in /design/common/Arrows/config. A system administrator could setup a tech.xml for all of the globally installed technologies. A user may also have their own tech.xml located anywhere the wish. Upon creating a technology object in an Arrows script, you simply choose whether to use the global tech.xml (default) or the local copy.

A tech.xml file contains several important elements. Each technology you want to give Arrows access to is bracketed by <technology> tags. The tags have a property called *name* which is a concatenation of the Vendor, the Node, and the Version. For example, if I have a technology available to me from TSMC that is a 45nm node and version 1.2, I would add to my tech.xml:

<technology name=”TSMC45nm1.2”>

</technology>

Inside of these technology tags, two types of elements need to be added. The first is a model path. Each model path tag corresponds to a path that is included at the top of any simulation using this technology. You can have as many model path tags as you'd like, and each one can contain an optional paramter called *section*. For example:

<modelPath section = “tt”>/common/ncsu-cdk-1.5.1/models/spectre/standalone/ami06P.m</modelPath>
<modelPath>/common/ncsu-cdk-1.5.1/models/spectre/standalone/ami06N.m</modelPath>

After the model paths, the technology must contain a list of device templates. The device templates tell Arrows how to write instances of a device for a particular technology. The best way to explain the device template format is with an example:

<deviceTemplate name="MOSFET">

<modelAlias model="ami06N">nmos</modelAlias>

<modelAlias model="ami06P">pmos</modelAlias>

<modelAlias model="ami06N">sramPullDown</modelAlias>

<modelAlias model="ami06P">sramPullUp</modelAlias>

<modelAlias model="ami06N">sramPassGate</modelAlias>

<netlistFormat simulator="spectre">

M{uniqueName} ({drain} {gate} {source} {body}) {model} w={width} l={length} as={ami600areaSourceDiffusion} ad={ami600areaDrainDiffusion} ps={ami600perimeterSourceDiffusion} pd={ami600perimeterDrainDiffusion}

</netlistFormat>

</deviceTemplate>

First, the device template is given a name. In this case, *MOSFET* is used. Any time an Arrows script is using this technology, it can instantiate a device called *MOSFET*. Clearly, no simulator will know what a *MOSFET* is - therefore, the next section of the device template includes model aliases. Model aliases map a device type to an actual model name. So if an Arrows script uses a device called *MOSFET* and it's type is *sramPullUp*, the simulation netlist will actually get a device model called *ami06P*. The beauty of this system is that I can specify a new technology with a device template called *MOSFET* and a type called *sramPullUp*, but a completely different model alias. Doing so allows my circuit to work with ANY technology.

The last part of the device template block is the netlistFormat tag. This tag tells Arrows how to create a netlist for a particular type of simulator using this specific technology. Items surrounded by { } will be replaced with properties for that device instance. This allows us to have simulator independence in addition to technology independence.

Making a circuit in XML format

Circuits for analysis in Arrows are also specified in XML format. This method was chosen to maintain technology independence while avoiding complex Python specification of circuit objects. Every circuit is bracketed by <circuit> tags with a name property. Inside of the circuit, <inputOutput> tags are used to specify the inputs and outputs to the circuit when instantiating it in a netlist. Similarly, <parameter> tags can be added to create variables within the circuits definition.

The circuit structure is composed of a series of <device> tags. Each tag contains a type, defined in the tech.xml file, and a uniqueName, to ensure that each device is unique. The <device> tags bracket a series of <modelAlias>, <property>, and <expression> tags. <modelAlias> specifies the alias to use in the tech.xml file. For example, <modelAlias>sramPullDown</modelAlias> will result in device with an ami06N model based on the tech.xml above. <property> tags specify the properties that will be substituted into the { } elements in the tech.xml's <netlistFormat> tag. If a property is added that does not exist in the <netlistFormat>, it will be ignored. **On the other hand, if a <netlistFormat> element does not have a corresponding property, an error will occur.** Finally, <expression> tags can be used when one property is a function of another.

For example, a sourceArea property is generally a function of the device's width property. Therefore, an expression is used since the sourceArea property cannot be determine a priori. Take a look at the example circuits to get an idea of what a complete circuit definition looks like.

Setup a directory to work from

Your working directory will be the location where you run Arrows, Cadence, etc. In this manual, we will setup the working directory to use the NCSU CDK installed above.

1. Create a directory specific to the process node you want to use. In my case, this will be AMI Semiconductor 0.6 micron process.

   1.1. mkdir ~/design/600nmAmi

2. Copy the CDS initialization and library files from the CDK common directory into your working directory.

   2.1. cp /design/common/ncsu-cdk-1.5.1/.cdsinit ~/design/600nmAmi/

   2.2. cp /design/common/ncsu-cdk-1.5.1/cdssetup/cds.lib ~/design/600nmAmi/

3. The files we copied in step 2 rely on some environment variable to work properly. Since these variables are specific to the NCSU CDK, we will NOT put them in our global shell initialization file. Instead, make a local file that you will source every time you want to use the design kit. For my case, using CSHELL:

   3.1. cd ~/design/600nmAmi

   3.2. for vi, use vi 600nmAmi.cshrc. Alternately, use your favorite editor.

   3.3. Enter the following lines into the file, changing the paths where appropriate:

```
setenv SYSTEM_CDS_LIB_DIR /tools/ic-5.141_usr5/tools/dfII/samples
setenv CDSHOME /tools/ic-5.141_usr5
setenv CDK_DIR /design/common/ncsu-cdk-1.5.1
setenv CDS_MMSIM_DIR /tools/mmsim-7.0
setenv CDS_INST_DIR /tools/ic-5.141_usr5
```

4. Now we need to copy a few files for Arrows. Take a look in the /design/common/Arrows/exampleUserDir. This directory contains what a typical work directory might look like to Arrows. For this setup, let's just copy it all into our work directory.

    4.1. cp -r /design/common/Arrows/exampleUserDir/* ~/design/600nmAmi

    4.2. cd ~/design/600nmAmi

    4.3. Edit the arrows.cshrc file according to the instructions inside. The most important variables to set are PATH, PYTHONPATH, and MMSIM_PATH. Without these, you will likely have the wrong version of Python running and your scripts will not know where to find Arrows. More importantly, you will not be able to run any of the simulations required without command line access to Spectre.

Quick Start Example

The included quick start example demonstrates how to setup and run a basic vector field analysis of a 6T SRAM cell in the AMI 600nm technology. The vector field is generated, and from it, butterfly curves are generated and a static noise margin measurement is taken. The state space with the superimposed butterfly curves is plotted in a PDF file for viewing.

The Arrows Quick Start Script

The quick start script is located in the exampleUserDir directory. It gives a step by step example of using Arrows to do some basic analysis.

To run the script:

1. cd ~/design/600nmAmi
2. edit quickStart.py so that the paths are real. For example, replace "~/design/600nmAmi" with an actual path, such as "/users/me/home/design/600nmAmi"
3. make sure the paths in arrows.cshrc are properly configured
4. source arrows.cshrc
5. python quickStart.py

Let's go over the script piece by piece to learn the basics of Arrows. Lines 1 through 29 setup the Arrows environment and technology objects. The environment object is instantiated in like 12 and initialized with a global supply voltage of 5.0 volts and a unique name that is a function of today's date. The unique name prevents overwriting of intermediate data when you run multiple Arrows scrips at the same time.

The AMI600 technology object is created and initialized in Line 18. It specifies the name of the technology as well as the version. The vendor, node, and version are concatenated internally to generate a unique identifier for the technology in the tech.xml file. For info on the tech.xml file, check the section above entitled "Configuring tech.xml". Line 22 simply tells the technology object to go ahead and scan the XML file for devices, models, formats, etc. Since the AMI technology does not contain special models or devices for voltage sources, we also include a "generic technology" which defines these basic circuit elements.

```
1.  import arrows
2.  from datetime import *
3.
4.  ##
    ********************************************************************************************
5.
6.  ## Set up the analysis environment.  This is where you
7.  ## specify a "uniqueId" to your controller so that
8.  ## it won't overwrite other files and you can run multiple
9.  ## at the same time.  Also specify the global
10. ## supply voltage.
11. idViaDate =
    str(datetime.now().month)+'-'+str(datetime.now().day)+'-'+str(datetime.now().year)
12. analysisEnv = arrows.Environment(uniqueId = idViaDate,
13.                           globalSupply = 5.0)
14.


15. #to generate the netlists, we use Technology objects which are configured using XML descrip-
    tions
16. #of the process technology, it's paths, model templates, etc.  Refer to config/tech.xml
17. #For this example, we are using a 0.6 micron technology from AMI in the NCSU design kit
18. amiTechnology = arrows.Technology(vendor = "Ami",
19.                                   node = "600",
20.                                   version = "a",
21.                                   environmentToUse = analysisEnv)
22. amiTechnology.configureFromXml()
23. #we also need a "generic technology" which defines standard circuit elements such as voltage
    and current sources
24. genericTechnology = arrows.Technology(vendor = "Generic",
25.                                   node = "",
26.                                   version = "",
27.                                   environmentToUse = analysisEnv)
28. genericTechnology.configureFromXml()
29.
```

Lines 30 through 35 create a circuit object. Our amiTechnology is linked to the circuit during initialization, and then the configureFromXml method is called. This method generates an internal representation of the circuit (in this case, a 6 transistor SRAM cell) and will eventually allow the circuit object to output subcircuit and instance statements specific to its linked technology. In this way, technology independence is achieved and the linked technology can be changed at any point.

```
30. #The bistable circuit we will analyze in this example is a standard 6T SRAM cell.
31. #The cell circuit is described in the circuits/sramCell6T.xml file
32. #We use it by creating a generic circuit object and then configuring it to be an SRAM cell
33. sixCell = arrows.Circuit(technologyToUse = amiTechnology)
34. sixCell.configureFromXml(analysisEnv.modelPath+"/circuits/sramCell6T.xml")
35.
```

Lines 36 through 54 define the test bench that will be used during the vector field analysis. The testbench object is first instantiated and initialized with the generic technology (line 38). A single DC voltage source is then added. This source is the global supply voltage and uses the value specified in the environment object.

In order to connect the cell properly in the testbench, lines 46 through 50 specify the connections for the inputs and outputs in the cell's XML definition. For the standby case, the bitlines and power node are connected to the DC global supply. All other inputs are grounded. Lastly, line 53 is included as a demonstration of how to modify the circuit's properties, even when configured using the XML file. With this type of scheme, one could easily run simulations where device size, threshold mismatch, etc. are parameterized.

```
36. #We will analyze the 6t SRAM cell using a testbench.  This allows us to specify bitline,
    wordline, and supply
37. #voltage values.  For this example, we will simply create 1 DC voltage source and connect
    everything to it.
38. sixTestbench = arrows.Testbench(technologyToUse = genericTechnology)  #make a new testbench
    here
39. #all we need a supply for vdd!.
40. sixTestbench.addDcVoltageSource(uniqueName = "0",
41.                                 plusNode="vdd!",
42.                                 minusNode="0",
43.                                 value=analysisEnv.globalSupply)
44.
45. ## Now set up the testbench connections to the cells
46. sixCell.inputsOutputs["leftBitline"] = "vdd!"   #vdd! for standby, 0 for write
47. sixCell.inputsOutputs["rightBitline"] = "vdd!"
48. sixCell.inputsOutputs["wordline"] = "0" #0 means standby mode.  set this to vdd! for either
    read or write
49. sixCell.inputsOutputs["power"] = "vdd!"
50. sixCell.inputsOutputs["ground"] = "0"
51.
52. #if we want to modify some properties of the circuit, we can do it here to override the XML
    configuration
53. sixCell.devices["leftPullDown"]["length"] = "1u"
54.
```

Lines 55 through 72 run the dynamical analysis (vector field analysis). The analysis object is instantiated and initialized using our 6T circuit, the AMI technology object, and an analysis environment. In addition, the names of the bistable nodes are given as a string list. These two nodes define the state space of the analysis. Lastly, a numberOfSteps is given which controls how many divisions the state space is quantized into for each dimension. After the vectorField simulation is actually executed in line 68, a separate method in line 72 is invoked to parse and clean up the intermediate simulation files. If cleanUpWhenDone is set to zero, all intermediate files will be left in the input directory for debugging purposes.

```
55. #The actual analysis is a VectorField analysis object.
56. #Initialize this object using our SRAM object, technology, etc.
57. #The bistableNodes are a list of the 2 nodes considered in the state space
58. #The numberOfSteps is the quantization parameter for the state space
59. analysis = arrows.VectorField(cellToAnalyze = sixCell,
60.                                bistableNodes = ["data","dataBar"],
61.                                technologyToUse = amiTechnology,
62.                                numberOfSteps = 50,
63.                                environmentToUse = analysisEnv,
64.                                testbenchToUse = sixTestbench)
65.
66. #Run the actual sim
67. #make sure that the .cshrc is sourced beforehand, or else spectre will fail
68. analysis.runVectorFieldSim()
69.
70. #Parse the output from the sim and populate our analysis object with the results
71. #cleanUpWhenDone determines whether or not we leave the intermediate files in the input/
    output directories
72. analysis.parseVectorFieldSim(cleanUpWhenDone = 1)
```

Lines 73 through 81 execute some analyses on the internal vector field data. The nullclines (butterfly curves) are extracted from the vector field in line 74. Based on the nullcline intersection, the metastable point is determined in line 77. Once that is found, the separatrix could be traced as in line 80. This is commented out for our example to save time. Notice that these commands are interdependent - one cannot trace the separatrix without first finding the metastable point, and one cannot find this point without first tracing the nullclines.

```
73. #Find the null clines (corresponds to the DC transfer curves of a standard buterfly simula-
    tion)
74. analysis.findNullclines()
75. #Find the metastable point by intersecting the nullClines
76. #Can't do this until the nullClines have been located
77. analysis.findMetastablePoint()
78.
79. #Trace the metastable point along the separatrix back to the boundaraies
80. #analysis.traceSeparatrix()
81.
```

Lines 82 through 108 encapsulate the visualization portion of the quick start script. First the SNM's are calculated using the smallest square method in line 85. It is important to note that printing the SNM value to the terminal is not the only function of this method call. Line 99 will not be able to properly draw the SNM boxes if it is not first calculated.

A VectorPlot object is instantiated and initialized in line 88. The analysis and its environment are passed in during this process. This object effectively creates the two dimensional state space plot internally for eventual output as a PDF or EPS file. Line 91 is commented out, but could be used to plot all of the vectors in the field. Lines 94 through 96 set the drawing color and plot the nullclines. For each dynamical analysis, there are two nullclines, U and V. Plotting both is

equivalent to plotting the cell's butterfly curves. Lines 102 through 104 are used to plot the metastable point and the separatrix, if it was traced. Lastly, line 108 outputs a PDF file in the output directory containing our data.

```
82. ## ************************** Now Create some output to look at **********************
83. ##
84. # Calculate the SNM as the least square fitting in the butterfly curve
85. print "SNM Vectors: "+repr(analysis.calculateSnm())
86.
87. #To plot some state space curves in a PDF or EPS file, we need a VectorPlot object
88. myVectorPlot = arrows.VectorPlot(analysis,analysisEnv)
89.
90. #Uncomment the next line to plot the actualy vector field
91. #myVectorPlot.drawNormalizedVectorField()
92.
93. #Set the color and plot the nullclines (butterfly curve)
94. myVectorPlot.setHexColor("#3E6BCE")
95. myVectorPlot.drawNullclineU()
96. myVectorPlot.drawNullclineV()
97. #Set the color and plot the "least square box" used to calculate SNM
98. myVectorPlot.setHexColor("#FCA12C")
99. myVectorPlot.drawSnmRulers()
100.
101.#Set the color and draw the metastable point and separatrix
102.myVectorPlot.setHexColor("#2BFC32")
103.myVectorPlot.drawMetastablePoint()
104.#myVectorPlot.drawSeparatrix()
105.
106.#Finally, put everything into a PDF file in our output directory
107.print "Creating PDF files"
108.myVectorPlot.createPdf(analysisEnv.outputPath+"/field"+analysisEnv.uniqueId+".pdf")
```

Overlay Example

The Arrows overlay example demonstrates how Arrows can analyze two different cell types under the same technology, testbench, and environment. The butterfly curves and SNM calculations for both cells are plotted on the same state space graph for easy comparison.

The Arrows Overlay Script

Lines 1 through 22 are the same as in the previous example. The environment, AMI600 technology, and generic technology objects are all instantiated and initialized.

```
1.  import arrows
2.  from datetime import *
3.
4.  ##
    ********************************************************************************************
5.  ## Notice this time, we just use a string as our unique ID
6.  analysisEnv = arrows.Environment(uniqueId = "overlayExample",
7.                               globalSupply = 5.0)
```

```
8.
9.  amiTechnology = arrows.Technology(vendor = "Ami",
10.                                   node = "600",
11.                                   version = "a",
12.                                   environmentToUse = analysisEnv)
13. amiTechnology.configureFromXml()
14. genericTechnology = arrows.Technology(vendor = "Generic",
15.                                   node = "",
16.                                   version = "",
17.                                   environmentToUse = analysisEnv)
18. genericTechnology.configureFromXml()
19.
20. #In this example, we will analyze 2 different sram cells and overlay their results on one
    plot
21. sixCell = arrows.Circuit(technologyToUse = amiTechnology)
22. sixCell.configureFromXml(analysisEnv.modelPath+"/circuits/sramCell6T.xml")
```

Line 23 is the first difference in this example where a second cell circuit is instantiated for a 7T cell. The same technology is used, but a local circuit XML file is used to define the portless structure.

```
23. portlessCell = arrows.Circuit(technologyToUse = amiTechnology)
24. portlessCell.configureFromXml("./circuits/portlessCell.xml")  #notice, the portless cell is
    defined in the user directory
25.
```

Lines 26 through 49 setup the testbench and connect the cells' inputs and outputs. Notice that a second DC source is added to allow us to apply an AXS voltage to the portless cell that is less than vdd!. This connection is reflected in the inputsOutputs list elements.

```
26. testbench = arrows.Testbench(technologyToUse = genericTechnology)  #make a new testbench here
27. testbench.addDcVoltageSource(uniqueName = "0",
28.                                   plusNode="vdd!",
29.                                   minusNode="0",
30.                                   value=analysisEnv.globalSupply)
31. #add an additional voltage source to act as the portless AXS signal
32. testbench.addDcVoltageSource(uniqueName = "1",
33.                                   plusNode="axs",
34.                                   minusNode="0",
35.                                   value=3.0)   #value is lower than the global supply
36.
37. ## Now set up the testbench connections to the cells
38. sixCell.inputsOutputs["leftBitline"] = "vdd!"   #vdd! for standby, 0 for write
39. sixCell.inputsOutputs["rightBitline"] = "vdd!"
40. sixCell.inputsOutputs["wordline"] = "vdd!" #do a read SNM this time
41. sixCell.inputsOutputs["power"] = "vdd!"
42. sixCell.inputsOutputs["ground"] = "0"
43.
44. portlessCell.inputsOutputs["leftBitline"] = "vdd!"
45. portlessCell.inputsOutputs["rightBitline"] = "vdd!"
46. portlessCell.inputsOutputs["axsLine"] = "axs"
```

```
47. portlessCell.inputsOutputs["power"] = "vdd!"
48. portlessCell.inputsOutputs["ground"] = "0"
49.
```

Lines 50 through 73 perform the same analysis instantiation, execution, and visualization as in the quick start example.

```
50. #The actual analysis is a VectorField analysis object.
51. #Initialize this object using our SRAM object, technology, etc.
52. #The bistableNodes are a list of the 2 nodes considered in the state space
53. #The numberOfSteps is the quantization parameter for the state space
54. analysis = arrows.VectorField(cellToAnalyze = sixCell,
55.                                bistableNodes = ["data","dataBar"],
56.                                technologyToUse = amiTechnology,
57.                                numberOfSteps = 50,
58.                                environmentToUse = analysisEnv,
59.                                testbenchToUse = testbench)
60. analysis.runVectorFieldSim()
61. analysis.parseVectorFieldSim()
62. analysis.findNullclines()
63. ## ************************** Now Create some output to look at ***********************
64. ##
65. # Calculate the SNM as the least square fitting in the butterfly curve
66. print "SNM Vectors: "+repr(analysis.calculateSnm())
67. myVectorPlot = arrows.VectorPlot(analysis,analysisEnv)
68. myVectorPlot.setHexColor("#3E6BCE")
69. myVectorPlot.drawNullclineU()
70. myVectorPlot.drawNullclineV()
71. myVectorPlot.setHexColor("#FCA12C")
72. myVectorPlot.drawSnmRulers()
73.
```

Lines 74 through 89 swap out the sixCell object for a portlessCell object in the analysis. Everything is run again for the new cell and plotted with some different colors. Lines 91 through 94 output the two overlayed butterfly curves as a single PDF file.

```
74. ## ************************** Now we will run the analysis again, but with the Portless
    cell ***********************
75. ##
76. analysis.cellToAnalyze = portlessCell
77. analysis.runVectorFieldSim()
78. analysis.parseVectorFieldSim()
79. analysis.findNullclines()
80. analysis.calculateSnm() #need to do this in order to draw SNM rulers
81. #and now plot the new results in a different color
82. myVectorPlot.setHexColor("#56CF3E")
83. myVectorPlot.drawNullclineU()
84. myVectorPlot.drawNullclineV()
85. myVectorPlot.setHexColor("#FCA12C")
86.
87. myVectorPlot.drawSnmRulers()
88.
89.
```

```
90. #Finally, put everything into a PDF file in our output directory
91. print "Creating PDF files"
92. myVectorPlot.createPdf(analysisEnv.outputPath+"/overlayExample.pdf")
93.
94.
```

Parametric Example

The parametric Arrows example demonstrates how Arrows can be used to sweep an analysis parameter and provide output as a text file instead of a PDF file.

The Arrows Parametric Script

Lines 1 through 42 are the same as in the quick start example.

```
1.  import arrows
2.  from datetime import *
3.
4.  ##
    ********************************************************************************************
5.  ## Notice this time, we just use a string as our unique ID
6.  analysisEnv = arrows.Environment(uniqueId = "parameterExample",
7.                               globalSupply = 5.0)
8.
9.  amiTechnology = arrows.Technology(vendor = "Ami",
10.                                   node = "600",
11.                                   version = "a",
12.                                   environmentToUse = analysisEnv)
13. amiTechnology.configureFromXml()
14. genericTechnology = arrows.Technology(vendor = "Generic",
15.                                   node = "",
16.                                   version = "",
17.                                   environmentToUse = analysisEnv)
18. genericTechnology.configureFromXml()
19.
20. sixCell = arrows.Circuit(technologyToUse = amiTechnology)
21. sixCell.configureFromXml(analysisEnv.modelPath+"/circuits/sramCell6T.xml")
22.
23. testbench = arrows.Testbench(technologyToUse = genericTechnology)  #make a new testbench here
24.
25. ## Now set up the testbench connections to the cells
26. sixCell.inputsOutputs["leftBitline"] = "vdd!"   #vdd! for standby, 0 for write
27. sixCell.inputsOutputs["rightBitline"] = "vdd!"
28. sixCell.inputsOutputs["wordline"] = "vdd!" #do a read SNM this time
29. sixCell.inputsOutputs["power"] = "vdd!"
30. sixCell.inputsOutputs["ground"] = "0"
31.
32. #The actual analysis is a VectorField analysis object.
33. #Initialize this object using our SRAM object, technology, etc.
34. #The bistableNodes are a list of the 2 nodes considered in the state space
35. #The numberOfSteps is the quantization parameter for the state space
36. analysis = arrows.VectorField(cellToAnalyze = sixCell,
37.                                bistableNodes = ["data","dataBar"],
```

```
38.                                 technologyToUse = amiTechnology,
39.                                 numberOfSteps = 50,
40.                                 environmentToUse = analysisEnv,
41.                                 testbenchToUse = testbench)
42.
```

Here, we will create a loop where the supple voltage is reduced in each iteration. Lines 44 through 45 create an output file an write the first line, a tab delimited header. Inside of the loop, a new supply voltage is defined in line 48. Each time through loop, a DC source with the new voltage is added to the testbench. The standard set of analyses are performed to measure SNM, and then the DC source is removed. After each iteration, the SNM and current supply voltage are written to the output file for later use.

```
43. #here, we are going to run a loop to determine SNM vs SupplyVoltage
44. outputFile = open(analysisEnv.outputPath+"/snmVsSupply.txt","w")
45. outputFile.write("Vdd\tSnm\n")
46. for index in range(0,10):
47.     #add the supply voltage with a value based on index
48.     newSupply = analysisEnv.globalSupply - (index*0.02)
49.     testbench.addDcVoltageSource(uniqueName = "supply",
50.                                  plusNode="vdd!",
51.                                  minusNode="0",
52.                                  value=newSupply)
53.     analysis.runVectorFieldSim()
54.     analysis.parseVectorFieldSim()
55.     analysis.findNullclines()
56.     snm = repr(analysis.calculateSnm())
57.     print "Supply:"+repr(newSupply)+" SNM:"+repr(snm)+"\n"
58.     outputFile.write(repr(newSupply)+"\t"+repr(snm)+"\n")
59.     #remove the supply DC source since we add one at the top of the loop
60.     testbench.remove("supply")
61.
62. outputFile.close()
```

Class Reference

Circuit Class

Relevant properties:

inputsOutputs:

- Description:
  - A dictionary whose keys represent inputs and output defined in the subcircuit, and whose values represent their connection in the instance definition
- Default: Empty

devices:

- Description:
  - A dictionary whose keys represent device names specified in the circuit XML, and whose values are a dictionary of properties for that device.
- Default: Empty

parameters:

- Description:
  - A dictionary whose keys represent circuit parameters specified in the circuit XML, and whose values are added to the instance definition.
- Default: Empty

Relevant methods:

__init__(environmentToUse, technologyToUse):

- Description:
  - Constructor method.
- Arguments:
  - *environmentToUse* is an object of type *Environment*
  - *technologyToUse* is an object of type *Technology*
- Return: None

configureFromXml(circuitXmlFilePath):

- Description:
  - When called, populates the circuit object with a definition from the associated XML file.
- Arguments:
  - *circuitXmlFilePath* is an absolute string path to a XML circuit description.
- Return: None

subCircuit([format]):

- Description:
  - When called, creates a subcircuit definition of the Circuit object for inclusion in a simulation netlist.
- Arguments:
  - *format* is a string describing the simulator format. Default and currently only option is "spectre"
- Return: A string containing the full subcircuit definition.

instance(uniqueName):

- Description:
  - When called, returns an instance definition of the Circuit object for inclusion in a simulation netlist.
- Arguments:
  - *uniqueName* is a string to ensure that the instance definition is not repeated.
- Return: A string containing the full instance definition.

Environment Class

Relevant properties:

arrowsPath:

- Description:
  - A string that contains the system environment variable "ARROWS_HOME"
- Default: None

uniqueId:

- Description:
  - A string name assigned to the analysis environment such that multiple or simultaneous runs of an Arrows script will not overwrite each other's data.
- Default: None

globalSupply:

- Description:
  - A float specifying the supply voltage used. This defines the extents of the vector field's state space (x and y axis limits.)
- Default: None

modelPath, configPath, viewPath, inputPath, outputPath:

- Description:
  - Strings that contain absolute paths to the various directories within an Arrows installation.
- Default: None

Relevant methods:

__init__(uniqueId, globalSupply, inputPath, outputPath):

- Description:
  - Constructor method.
- Arguments:
  - *uniqueId* is a name assigned to the analysis environment such that multiple or simultaneous runs of an Arrows script will not overwrite each other's data.
  - *globalSupply* is a float specifying the supply voltage used. This defines the extents of the vector field's state space (x and y axis limits.)
  - *inputPath* specifies a directory to use for input to Arrows if the default /userDirectory/input is unacceptable - keep in mind that when Arrows generates a netlist for simulation, it will put that netlist into this input path.
  - *outputPath* specifies a directory to use for Arrows output if the default /userDirectory/input is unacceptable
- Return: None

Technology Class

Relevant properties:

vendor, node, version:

- Description:
  - Strings from a tech.xml file corresponding to the unique properties of a technology.
- Default:None

modelTypes:

- Description:
  - A dictionary whose keys are read from the tech.xml file for each type of model and whose keys are the string definitions of those types for a netlist file.
- Default:None

modelPaths:

- Description:
  - A list of all the paths required for inclusion in a simulation of a particular technology.
- Default:None

devices:

- Description:
  - A dictionary of device types. Each key points to an internally defined *device* class that contains the properties for model aliasing and netlist formatting.
- Default:None

Relevant methods:

__init__(vendor, node, version, environmentToUse):

- Description:
  - Constructor method.
- Arguments:
  - *vendor, node,* and *version* are string objects used to identify a particular technology definition in an XML file by concatenating all three into a single string internally.
  - *environmentToUse* is an *Environment* object.
- Return: None

generateModelIncludeFile():

- Description:
  - Creates a file for inclusion in a simulation. The file contains all of the model paths for a particular technology.
- Return: None

device(name, format, properties):

- Description:
  - Creates a string for instantiating a device for this technology node into a netlist.
- Arguments:
  - *name* is a unique name string to identify the device in a netlist
  - *fotmat* is a string specifying which netlist format to use. Currently, “spectre” is the only option.
  - *properties* is a dictionary of properties and values corresponding to the device definition in the tech.xml file.

- Return: None

Testbench Class

This class encapsulates the testbench used when running vector field analysis. It requires a technology object and stores statements internally to instantiate testbench elements such as voltage and current sources.

Relevant properties:

technologyToUse:

- Description:
  - A *Technology* object from which basic circuit elements can be instantiated.

Relevant methods:

__init__(technologyToUse):

- Description:
  - Constructor method.
- Arguments:
  - *technologyToUse* is the *Technology* object whose methods are called internally.
- Return: None

addDcVoltageSource(uniqueName, plusNode, minusNode, value):

- Description:
  - Adds an instance of a DC voltage source to the current testbench netlist.
- Arguments:
  - *uniqueName* is the string instance name used in the netlist
  - *plusNode, minusNode* are the string connections for the voltage source. i.e. “vdd!”, “0”
  - *value* is the float value for the DC voltage
- Return: None

writeToFile(fileHandle):

- Description:
  - Adds the testbench netslist to a file via it’s opened file handle.
- Arguments:
  - *fileHandle* is a handle to a ascii text file created using the open() method in Python
- Return: None

remove(uniqueName):

- Description:
  - Removes an element from the testbench netlist.
- Arguments:
  - *uniqueName* is the name of the element to be removed from the internally maintained list of testbench devices.
- Return: None

clear():

- Description:
  - Clear all elements in the internal list of testbnech elements.
- Arguments: None

- Return: None

VectorField Class

This class is the main workhorse of the Arrows framework for vector field analysis. It handles generation of simulation netlists, simulation execution, post processing, and data analysis.

Relevant properties:

cellToAnalyze:
- Description:
  - A *Circuit* object representing a bistable system to analyze.

bistableNodes:
- Description:
  - A list of two strings corresponding to the bistable node names in the circuit netlist. ***These must be in the inputOutput list of the subcircuit definition.

numberOfSteps:
- Description:
  - An integer number of steps by which to quantize the state space in both dimensions.

environmentToUse, technologyToUse, testbenchToUse:
- Description:
  - *Environment, Technology, Testbench* objects used in netlist generation and analysis.

X, Y:
- Description:
  - Two lists of state space X and Y float coordinates. Corresponds to the points in the quantized state space to be used during analysis. Since the space is 2 dimensional, the Y coordinates are repeated every numberOfSteps to ensure that these two lists are the same length.

U, V:
- Description:
  - Two lists of U and V float magnitudes. These lists correspond to the coordinates in the X,Y lists and represent vector components of the state space derivatives at each point. For example, the vector (U[10],V[10]) is the arrow to be plotted at the coordinated (X[10], Y[10]).

magnitudes:
- Description:
  - A list of the float magnitudes (euclidian) of every vector described by the U,V lists.

nullclineUx, nullclineUy, nullclineVx, nullclineVy:
- Description:
  - Lists of x and y coordinates of the U and V nullclines. A nullcline is defined as the collection of points in the state space where one of the vector components are zero, either the U or the V.

metastablePoint:

- Description:
  - A float tuple of the metastable point calculated via the intersection of the U and V null-clines. Since nullclines intersect at 3 points for a bistable system, it is assumed that the metastable point is that point whose x and y coordinates are most similar (defines the point most central in the state space).

stablePointA, stablePointB:

- Description:
  - Two float tuples providing the coordinates of the two stable points calculated using nullcline intersection. See metastable point above.

separatrixPoints:

- Description:
  - A list of tuples containing (x,y) coordinates for separatrix points. Since the separatrix is interpolated, these points are not necessarily in the X,Y coordinate list.

snmRulers:

- Description:
  - A list of ruler coordinates (largest box inside of butterfly curve.) Each ruler is a float tuple of the form (x1,y1,x2,y2) providing the coordinates of the SNM box.

snms:

- Description:
  - A list of the SNMs (noise margins) for each lobe of the butterfly curve using the largest square method. The smallest of these two list elements defines the SNM of the system.

Relevant methods:

__init__(cellToAnalyse, bistableNodes, technologyToUse, numberOfSteps, environmentToUse, testbenchToUse):

- Description:
  - Constructor method.
- Arguments:
  - *cellToAnalyze, technologyToUse, environmentToUse, testBenchToUse* are self explanatory. Each of these objects is passed in during initialization to control the terms of the analysis.
  - *bistableNodes* is a string list of the bistable node names in the circuit to analyze.
  - *numberOfSteps* is the quantization factor of the state space.
- Return: None

runVectorFieldSim():

- Description:
  - Executes a variety of internal methods to create the simulation netlists and run the simulation.
- Arguments:
  - None
- Return: None

parseVectorFieldSim([outputFile], [cleanUpWhenDone]):

- Description:
  - Parses the simulation output to determine vectors from the operating points and nodal capacitance.

- Arguments:
  - *outputFile* is an optional file path string where X,Y,U,V data will be dumped after parsing.
  - *cleanUpWhenDone* is an optional boolean defaulting to true. If false, all intermediate simulation files will be left in the output directory.
- Return: None

findNullclines():

- Description:
  - Searched the vector field for null cline components and stores the result internally.
- Arguments: None
- Return: None

calculateSnm():

- Description:
  - Calculates the noise margins of the system by rotating the nullclines about the 45 degree line, resampling them, and then subtracting them. The result is stored internally.
- Arguments: None
- Return: The minimum float value of the two measured SNMs is returned.

findMetastablePoint():

- Description:
  - Calculates the metastable point by intersecting the nullclines. Assumes that the null-clines have already been determined using the findNullClines() method.
- Arguments: None
- Return: None - result is stored internally as a property.

traceSeparatrix():

- Description:
  - Traces the separatrix (boundary between the two stable manifolds) inside of the vector field using an interpolation algorithm. Assumes that the metstable point has already been found using the findMetastablePoint() method. Separatrix points are stored internally as a property and may not coincide with points in the X,Y state space lists.
- Arguments: None
- Return: None.

measureSeparatrixAreas():

- Description:
  - Measures the area of the state space on either side of the separatrix. Since the separatrix is not always a 45 degree line bisecting the state space, this is a good measure of mismatch / impose skew in the bistable circuit.
- Arguments: None
- Return: A tuple of (ratioA, ratioB) where the sum of the two ratios is 1 and each represents a fraction of the total state space area.

UVatXY(x,y):

- Description:
  - A useful utility method to provide a U,V value for any point (x,y) using two sided vector interpolation.
- Arguments: x,y are floating point coordinates for any value within the state space boundaries.
- Return: A tuple (U,V) of the interpolated vector components.

stablePointClosestToSeparatrix():

- Description:
  - Calculates which stable point is closest to the separatrix.
- Arguments: None
- Return: A triplet (distance, stablePoint, and separatrixPoint)

distanceToSeparatrix(pointToMeasureFrom):

- Description:
  - Calculates the euclidian distance from any point to the separatrix.
- Arguments: pointToMeasureFrom is a float tuple of coordinates.
- Return: A float of the distance in state space units (volts).

distanceToMetastable(pointToMeasureFrom):

- Description:
  - Calculates the euclidian distance from the metstable point to any provided point within the state space boundaries.
- Arguments: pointToMeasureFrom is a float tuple of coordinates.
- Return: A float of the distance in state space units (volts).

integrateSeparatrix():

- Description:
  - Calculates the integral of the state space underneath the separatrix.
- Arguments: None.
- Return: A float integral value.

weightedDistanceToSeparatrix(pointToMeasureFrom):

- Description:
  - Calculates the weighted distance from any point to its closest separatrix point using the U,V data in the vector field. This is the method used to calculate Separatrix Affinity when the pointToMeasureFrom is specified as one of the stable points.
- Arguments: pointToMeasureFrom is a float tuple of coordinates.
- Return: A float of the weighted distance from the measurement point to the separatrix point.

weightedDistanceToMetastable(pointToMeasureFrom):

- Description:
  - Calculates the weighted distance from any point to the metastable point using the U,V data in the vector field.
- Arguments: pointToMeasureFrom is a float tuple of coordinates.
- Return: A float of the weighted distance.

VectorPlot Class

This class is the main visualization tool for Arrows. When linked with a VectorField class, it supports plotting of bistable state spaces, nullclines, SNM rulers, serparatrix traces, etc. using the Python package PyX.

Relevant properties:

vectorField:

- Description:
  - A *VectorField* object to be visualized.

environmentToUse:

- Description:
  - An *Environment* object used to determine the extents of the state space boundary from the globalSupply property.

xLabel, yLabel:
- Description:
  - Strings used to label the axis of the state space.

Relevant methods:

__init__(vectorFieldAnalysis, environmentToUse, [xLabel, yLabel]):
- Description:
  - Constructor method.
- Arguments:
  - *vectorFIeldAnalysis* is the VectorField object to visualize.
  - *environmentToUse* is the Environment object associated with the analysis.
  - *xLabel and yLabel* are optional strings used to generate the plots.
- Return: None

getGraph():
- Description:
  - Utility method used to get the internal representation of the vector field plot. This method is used when you have two different *VectorPlot* objects and you want to combine their data. Used in conjunction with the setGraph() method.
- Arguments: None
- Return: A PyX *graph* object.

setGraph(graphToUse):
- Description:
  - Utility method used to set the internal representation of the vector field plot. This method is used when you have two different *VectorPlot* objects and you want to combine their data. Used in conjunction with the getGraph() method.
- Arguments:
  - *graphToUse* is a *PyX graph* object.
- Return: None.

setColor(red, green, blue):
- Description:
  - Utility method used to set the internal drawing color. Use this method before executing a plot command of any kind to control the result.
- Arguments:
  - *red, green, blue* are floats between 0 and 1 representing the RGB composition of the color.
- Return: None.

setHexColor(hexColor):
- Description:
  - Utility method used to set the internal drawing color. Use this method before executing a plot command of any kind to control the result.
- Arguments:
  - *hexColor* is a string of the format "#A2C80B" where every pair of hex characters represents an 8 bit value for R, G, and B components of the color.
- Return: None.

drawVectorField():

- Description:
  - Adds the U,V arrows to the vector field plot. These arrows can be very lage and may result in unreadable outputs. Refer to the drawNormalizedVectorField()method below.
- Arguments: None.
- Return: None.

drawNormalizedVectorField(normScale, filter):

- Description:
  - Same as drawVectorField() but the arrows lengths are normalized to the longest arrow in the field.
- Arguments:
  - *normScale* is a float factor multiplied by the largest normalized vector length
  - *filter* is a float which "drops" vectors less than this length from the field drawing.
- Return: None.

drawNullclineU() , drawNulclineV():

- Description:
  - Draws the nullclines U or V on the vectorfield.
- Arguments: None
- Return: None.

drawSnmRulers():

- Description:
  - Draws the boxes and diagonals for the SNM largest square methods.
- Arguments: None.
- Return: None.

drawMetastablePoint():

- Description:
  - Places a dot at the metastable point calculated in the vector field.
- Arguments: None.
- Return: None.

drawSeparatrix():

- Description:
  - Draws the separatrix points on the vector field.
- Arguments: None.
- Return: None.

createEps(fileName):

- Description:
  - Exports the PyX graph as an EPS file.
- Arguments:
  - *fileName* is a string path to the output filename.
- Return: None.

createPdf(fileName):

- Description:
  - Exports the PyX graph as a PDF file.
- Arguments:
  - *fileName* is a string path to the output filename.
- Return: None.