Layout API Tutorial

Version 2022.06-SP2, December 2022

Feedback

Contents

1. **Introduction** 5

Object Oriented Programming 5

Python Programming Language 7

Developing Parameterized Cells Using Python 10

Python Source Code Examples 14

Parameterized Cell Development Process 14

2. **MOS Transistor Unit Design Example** 16

Introduction 16

Defining Parameters 17

Processing Parameters 19

Generating Layout 22

Viewing the Generated Layout 26

Verifying the Generated Layout 31

3. **Complete MOS Transistor Design Example** 34

Introduction 34

Defining Parameters 36

Processing Parameters 37

Generating Topology 40

Sizing Devices 41

Generating Layout 42

Placing Transistor Units 43

Creating Bars 44

Creating Routing 46

Creating Device Pins 47

Creating Contact Ring 48

Viewing the Generated Layout 49
Verifying the Generated Layout 54

Feedback

1

Introduction

This tutorial shows how the Python API can be used to develop parameterized cells which represent different types of physical layout designs. These parameterized cells are developed using the Python programming language, so that they are referred to as `PyCells` (PyCell™ for Python parameterized cells). After working through the design examples contained in this tutorial, the reader will be ready to explore and use the Santana system to develop their own parameterized cell designs.

The Santana system makes use of the Python programming language, to provide the designer with the ability to describe their design intent at a higher level of abstraction than is usually done. This is done by making use of the various object-oriented features of the Python programming language. Thus, as an introduction to the design examples described in this tutorial, a short description of the Python programming language, as well as the concepts underlying Object Oriented Programming (OOP) are presented. Readers that are already familiar with Object Oriented Programming concepts and Python programming can skip these short introductory sections.

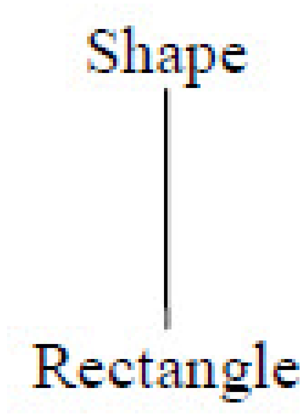
This tutorial consists of three separate sections, organized as follows. The first section contains background material on Object Oriented Programming and the Python programming language. A brief description of parameterized cell development using the Python language is also provided. The second section contains a complete design example, showing how the Python API can be used to quickly develop a parameterized cell design for the basic MOS transistor unit. The third section builds upon this basic MOS transistor unit design, to show how to develop a more complicated parameterized cell design for the MOS transistor. This more complete MOS transistor design also includes bars and routing, as well as an optional contact ring.

Object Oriented Programming

The concepts of Object Oriented Programming (OOP) can be very nicely applied to the problem of developing physical layout designs. The fundamental concepts in layout design can be directly represented as objects in an object-oriented programming language. For example, the layer in an integrated circuit can be represented as an object. As another example, the shapes which can be drawn on a layer in an integrated circuit can also be represented as objects. For example, this would include shapes such as rectangles, polygons, circles, ellipses, lines and donuts.

In order to define an object in an object-oriented programming language, a class definition statement is used. This class definition statement defines the basic properties for an object of this class. For example, if a rectangle were defined using a class definition statement, then properties such as the width and height would be defined. In addition, functions (usually called class methods) would be defined in this class definition, which would allow the user of this class to access these properties. For example, the user of the rectangle class would be able to access the width and height of a rectangle, as well as set the values for these width and height properties. In general, these class methods are used to help us associate functions with their data. As simple as this concept seems, it is one of the underlying concepts behind OOP.

The real power of this class definition concept comes into play when the related concepts of class derivation and inheritance are introduced. For example, let's consider the example of shapes and rectangles which are used in integrated circuit design layout. We can use a class definition statement to define a base class for a shape object. This shape object would have properties such as the layer on which the shape is drawn. The rectangle object would also be a shape object (since it is a shape), but it would also have additional properties, such as width and length. Thus, we can define a rectangle class as being a class which is derived from the base shape class. The power of doing this is that this rectangle object will then automatically inherit all of the properties of the base shape class object. This is the fundamental concept behind class derivation and inheritance. We say that the rectangle class is derived from the base shape class, and that it inherits all of the properties of the base shape class. This can be simply represented using a class inheritance diagram:



This class inheritance diagram indicates that the Rectangle class is derived from the Shape class, and that the Rectangle class inherits all of the properties of the Shape class.

Although this is a simple example, it nonetheless illustrates one of the fundamental ideas of OOP. Class inheritance provides consistency, making it easier to manage complexity.

The real power of OOP is realized when more complicated classes are considered, and when there are several classes which are derived from one or more base classes. It should be noted that OOP has some additional overhead for simple designs, but the use of OOP provides great benefits for more complex designs.

The Python API provides a large number of base classes and derived classes, which are used to define important objects which are used to create parameterized cells for physical design layout. For example, classes for basic layout objects, such as all possible shapes which can be drawn on an integrated circuit layer are provided. In addition, classes for more complicated objects such as contacts, bars, paths, routes and contact rings are provided. These class definitions can then be accessed and used in layout designs through the use of the Python language extensions which are provided by the Santana system.

Python Programming Language

Now that we have very briefly described the fundamental concepts underlying OOP, we will show how the Python programming language can be used to implement these OOP concepts. Python is a popular open-source programming language, which directly provides the constructs used in OOP, and therefore is a powerful object-oriented programming language. In addition, Python is an interpreted (versus compiled) language, so that it is easy to use the Python interpreter to experiment with the Python source code examples which are discussed in this Tutorial.

Let's consider again the simple example of shapes and rectangles which are used in physical design layout. We can use a Python class definition statement to define a class for a shape object as follows:

```
class Shape:
  def __init__(self):
    self.layer = 'metal1'
  def getLayer(self):
    return(self.layer)
  def setLayer(self, layer):
    self.layer = layer
```

Note that a class is defined in Python using the `class` statement, and that the class methods are defined as functions for this class. There is a special method with the name `__init__` (with two leading underscores and two trailing underscores) which is called whenever an object for this class is created. Also note that by convention, the word `self` is used to refer to an instance of the class; `self` is always used as the first parameter for any method, and all class instance variables (such as `self.layer` in our example), are defined using `self`.

It should also be noted that unlike other languages, the Python programming language uses indentation for statement grouping. For example, the C and C++ languages use left and right curly braces to group statements, while the Skill language uses left and right parentheses. Instead of using a special character to group statements, Python simply uses indentation. That is why the `def` statements for each method, as well as the individual statements which make up each method are indented. (Note that any number of spaces can be used for indentation, as long as it is consistently used. The Python code examples used in this Tutorial consistently use two spaces for indentation levels).

The above Python sample source code example can be used with the Python interpreter. This is most easily done by first invoking the Python interpreter by typing `cnpy` at the command-line. After the Python interpreter prompt `>>>` is printed, indicating that the interpreter is ready to accept input, simply type in the above lines of Python source code used to define the Shape class. After this has been done, note that there are built-in functions, which help to explain the Python code which has been entered. For example, type `dir(Shape)` to get a list of the Shape class methods which were defined. Another useful Python interpreter command is the `help()` command; type `help(Shape)` to get a more complete description of these Shape class methods.

Once this Shape class has been defined, we can then create a Shape object, using the following line of Python code:

```
s1 = Shape()
```

This creates a Shape object named `s1`, which is an instance of the Shape class. We can then use the methods which were defined for this Shape class object as follows:

```
s1.getLayer()            # will return 'metal1'
s1.setLayer('metal2')    # set layer to 'metal2'
s1.getLayer()            # will now return 'metal2'
```

Note that in order to use a class method for an object of the class, you simply use the name of the class object, followed by a dot, followed by the method name. Thus, the Python statement `s1.getLayer()` indicates that the `getLayer` method should be called for the Shape class object. Also note that although the word `self` is always used as the first parameter in the definition of a class method, it is not needed, when the class method is actually used (as Python automatically inserts the `self` word, when calling methods).

We can then use this class definition statement for the Shape object to define a Rectangle class, which will be a class which is derived from the base Shape class. This can be done using the following Python code:

```
class Rectangle(Shape):
  def __init__(self, width, height):
    Shape.__init__(self)
    self.width = width
    self.height = height
  def getWidth(self):
    return(self.width)
  def getHeight(self):
    return(self.height)
  def setWidth(self, width):
    self.width = width
  def setHeight(self, height):
    self.height = height
  def Area(self):
    return(self.width * self.height)
```

Note that a derived class is defined in Python by using the name of the base class after the Python `class` statement. Thus, the class statement `class Rectangle(Shape):` indicates that the Rectangle class is being defined, and will be a class which is derived from the base Shape class. Also note that when the special `__init__` method is defined for this Rectangle class, it first calls the special `__init__` method for the base Shape class. Although this Rectangle class is defined as being derived from the base Shape class, it is still necessary to explicitly call the `__init__` method for the base Shape class. Finally, since we can easily calculate the area of a rectangle (versus an arbitrary shape), we have defined the `Area` method to calculate the area of the rectangle object.

Once this Rectangle class has been defined, we can then create a Rectangle object, using the following line of Python code, which will create a rectangle having a width of 5 and a height of 10:

```
r1 = Rectangle(5,10)
```

This creates a Rectangle object named `r1`, which is an instance of the Rectangle class. We can then use the methods defined for this Rectangle class object as follows:

```
r1.getWidth()            # will return 5
r1.getHeight()           # will return 10
r1.Area()                # will return 50
r1.setWidth(10)          # set width to 10
r1.getWidth()            # will now return 10
r1.Area()                # will now return 100
```

Since this Rectangle class was derived from the base Shape class, we can also use any of the methods which were defined for the Shape class, as the Rectangle class inherits all of the properties of the base Shape class. Thus, we could use the following Python code to determine the layer on which this Rectangle object was drawn:

```
r1.getLayer()            # returns 'metal1' for any Shape object
r1.setLayer('metal2')    # now set layer to 'metal2'
r1.getLayer()            # will now return 'metal2'
```

Although this is a very simple example, it illustrates the general principles of class definition, class derivation and class inheritance. In fact, this example illustrates the general approach that is used to develop parameterized cell designs using the system. Using the base classes which are provided by the Python API, the designer derives new classes based upon these base classes to define the parameterized cell. This approach will be illustrated in each of the sections of this tutorial.

It is obviously not possible to fully describe the many features and capabilities of the Python programming language in just a few introductory pages of this Tutorial. Our approach in this short introduction has been to focus on the underlying concepts of

OOP and how these are directly provided by the Python programming language. For more general background concerning the Python programming language, it is strongly suggested that the Python web site (www.python.org) be consulted, which contains a wealth of information. For example, this web site contains tutorials, complete reference documentation, as well as links to a number of other Python-related web sites. There are also a number of excellent books available on the Python programming language. In particular, the book *Learning Python (second edition)* by Mark Lutz is highly recommended as a good starting point for learning how to program in the Python language. A more advanced book on the Python programming language is *Python in a Nutshell* by Alex Martelli. Both of these books are published by O'Reilly Press.

Developing Parameterized Cells Using Python

Now that we have very briefly described the fundamental concepts underlying OOP, and how the Python programming language implements these OOP concepts, we will show how parameterized cell designs can be developed using Python. This will be done by working through a very simple example of a parameterized cell design. Although this example is extremely simple, it serves to illustrate the basic steps required to develop parameterized cell designs using Python. In addition, it provides the basic understanding of the PyCell parameterized cell concept, which will be used in the more realistic design examples covered in subsequent sections of this Tutorial.

This very basic parameterized cell design example simply constructs a rectangle shape on a layer of an integrated circuit. There will be only three parameters for this parameterized cell: the width of the rectangle, the height of the rectangle, and the layer on which this rectangle should be created. When the user of this parameterized cell specifies the values for these parameters, the Python code which will be written for this PyCell will make use of these parameter values to directly create the required rectangle.

As was discussed earlier in the section on Object Oriented Programming, the real power of classes comes into play when class derivation and inheritance are used. In our situation here, the Python API provides a very important base class which should be used to derive any parameterized cell design. This key base class is called `DloGen`, which stands for “Dynamic Layout Object Generator.” This `DloGen` base class provides a large number of class methods (over 50 class methods in total), all of which perform some important operation involved in the design of a parameterized cell. Thus, when we derive our own class from this base `DloGen` class, we automatically inherit a great deal of valuable functionality. Let's call the class for our parameterized cell design `MyRect`, and then this class inheritance can be simply represented using the following class inheritance diagram:

DloGen
MyRect

This class inheritance diagram indicates that our `MyRect` parameterized cell rectangle class is derived from the base `DloGen` class, and inherits all of the important properties of this `DloGen` base class.

Now that we have derived our `MyRect` parameterized cell class from the `DloGen` base class, we need to define the methods for our new class. In the case of a parameterized cell class, there are at least three steps which are involved in developing a parameterized cell:

1. Define the parameters for the cell – define the parameters which can be used to customize the parameterized cell.
2. Process these parameters – read and process the parameter values which are set by the user of the parameterized cell.
3. Generate the actual layout – create the actual layout for the parameterized cell circuit, based upon the parameter values processed in the second step.

Note that each of these steps can be performed by a single class method for our `MyRect` parameterized cell class, as follows:

1. Define the parameters for the cell – `defineParamSpecs()`
2. Process the parameters – `setupParams()`
3. Generate the actual layout – `genLayout()`

These are the three class methods which we will have to develop for our derived rectangle parameterized cell class. Note that the names and parameters used for these class methods are fixed by the `DloGen` base class. These are the names for the class methods which will be called by the Santana system when our parameterized cell design is used. We can then represent this class inheritance and our three class methods using the following Python code structure:

```
class MyRect(DloGen):

  def defineParamSpecs(cls, specs):
    # code to define parameters

  def setupParams(self, params):
    # code to process parameter values

  def genLayout(self):
    # code to generate rectangle layout
```

Note that in the case of these first two methods, the `specs` and `params` parameters will be used by the Santana system to obtain the array of parameter specifications, and then to pass the parameter values entered by the user of this parameterized cell.

For the first step, the definition of the parameters for our parameterized cell, our parameterized cell will have three parameters: the width of the rectangle, the height of the rectangle, and the layer on which the rectangle should be drawn. In order to specify these parameters, we make use of the `ParamSpecArray` class which is provided for this purpose by the Python API; this class is used to define an array of parameter specifications. Thus, we can write the Python code for our first class method as follows:

```
@classmethod
def defineParamSpecs(cls, specs):
  specs('width', 1.0)
  specs('height, 2.0)
  specs('layer', Layer('metal1'))
```

In the case of the rectangle `width` and `height` parameters, this Python code defines the names for these parameters, and also specifies default values of 1.0 and 2.0, respectively. In the case of the `layer` parameter, the default value is specified to be `metal1` by using the `Layer` class from the Python API to explicitly construct this Layer object. If the user does not specify a value for a parameter, then the default value will be used.

Note that the special Python descriptor syntax `@classmethod` on the line preceding the `defineParamSpecs` function definition is required. This special syntax tells Python to verify these parameter specifications, before creating class objects for this rectangle class.

For the second step, the processing of the parameter values entered by the user of our parameterized cell, note that these parameter values are contained in the `params` parameter which is passed to our `setupParams()` class method. This `params` parameter can be thought of as a Python dictionary, where the keys are the parameter names and the values are the parameter values entered by the user. What we want to do in this class method is to save these parameters as class variables, so that they can be used by the `genLayout()` class method which generates the actual rectangle layout. This can be done using the following Python code:

```
def setupParams(self, params):
  self.width = params['width']
  self.height = params['height']
  self.layer = params['layer']
```

For the third and final step, the generation of the rectangle layout for our parameterized cell, we need to construct a rectangle with the user-specified dimensions on the specified layer. Fortunately, the Python API provides a rich set of basic geometric classes which can be used to construct various types of shapes on different layers. In order to construct a rectangle shape on a layer, we can make use of the `Rect` and `Box` classes, as shown in the following Python code:

```
def genLayout(self):
  x = self.width / 2.0
  y = self.height / 2.0
  Rect(self.layer, Box(-x, -y, x, y))
```

The `Box` class object creates a box using the specified x and y coordinate values; this box is centered at the origin and has the same width and length as our desired rectangle.

This box object is then passed to the `Rect` class object, which uses it to specify the bounding box which defines the rectangle which is being created. This code then creates our desired rectangle shape on the specified layer.

Thus, the complete Python code for our parameterized cell class is the following:

```
class MyRect(DloGen):

  @classmethod
  def defineParamSpecs(cls, specs):
    # define parameters and default values
    specs('width', 1.0)
    specs('height, 2.0)
    specs('layer', Layer('metal1'))

  def setupParams(self, params):
    # process parameter values entered by user
    self.width = params['width']
    self.height = params['height']
    self.layer = params['layer']

  def genLayout(self):
    # generate rectangle layout
    x = self.width / 2.0
    y = self.height / 2.0
    Rect(self.layer, Box(-x, -y, x, y))
```

Thus, in about 15 lines of Python code, we have developed a complete parameterized cell design example. Although it is an extremely simple design example, it nonetheless illustrates the overall process and the steps involved to create a parameterized cell using the Santana design environment.

Python Source Code Examples

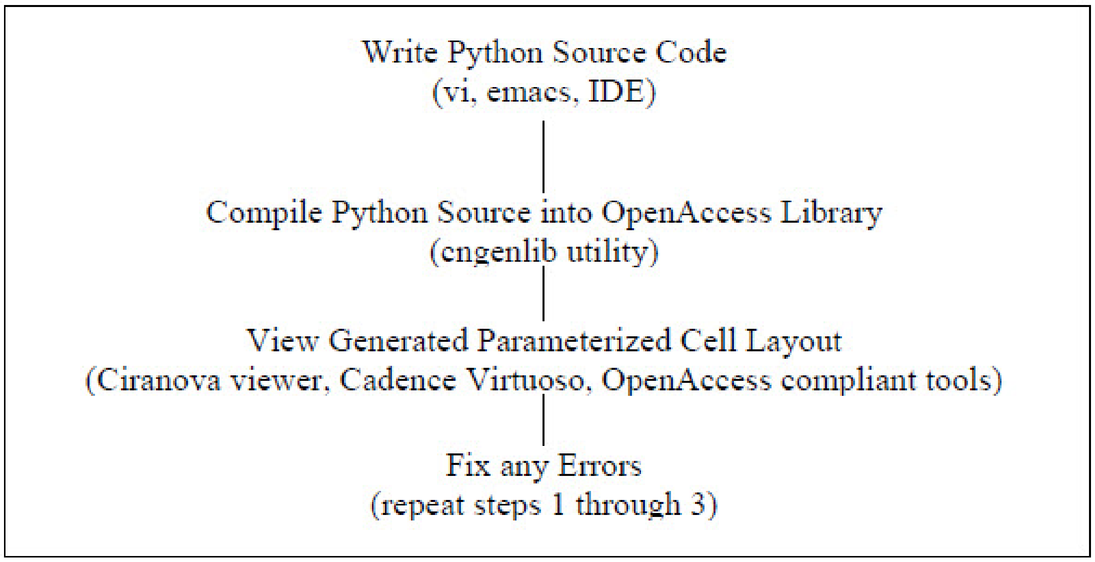
Note that it is not necessary to type in or copy and paste the Python source code examples from this tutorial, in order to use them with the Python interpreter. All of these examples are available in the $CNI_ROOT/tutorial directory. The MyTutorialPyCells subdirectory includes the Python source code for the rectangle parameterized cell example just discussed, as well as the complete Python source code for the transistor unit and MOS transistor design examples which will be discussed in the rest of this tutorial. This MyTutorialPyCells subdirectory should be copied into the reader's home directory; these tutorial instructions assume that all commands will be run from the reader's home directory.

Parameterized Cell Development Process

Now that we have briefly described how the Python programming language can be used to develop parameterized cells, we will examine the overall process for developing a parameterized cell. There are four major steps in the parameterized cell development process:

1. Use the extensions to the Python programming language to develop the Python source code for a parameterized cell. This can be done using vi, emacs, any other text editor or an Integrated Development Environment (such as the WingIDE or the IDLE IDE which is shipped as part of the standard Python distribution).
2. Compile this Python source code into an OpenAccess database library file. This is done using the `cngenlib` utility, which compiles the Python source code into an OpenAccess library. This OpenAccess library file uses the generated Python byte code to evaluate the parameterized cell design whenever necessary.
3. View the generated layout for this parameterized cell, using the viewer tool, or any OpenAccess compliant viewing or editing tool, such as the Cadence Virtuoso layout editor. Since the compiled parameterized cell is saved as an Open Access database library file, it can be read by any number of different OpenAccess compliant viewing or editing tools, which are available from several EDA tool vendors.
4. Based upon the viewing of the generated layout performed in the previous step, make necessary changes to the Python source code to correct any problems. After making these changes, re-compile the Python source code into the OpenAccess library database, and view the generated layout to ensure that it is correct. Note that in addition to viewing the generated layout, it may be required to run a physical verification program (such as the Mentor Graphics Calibre DRC program) to fully verify the correctness of the layout generated for this parameterized cell design.

This overall development process can be represented as follows:


Write Python Source Code
(vi, emacs, IDE)
Compile Python Source into OpenAccess Library
(cngenlib utility)
View Generated Parameterized Cell Layout
(Ciranova viewer, Cadence Virtuoso, OpenAccess compliant tools)
Fix any Errors
(repeat steps 1 through 3)

The rest of this Python API tutorial will show how different parameterized cell design examples can be created using this overall parameterized cell development process. It is important to keep in mind this high-level overview of the development process, to better understand the detailed implementation of these parameterized cell design examples.

2

MOS Transistor Unit Design Example

Introduction

In this section of the Tutorial, we will make use of the introductory material which we have already covered, to develop the Python code for a basic MOS transistor unit parameterized cell. This MOS transistor unit is the basic MOSFET transistor, which consists of a source, drain and gate.

As was the case for the simple example of the `MyRect` rectangle parameterized cell design which was discussed in the previous section Developing Parameterized Cells Using Python, the following are the three major steps involved in developing a parameterized cell design:

1. define the parameters for the cell – define the parameters which can be used to customize the parameterized cell.
2. process these parameters – read and process the parameter values which are set by the user of the parameterized cell.
3. generate the actual layout – create the actual layout for the parameterized cell circuit, based upon the parameter values processed in the second step.

Note that each of these three separate steps can be performed by a single class method for our `MyTransistorUnit` parameterized cell class, which is derived from the `DloGen` base class. This class inheritance and our three class methods can then be represented by using the following Python code structure:

```
class MyTransistorUnit(DloGen):

  def defineParamSpecs(cls, specs):
    # code to define parameters

  def setupParams(self, params):
    # code to process parameter values

  def genLayout(self):
    # code to generate transistor unit layout
```

The rest of this section of the Tutorial will develop the Python code for these three class methods, illustrating how the Python API can be easily used to develop new parameterized cell designs. After this section of the Tutorial has been finished, the complete design for a transistor unit parameterized cell will have been developed.

Defining Parameters

The first step in the design process is to decide which aspects of the transistor unit should be parameterized, so that the designer who uses this parameterized cell can set these parameters to customize the layout which gets generated. There are two different types of MOS transistors, `nmos` and `pmos` types, so this transistor type should be a parameter. The length and width of the transistor unit should also be parameters. The oxide which will be used to construct this transistor unit can be either a thin oxide or a thick oxide, so this oxide type should be another parameter.

In order to make this design even more parameterized, the amount of overlap for the diffusion layer for the source contact and the drain contact for this transistor unit can also be parameter values. In addition, the transistor fill layer can be specified by the user as a possible parameter. Even though these parameters are not required to create the transistor unit, it will make it easier to use this transistor unit in other designs. In general, it is a good idea to provide additional parameters which will make it easier to use the parameterized cell being designed.

Once we have decided on the parameters for this transistor unit, then we need to define these parameters using the `defineParamSpecs()` method for our `MyTransistorUnit` class. In addition to defining these parameters, we should also specify the legal values for these parameters. For example, for the transistor type, there are only two possibilities: `nmos` or `pmos`. Similarly, for the oxide type parameter, there are also only two possibilities: `thin` or `thick`. By contrast, for the width and length parameters, the user can essentially specify any value. However, it would be better to check the technology design rule file, and make sure that the width or length value is at least as large as the minimum value allowed for this technology. This design rule checking can be done through the Python API.

The specification of the parameters and their legal values is handled by means of the `ParamSpecArray` class in the Python API. This class is an array of parameter definitions, along with optional constraints which specify legal values for each parameter. The transistor type parameter can be defined using the following Python code:

```
specs('tranType', 'pmos', 'MOSFET type (nmos or pmos)',
      ChoiceConstraint(['pmos', 'nmos']))
```

This Python code defines a parameter with the name `tranType`, having a default value of `pmos`; the `ChoiceConstraint` indicates there are only two possible values for this parameter: `pmos` or `nmos`. If the user enters any other value for this parameter, then an error message will be generated. The third parameter is an optional documentation string, which provides additional information about the parameterized cell parameter.

In the case of the width and length parameters, there are minimum width and length values which are defined in the technology file. This technology file contains various MOSFET model parameters, including minimum transistor width and length values. These transistor model values depend upon both the transistor type (`nmos` or `pmos`), as well as the oxide type (`thin` or `thick`), as shown in the following Python code:

```
width = specs.tech.getMosfetParams(tranType, oxide, 'minWidth')

length = specs.tech.getMosfetParams(tranType, oxide, 'minLength')
```

We can also define an allowed range of values for these width and length parameters. This can either be done by using the maximum width and length values stored in the technology file, or by simply specifying a maximum allowed value. For this tutorial, we will simply define the maximum value to be 10 times the minimum value. This range of allowed values can then be specified as follows:

```
RangeConstraint(width, 10 * width, USE_DEFAULT)
```

The `USE_DEFAULT` parameter indicates that if the value entered by the user falls outside this allowed range, then the default value will be used instead.

In the case of the parameters for the amount of overlap for the source and drain diffusion, the user can specify any floating-point value. There are no required minimum or maximum values; so that by default, these diffusion overlap values can be set to zero. The parameter for the transistor fill layer can be any valid layer; this parameter can be defined to have a default value of `Layer('metal1')`.

Thus, the Python code for the `defineParamSpecs()` method which defines all of the parameters for our parameterized cell, can be written as follows:

```
@classmethod
def defineParamSpecs(cls, specs):

  # first use variables to set default values for all parameters
  tranType = 'pmos'
  oxide = 'thin'
  width = specs.tech.getMosfetParams(tranType, oxide, 'minWidth')
  length = specs.tech.getMosfetParams(tranType,oxide,'minLength')

  # now use these default values in the parameter definitions
  specs('tranType', tranType, 'MOSFET type (pmos or nmos)',
         ChoiceConstraint(['pmos', 'nmos']))
  specs('oxide', oxide, 'Oxide type (thin or thick)',
         ChoiceConstraint(['thin', 'thick']))
  specs('width', width, 'device width',
         RangeConstraint(width, 10*width, USE_DEFAULT))
  specs('length, length, 'device length',
         RangeConstraint(length, 10*length, USE_DEFAULT))

  specs('sourceDiffOverlap', 0.0)
  specs('drainDiffOverlap', 0.0)
  specs('xtorFillLayer', Layer('metal1'))
```

Note that the special Python `@classmethod` syntax on the line preceding the `defineParamSpec` function definition is required. This special syntax is used to allow these parameter specifications to be verified before creating any class objects. This approach also makes it easier to derive other classes from this transistor unit class.

Processing Parameters

Now we need to define the `setupParams()` method, which corresponds to the second step in our process, which is to process the parameter values which are entered by the user of this parameterized cell design. In this case, the minimum width and length values will need to be adjusted, since these minimum values depend upon both the transistor type and the oxide type entered by the parameterized cell user. For example, the minimum width and length values for a `thick` oxide will be larger than for a `thin` oxide.

Note that the set of parameters which are specified by the user is passed to the `setupParams()` method as the `params` parameter; this parameter is a `ParamArray` object, and consists of an array of parameter settings. We can save all of the parameter values supplied by the user of the parameterized cell into local class variables, as follows:

```
def setupParams(self, params):

  self.width = params['width']
  self.length = params['length']
  self.oxide = params['oxide']
  self.tranType = params['tranType']
  self.sourceDiffOverlap = params['sourceDiffOverlap']
  self.drainDiffOverlap = params['drainDiffOverlap']
  self.xtorFillLayer = params['xtorFillLayer']
```

This is a good programming practice, as it allows other class methods to directly access these variables, without needing to pass them as additional parameters, when these class methods are called. We should also recalculate the required minimum width and length values, using the transistor type and oxide type which were entered by the user:

```
self.width = max(self.width,
                 self.tech.getMosfetParams(self.tranType,
                 self.oxide, 'minWidth'))

self.length = max(self.length,
                  self.tech.getMosfetParams(self.tranType,
                  self.oxide, 'minLength'))
```

In addition to re-adjusting these width and length values, we should also make sure that these values lie on manufacturing grid points; otherwise, DRC errors may be generated.

This can be done by `snapping` to the nearest manufacturing grid point, using the `Grid` class provided by the Python API:

```
grid = Grid(self.tech.getGridResolution())
self.width = grid.snap(self.width, SnapType.ROUND)
self.length = grid.snap(self.length, SnapType.ROUND)
```

We should also specify the layer which will be used to create the gate for the transistor unit, as well as the layers which should be used for diffusion and metal. It is a good programming practice to save these specifications as local class variables, so that we do not need to re-construct these Layer objects:

```
self.gateLayer = Layer('poly1')
self.diffLayer = Layer('diff')
self.metalLayer = Layer('metal1')
```

Note that the `Layer` class provided by the Python API can be used to construct these layers. When each of these layer objects is created, the name string being passed is used to find the appropriate layer definition from the technology file.

In addition, we need to determine which layers will be used as enclosure layers when enclosure rectangles are generated for the parameterized cell layout. These enclosure rectangles are used to enclose the transistor unit on the different enclosure layers to isolate the transistor unit. This set of enclosure layers is also based upon the transistor type and the oxide type parameter values which are entered by the user. In the case of the `nmos` transistor, there is no well layer used, since the common substrate can be used for all transistors; however, in the case of the `pmos` transistor, there can be a single n-well used for each transistor. In terms of the oxide type, the `thick` oxide will use the oxide layer for enclosure, but not for the `thin` oxide case. This can be expressed as follows:

```
if self.tranType == 'nmos':
  self.encLayers = [Layer('nimp')]
else:
  self.encLayers = [Layer('pimp'), Layer('nwell')]

if self.oxide == 'thick':
  self.encLayers.append(Layer('od2'))
```

Thus, the self.encLayers class variable will contain a list of the layers which should be used to generate enclosure rectangles.

It is also a good programming practice to define any class variables which are related to process technology in this step of the process of creating a parameterized cell design. For example, in this transistor unit design, we should consider the minimum extension of the poly gate layer beyond the active gate area. For most process technologies, the gate polysilicon must have a minimum extension beyond the active gate area to ensure proper device operation. This minimum extension value can be determined from the technology file, using the following Python code:

```
self.endcap = self.tech.getPhysicalRule('minExtension',
                           self.gateLayer, self.diffLayer)
```

The Python code for this `setupParams()` method which processes all of the parameter values for our parameterized cell can then be written as follows:

```
def setupParams(self, params):

  # save parameter values using class variables
  self.width = params['width']
  self.length = params['length']
  self.oxide = params['oxide']
  self.tranType = params['tranType']
  self.sourceDiffOverlap = params['sourceDiffOverlap']
  self.drainDiffOverlap = params['drainDiffOverlap']
  self.xtorFillLayer = params['xtorFillLayer']

# readjust width and length, as minimum values may be different
  self.width = max(self.width,
                 self.tech.getMosfetParams(self.tranType,
                 self.oxide, 'minWidth'))
  self.length = max(self.length,
                  self.tech.getMosfetParams(self.tranType,
                  self.oxide, 'minLength'))

# also snap width and length values to nearest grid points
  grid = Grid(self.tech.getGridResolution())
  self.width = grid.snap(self.width, SnapType.ROUND)
  self.length = grid.snap(self.length, SnapType.ROUND)

# save layer values using class variables
  self.gateLayer = Layer('poly1')
  self.diffLayer = Layer('diff')
  self.metalLayer = Layer('metal1')

# define layers which should be used for enclosure rectangles
  if self.tranType == 'nmos':
    self.encLayers = [Layer('nimp')]
  else:
    self.encLayers = [Layer('pimp'), Layer('nwell')]
  if self.oxide == 'thick':
    self.encLayers.append(Layer('od2'))
# determine minimum extension for gate poly layer
  self.endcap = self.tech.getPhysicalRule('minExtension',
                           self.gateLayer, self.diffLayer)
```

Generating Layout

Now we actually generate the layout for this transistor unit, by the use of the `genLayout()` class method. This is a method which uses all of the information that has been gathered so far to draw the desired rectangles and shapes on the different layers to generate the layout for the parameterized cell design. In the case of the MOS transistor unit, we need to generate the source, drain and gate. In the case of the source and drain, it is necessary to generate contacts, which will connect the diffusion and metal layers. In the case of the gate, it is necessary to generate a gate rectangle, which will be constructed on the polysilicon layer.

In addition to constructing these shapes for the source, drain and gate on the different layers, this method should also generate the terminals and pins for this transistor unit device. These terminals will be generated for the source, gate, drain and bulk (or substrate), while pins will be generated for the source, gate and drain. In addition, it is necessary to generate the enclosure rectangles which will be used to enclose this transistor unit on the different enclosure layers. Note that these enclosure layers were determined in the development of the `setupParams()` method.

The first step in our layout generation is to generate the poly rectangle for the transistor unit gate. This is done by means of the `Box` and `Rect` classes, provided by the Python API. This `Box` class is used to specify the dimensions of the rectangle which will be used for the transistor unit gate. Note that in addition to the transistor unit width and length values, we also need to consider the minimum extension for the polysilicon beyond the active gate area. These width, length and extension values can be used to construct this gate rectangle, using the following Python code:

```
gateBox = Box(-self.endcap, 0, (self.width + self.endcap),
               self.length)
gateRect = Rect(self.gateLayer, gateBox)
```

This will construct a rectangle on the gate layer, with the dimensions specified by the transistor unit width and length, as well as the polysilicon minimum extension value.

In order to create source and drain contacts for this transistor unit, the `DeviceContact` class provided by the Python API can be used. This DeviceContact class provides methods to not only create device contacts, but also to resize them as needed. In order to create a DeviceContact class object, it is necessary to specify the two layers which are being connected, the box which specifies the device contact size, along with a name for the DeviceContact instance which is created. Thus, these source and drain contacts can be constructed as follows:

```
self.sourceContact = DeviceContact(self.diffLayer,
                                   self.metalLayer,
                                   gateBox, name = 'S')

self.drainContact = DeviceContact(self.diffLayer,
                                  self.metalLayer,
                                  gateBox, name = 'D')
```

We should stretch these two contacts, so that they will be as wide as the width of our transistor unit. This can be done with the following Python code:

```
sourceBox = self.sourceContact.getRect1().getBBox()
self.sourceContact.stretch(self.metalLayer, sourceBox)

drainBox = self.drainContact.getRect1().getBBox()
self.drainContact.stretch(self.metalLayer, drainBox)
```

Now that the gate rectangle and the source and drain contacts have been created, it is necessary to properly place them into the layout for our parameterized cell. What we would like to do is to have the gate rectangle placed between the source contact and the drain contact, and placed as close as possible, without violating any DRC design rules. This can be done using the Python API in different ways. One approach would be to read the appropriate DRC design rules from the technology file, and then explicitly place each of these shapes on the appropriate layers. Another approach is to use the built-in `smart place` functions, which themselves read the appropriate design rules from the technology file, and then use this information for optimal placement of these design objects. For example, we can very easily use the `fgPlace()` method to `smart place` the gate rectangle between the source and drain contacts with the following Python code:

```
fgPlace(self.sourceContact, SOUTH, gateRect)

fgPlace(self.drainContact, NORTH, gateRect)
```

These `fgPlace()` statements will place the source contact SOUTH of the gate rectangle, and the drain contact NORTH of the gate rectangle, using minimum distance placement, as determined by the appropriate DRC rules in the technology file.

It is also necessary to create the diffusion for the gate. This will be constructed as a rectangle on the diffusion layer, which will extend from the top of the source contact to the bottom of the drain contact. This diffusion rectangle can be constructed by getting the bounding box for the source and drain contacts, as shown in the following Python code:

```
top = self.drainContact.getBBox().bottom
bottom = self.sourceContact.getBBox().top
diffBox = Box(0, bottom, self.width, top)
diffRect = Rect(self.diffLayer, diffBox)
```

In addition, we need to add extra diffusion overlap outside these source and drain contacts, if either of these parameters have been specified by the user of our parameterized cell. This can be done by changing the size of the bounding boxes for these source and drain contacts. Additional diffusion is added to the bottom of the source contact, while additional diffusion is added to the top of the drain contact. This can be constructed as shown in the following Python code:

```
if self.sourceDiffOverlap > 0:
  sBox = self.sourceContact.getBBox(self.diffLayer)
  sBox.setBottom(sBox.bottom - self.sourceDiffOverlap)
  Rect(self.diffLayer, sBox)

if self.drainDiffOverlap > 0:
  dBox = self.drainContact.getBBox(self.diffLayer)
  dBox.setTop(dBox.top + self.drainDiffOverlap)
  Rect(self.diffLayer, dBox)
```

As a final step in our construction of the actual layout for our parameterized cell transistor unit design, enclosure rectangles can be generated for each of the enclosure layers which were determined in the `setupParams()` method. Fortunately, there is already a method in the Python API which will do this automatically. This `fgEnclose()` method can be used as follows:

```
fgEnclose(self.makeGrouping(), self.encLayers)
```

Note that the first parameter in this method call, self.makeGrouping(), is used to obtain all of the physical components in our transistor unit; this will consist of the gate rectangle and the source and drain contacts. These physical components will then be used when the enclosing rectangles are constructed on each of the enclosure layers.

Now that we have generated the basic layout for our transistor unit parameterized cell, it is also necessary to create terminals and pins for this design. In the case of the terminals, we should construct a terminal for the source, drain and gate. In addition, a terminal for the bulk (or substrate) should also be constructed. These terminals and pins are created as part of generating the transistor unit layout, in order to make it easier to use this parameterized cell in other designs. In addition, these terminals and pins can be used to integrate this transistor unit design with other layout tools. This can be done by means of the `addTerm()` method, as follows:

```
self.addTerm('S', TermType.INPUT_OUTPUT)   # source terminal
self.addTerm('D', TermType.INPUT_OUTPUT)   # drain terminal
self.addTerm('G', TermType.INPUT)          # gate terminal
self.addTerm('B', TermType.INPUT_OUTPUT)   # bulk terminal
self.setTermOrder(['D', 'G', 'S', 'B'])
```

In terms of adding pins, we need to add pins for the gate rectangle, as well as for the source and drain contacts. This can be done using the `addPin()` methods, as follows:

```
self.addPin('G', 'G', gateBox, self.gateLayer)

self.addPin('S', 'S',
    self.sourceContact.getBBox(self.metalLayer), self.metalLayer)
self.addPin('D', 'D',
    self.drainContact.getBBox(self.metalLayer), self.metalLayer)
```

Notice that in the case of constructing pins (versus terminals), it is also necessary to specify the bounding box for the Pin object, as well as the desired layer.

Thus, the Python code for this `genLayout()` method which will generate all of the actual layout for our parameterized cell can be written as follows:

```
def genLayout(self):

  # first create the rectangle for the gate
  gateBox = Box(-self.endcap, 0, (self.width + self.endcap),
                self.length)
  gateRect = Rect(self.gateLayer, gateBox)

  # now create contacts for the source and drain
  self.sourceContact = DeviceContact(self.diffLayer,
                                     self.metalLayer,
                                     gateBox, name = 'S')
  self.drainContact = DeviceContact(self.diffLayer,
                                    self.metalLayer,
                                    gateBox, name = 'D')

  # stretch source and drain contacts to full transistor extent
  sourceBox = self.sourceContact.getRect1().getBBox()
  self.sourceContact.stretch(self.metalLayer, sourceBox)
  drainBox = self.drainContact.getRect1().getBBox()
  self.drainContact.stretch(self.metalLayer, drainBox)

  # use smart place to place gate between source and drain
  fgPlace(self.sourceContact, SOUTH, gateRect)
  fgPlace(self.drainContact, NORTH, gateRect)

  # create the gate diffusion rectangle,
  # from top of source to bottom of drain
  bottom = self.sourceContact.getBBox().top
  top = self.drainContact.getBBox().bottom
  diffBox = Box(gateBox.left, top, gateBox.right, bottom)
  diffRect = Rect(self.diffLayer, diffBox)

  # add any extra diffusion outside source and drain contacts
  if self.sourceDiffOverlap > 0:
    sBox = self.sourceContact.getBBox(self.diffLayer)
    sBox.setBottom(sBox.bottom - self.sourceDiffOverlap)
    Rect(self.diffLayer, sBox)
  if self.drainDiffOverlap > 0:
    dBox = self.drainContact.getBBox(self.diffLayer)
    dBox.setTop(dBox.top + self.drainDiffOverlap)
    Rect(self.diffLayer, dBox)

  # define transistor enclosure rectangles on enclosure layers
  fgEnclose(self.makeGrouping(), self.encLayers)

  # add terminals for this transistor unit
  self.addTerm('S', TermType.INPUT_OUTPUT)   # source terminal
  self.addTerm('D', TermType.INPUT_OUTPUT)   # drain terminal
  self.addTerm('G', TermType.INPUT)          # gate terminal
  self.addTerm('B', TermType.INPUT_OUTPUT)   # bulk terminal
  self.setTermOrder(['D','G','S','B'])

  # also define pins for this transistor unit
  self.addPin('G', 'G', gateBox, self.gateLayer)
  self.addPin('S', 'S',
    self.sourceContact.getBBox(self.metalLayer), self.metalLayer)
  self.addPin('D', 'D',
    self.drainContact.getBBox(self.metalLayer), self.metalLayer)
```

Notice that due to the high-level classes provided by the Python API, we can generate the layout for this transistor unit in a small number of Python statements, around twenty lines of Python code. In addition, it is very easy for someone else to read this Python code to see how the layout for this transistor unit was generated.

Viewing the Generated Layout

Now that we have developed the complete Python source code for our transistor unit parameterized cell, it is necessary to view the layout which will be generated. This will allow us to verify that this Python source code produces the expected results.

However, before we can use the tools to view the generated layout for our parameterized cell design, we first need to compile the Python source code for our parameterized cell into an OpenAccess library. The reason that this needs to be done is that all of the viewing and debugging tools are designed to work with OpenAccess libraries of parameterized cells. These parameterized cell designs are saved and stored using an OpenAccess database, and this OpenAccess database is stored in a library of parameterized cell designs.

In order to create OpenAccess libraries for parameterized cell designs, Synopsys has developed a stand-alone utility program which will convert one or more Python parameterized cell designs into the OpenAccess library format. This `cngenlib` utility provides an option to display the parameterized cells after they are built, using the viewer tool.

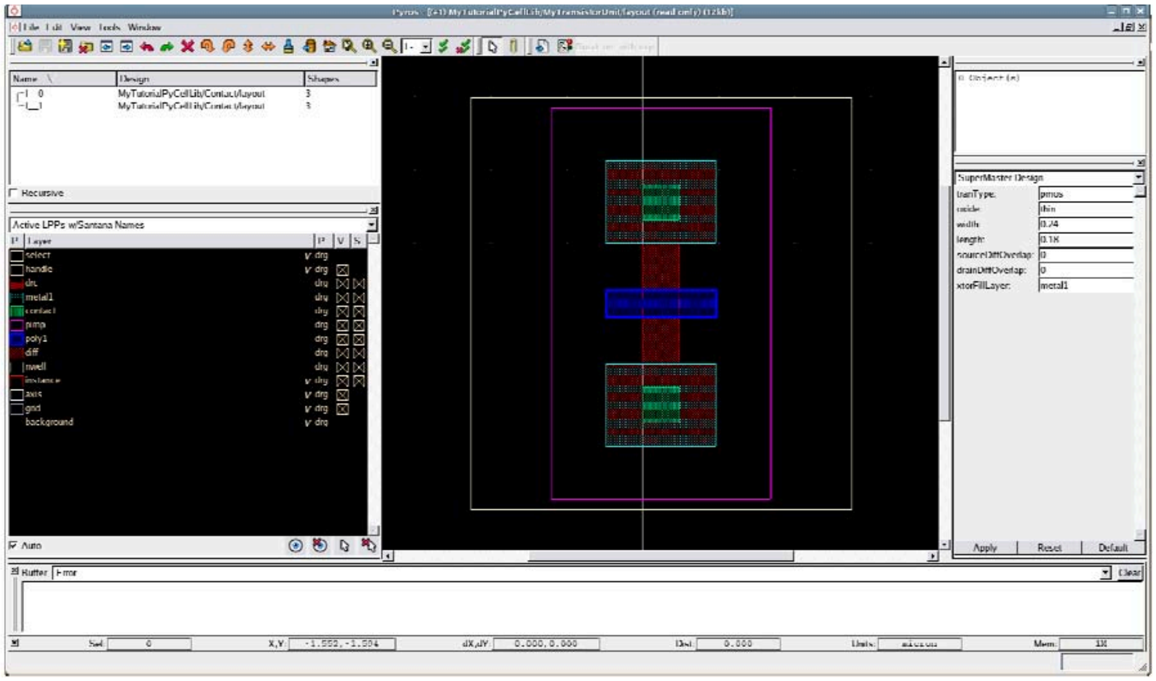
For example, the following command-line invocation of this `cngenlib` utility will create the OpenAccess library for our parameterized cell design:

```
cngenlib --create --view \
  --techfile $CNI_ROOT/tech/cni130/santanaTech/Santana.tech \
  pkg:MyTutorialPyCells MyTutorialPyCellLib \
  ~/MyTutorialPyCellLib
```

The `--create` option means that the OpenAccess library should be created; make sure that the directory ~/MyTutorialPyCellLib does not already exist. The `--view` option means that the viewer tool should be brought up to display the parameterized cells after they are created. The `--techfile` option specifies the location of the technology file, while the `pkg:MyTutorialPyCells` option specifies the Python package name for the Python parameterized cell source code. The last two options specify the OpenAccess library name (`MyTutorialPyCellLib`), located in the user's home directory.

The Python source code for the parameterized cells developed in this tutorial can be found in the $CNI_ROOT/tutorial/MyTutorialPyCells directory; this directory should be copied into the user's home directory, before running this `cngenlib` command.

If this `cngenlib` command is successfully executed, then the viewer should display the transistor unit parameterized cell as shown in the following screen shot:



SuperMaster Design

tranType:	pmos
oxide:	thin
width:	0.24
length:	0.18
sourceDiffOverlap:	0
drainDiffOverlap:	0
xtorFillLayer:	metal1

Note that when this parameterized cell is created, it is created using default parameter values which were specified by the code in the `defineParamSpecs()` class method.

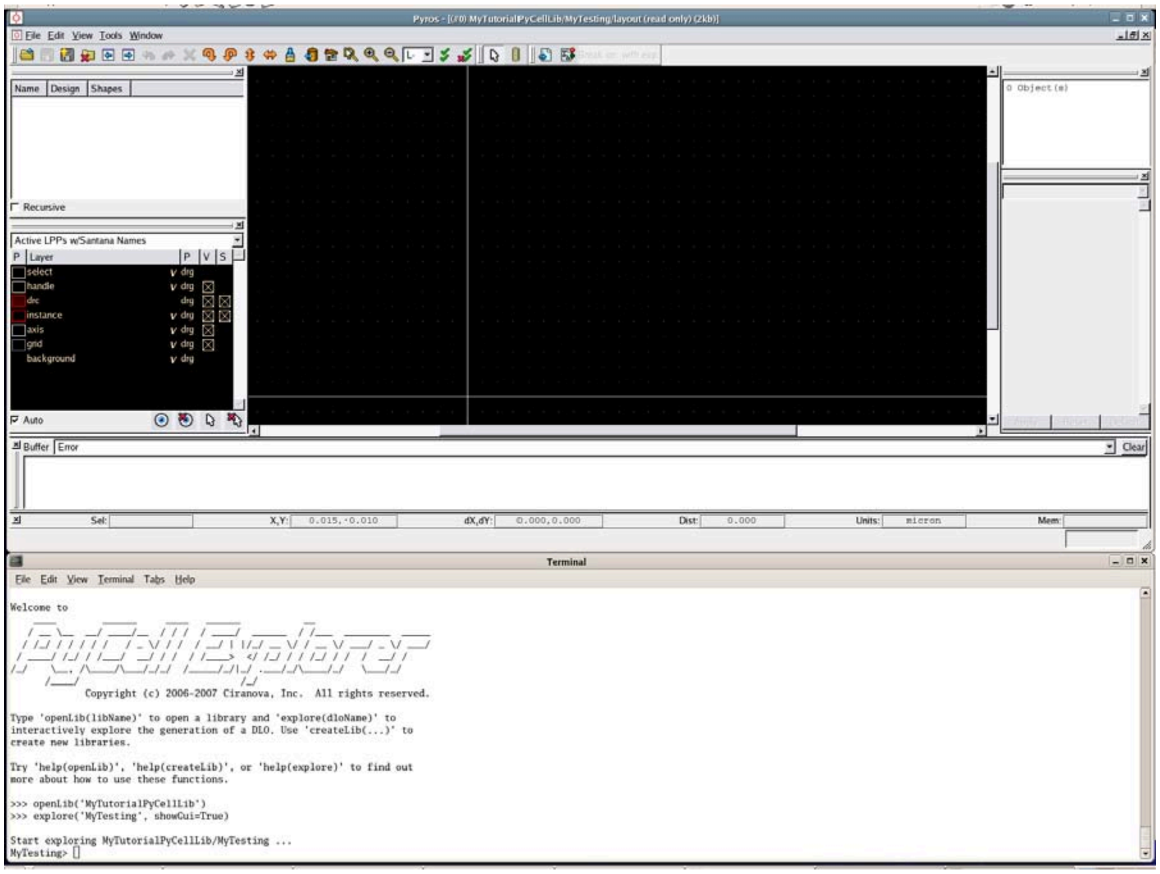
Also note that when this `cngenlib` utility is executed, the Python interpreter will be invoked to compile the Python source code. If there are any errors in this Python source code, then the `cngenlib` utility will fail and generate one or more error messages. These errors must be corrected before the OpenAccess library will be created. The parameterized cell will only be displayed in the viewer tool, if there are no errors.

Note that in addition to this `cngenlib` utility which generates the OpenAccess library for our parameterized cell design, there are also other tools which can be used to view the generated layout. For example, once this OpenAccess library has been created, then the `pyros`™ viewer tool can be invoked directly to view any of the parameterized cells in this library. Another approach is to use the PyCell Explorer tool, which can easily create different instances of our parameterized cell. This PyCell Explorer tool can be used in conjunction with the viewer tool, so that as Python code is executed, the layout which gets generated is displayed in the viewer tool. And finally, the WingIDE tool can be used to debug a Python parameterized cell, and to also interactively step through the Python code to see the layout which gets generated at each step. The use of this WingIDE debugging tool is covered in the section of this tutorial covering the PyCell Studio™ product.

We will now show how the PyCell Explorer tool can be used to create different instances of our parameterized cell design, and how these instances are displayed in the viewer tool. The Python interpreter `cnpy` is used to invoke this PyCell Explorer tool:

```
cnpy

>> import cni.exp
>> openLib('MyTutorialPyCellLib')
>> explore('MyTesting', showGui=True)
```

The `import cni.exp` command imports the Python code for the PyCell Explorer tool, while the `openLib()` command opens up the 'MyTutorialPyCellLib' parameterized cell library which we just created using the `cngenlib` utility. Finally, the `explore()` command opens up a new design for us to use to create instances of our parameterized cell. Note that this command also has an option `showGui` to bring up the viewer tool.



```
Welcome to

            Copyright (c) 2006-2007 Ciranova, Inc.  All rights reserved.

Type 'openLib(libName)' to open a library and 'explore(dloName)' to
interactively explore the generation of a DLO. Use 'createLib(...)' to
create new libraries.

Try 'help(openLib)', 'help(createLib)', or 'help(explore)' to find out
more about how to use these functions.

>>> openLib('MyTutorialPyCellLib')
>>> explore('MyTesting', showGui=True)

Start exploring MyTutorialPyCellLib/MyTesting ...
MyTesting> []
```

At this point, we are now ready to create instances of our parameterized cell design, which can be done through the use of the Python API `Instance` command. This `Instance` command is used to create an instance of a parameterized cell, using the parameter values which are specified by the user. These parameter values are specified using the `ParamArray` command to create an array of parameter values; if no parameter values are specified, then all parameters will be set to their default values. For example, an instance of our transistor unit which uses only default parameter values can be created as follows:

```
p = ParamArray()
tran1 = Instance('MyTransistorUnit', p, None, 'tran1')
```

The first parameter for this Instance command specifies the library name and cell name for the parameterized cell, while the second parameter specifies the array of parameter values which should be used when this parameterized cell is created. The third parameter is used to specify the desired connectivity for this instance; a value of None indicates that no connectivity should be generated. The last parameter is the name which should be used for this instance of our parameterized cell.

Note that using the PyCell Explorer, we can easily write Python code which will create different instances of our transistor unit, each of which has different parameter values. This allows us to quickly test different combinations of parameter values to make sure that our parameterized cell is generating the expected layout for different parameter settings. For example, the following Python code will create four different transistor units, testing all of the different possible settings for the transistor type (`pmos` or `nmos`) and the oxide type (`thin` or `thick`):

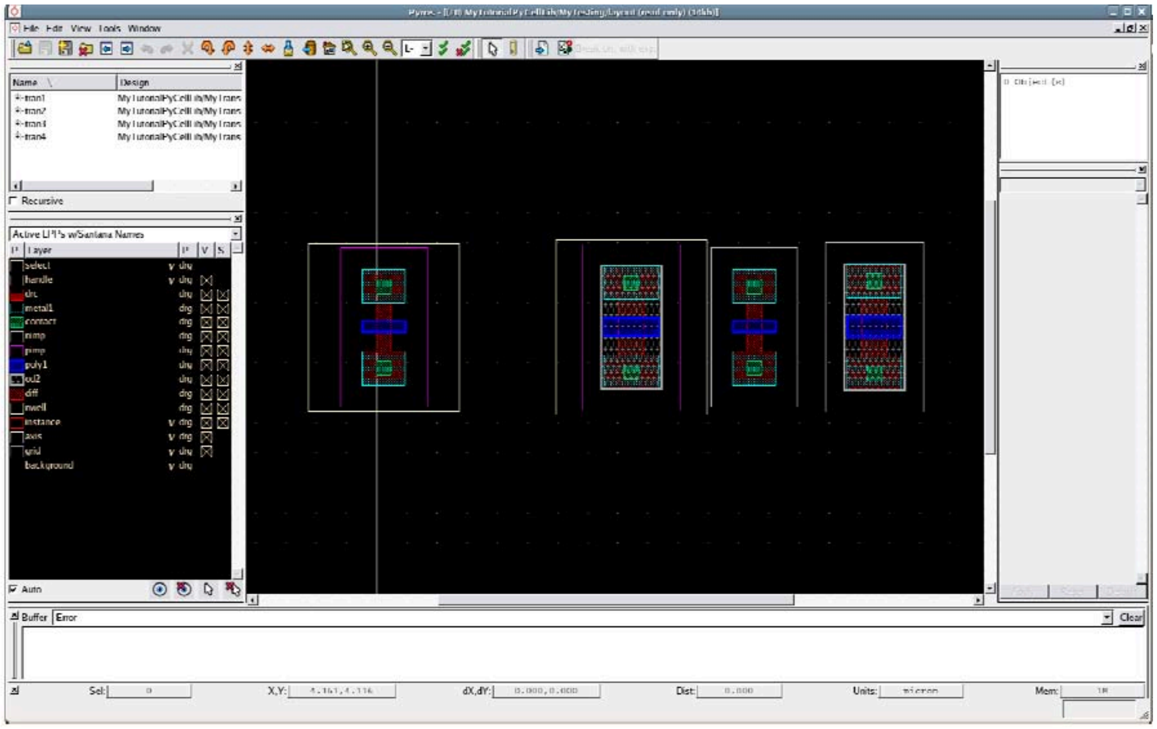
```
p = ParamArray()

p['oxide'] = 'thin'
p['tranType'] = 'pmos'
tran1 = Instance('MyTransistorUnit', p, None, 'tran1')

p['oxide'] = 'thick'
p['tranType'] = 'pmos'
tran2 = Instance('MyTransistorUnit', p, None, 'tran2')

p['oxide'] = 'thin'
p['tranType'] = 'nmos'
tran3 = Instance('MyTransistorUnit', p, None, 'tran3')

p['oxide'] = 'thick'
p['tranType'] = 'nmos'
tran4 = Instance('MyTransistorUnit', p, None, 'tran4')
```

In addition, we can also use the Python API functions to `smart place` these instances in specified relationships to each other. For example, the following Python code will place these transistor units next to each other:

```
fgPlace(tran2, EAST, tran1)
fgPlace(tran3, EAST, tran2)
fgPlace(tran4, EAST, tran3)
```

We can then look at the layout which gets generated for each of these transistor unit instances in the viewer tool, and visually verify that the generated layout is correct. For example, the layers which are used to generate the enclosure rectangles will be different, based upon the transistor type and the oxide type. If the oxide type is `thick`, then the oxide layer will be one of the enclosure layers, while the oxide layer will not be used as an enclosure layer when the `thin` oxide parameter value is selected.

If the above Python source code has been entered and successfully executed, then the viewer tool should look like the following, after using the `Zoom Home` toolbar command:



This graphical display makes it very clear exactly which layers are being used to generate enclosure rectangles for each of these different instances of the transistor unit parameterized cell. Note that the first two instances are spaced apart from one another; the `fgPlace()` method used the spacing rule which is defined for the NWell layer.

Note that using this approach with the PyCell Explorer, it is very easy to write Python code which will test various combinations of parameter settings for the parameterized cell. In addition, the immediate graphical feedback from the viewer tool allows possible layout generation problems to be detected. For example, the Python language has a built-in `random` module which can be used to generate random values for different parameter settings. Thus, it is possible to generate complete test suites for a parameterized cell using this PyCell Explorer tool.

Verifying the Generated Layout

Now that we have viewed the layout which will be generated for our transistor unit parameterized cell, the issue of physical verification should also be briefly discussed. Although we can visually inspect the layout which is generated for different parameter settings, this approach will only allow us to find obvious errors in the generated layout. One solution is to run one of several stand-alone physical verification programs, which are available from various EDA vendors, such as Calibre (Mentor Graphics), Assura (Cadence) or Hercules (Synopsys). Another approach is to make use of the convenient interactive DRC program which is already provided in the viewer tool. This interactive program is intended for use with parameterized cells (versus full-chip layout verification), and the results of running DRC are graphically presented in the viewer tool. Thus, this is a very convenient way to verify the layout which gets generated for a parameterized cell.

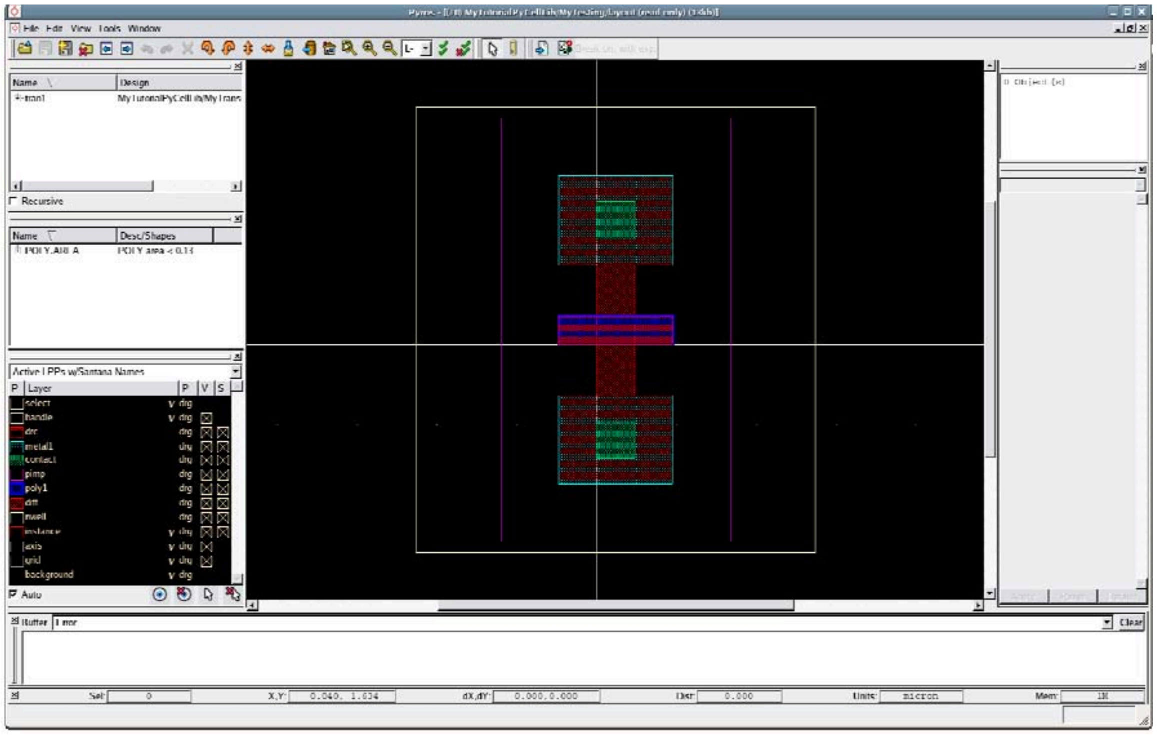
Note that the higher-level objects in the Python API are already `DRC correct by construction`. This is handled by the `Geometry Engine` which underlies these higher-level design objects, as well as all of the `design rule aware` class methods, such as the `fgPlace() smart place` method, which was discussed earlier in this Tutorial. This interactive DRC program makes use of this same `Geometry Engine` technology to provide a convenient means to perform design rule checking. Although these higher level objects will be constructed according to the design rules specified in the technology file for the process which is being used, it is still necessary to perform design rule checks on parts of the layout which are manually generated, or constructed through the Python API, without making use of these higher-level design objects.

In order to run this interactive DRC program, simply click on the double-check toolbar icon in the viewer tool. This will execute the DRC program as a separate thread (for performance reasons), and if any DRC errors are found, then special DRC markers will be generated on the Santana DRC layer. These error markers will be generated next to the geometric shapes which contain the DRC errors, so that it will be easier to find and debug these DRC errors.

In order to see how this DRC programs works with the Transistor Unit which we have created in this tutorial, invoke the PyCell Explorer as before, and then create a single instance of our Transistor Unit, as follows:

```
p = ParamArray()
tran1 = Instance('MyTransistorUnit', p, None, 'tran1')
```

Now simply click on the double-check toolbar icon in the viewer tool. This will execute the DRC program, and display a dialog with the results. In addition, since there are some errors in our parameterized cell, DRC marker geometries will be generated in the main viewing window. If the above Python source code has been entered and successfully executed, then the viewer tool should look like the following:



Note that the DRC window pane shows the DRC rule violation, and the main viewing window highlights the DRC error marker geometry. The problem we have is that there is a minimum area design rule for the polysilicon layer, and the gate rectangle which we created is not quite large enough to meet this minimum area design rule. Recall that when we created this gate rectangle, we did not consider any special minimum area or length rules. However, we can handle this situation by making use of a special method for the Box class in the Python API which was created precisely for this type of situation. This Python code can be changed as follows:

The original code:

# first construct the rectangle for the gate

```
gateBox = Box(-self.endcap, 0, (self.width + self.endcap),
               self.length)
gateRect = Rect(self.gateLayer, gateBox)
```

The updated code:

```
# first construct the rectangle for the gate
gateBox = Box(-self.endcap, 0, (self.width + self.endcap),
               self.length)

if self.tech.physicalRuleExists('minArea', self.gateLayer):
   minArea = self.tech.getPhysicalRule('minArea', self.gateLayer)
   grid = Grid(self.tech.getGridResolution())
   gateBox.expandForMinArea(NORTH, minArea, grid)

gateRect = Rect(self.gateLayer, gateBox)
```

What we have done here is to add four lines of Python source code, which first checks to see if there is a design rule for minimum area for the polysilicon layer, and then uses the special `expandForMinArea()` method for the Box class to expand our gate rectangle in the NORTH direction, so that it will meet this minimum area requirement. Note that this method will both expand the box for minimum area, and also ensure that the resulting box lies on grid point values.

At this point, you can simply edit the Python code in the file ~/MyTutorialPyCells/tutorial2.py, and remove the comments around these four lines of code shown above. Note that these comments were added to comment out these lines of Python code, so that we could easily illustrate the usage of the DRC program with a "real" design rule violation.

After changing these three lines of Python source code, you should be able to re-run the `cngenlib` utility to re-compile the Transistor Unit parameterized cell. Once this is done, then the above example can be re-run. In this case, you should no longer have this minimum area DRC rule violation; in fact, there should not be any DRC rule checking errors generated when you re-run the DRC program on the updated version of this Transistor Unit parameterized cell.

Feedback

3

Complete MOS Transistor Design Example

Introduction

In this section, we will develop the Python code for a basic MOS transistor, by making use of the MOS transistor unit design which was previously developed. This will be done to illustrate the basic concepts of design reuse and show how existing classes can be used to create even more complicated parameterized cells. The MOS transistor which we will develop is the basic MOSFET transistor, in which the user of this transistor parameterized cell will be able to specify the number of fingers which should be used when generating the layout for the transistor, as well as whether bars should be generated.

The reason that we make the number of fingers for a transistor a parameter for this parameterized cell is that it allows the user to obtain more optimal layout results. Recall that when layout is generated for wide transistors, it is advantageous to decompose the transistor into parallel `fingers`, in order to reduce the gate resistance. The use of multiple fingers allows the user of the transistor parameterized cell to generate a physical design layout which will be more optimal for their process technology.

Bars are used to provide routing between different layout objects in a design. The reason that we would like to make the generation of bars an option for this parameterized cell, is that it allows the designer to more easily connect this transistor with other devices in their final physical design layout. Although the designer can add these Bar objects later, it is more convenient to have these bars generated by this parameterized cell.

The final parameter which we will consider for the transistor parameterized cell is the optional generation of a contact ring object around the final transistor layout. This contact ring (or guard ring) can be used to isolate this transistor from noise generated by other circuits in the final design layout, or to possibly prevent latch-up. This is a useful option, depending upon the other devices in the final design layout.

The following are the basic steps which are involved in developing a parameterized cell:

1. Define the parameters for the cell – define the different parameters which can be used to customize the parameterized cell; also specify their types and allowable values.
2. Process the parameters – take the parameter settings which are entered by the user, and validate these parameter values for correctness. Also create a more useful format for using these variables, which will be used by the other methods for this

parameterized cell. This step can be used to simplify the code required in the other steps.

3. Generate the topology – if the parameterized cell consists of multiple devices, define the overall circuit topology (or connectivity) for the different devices which are contained in the parameterized cell circuit.
4. Size the devices – if the parameterized cell consists of multiple devices, define the sizes for the different devices which are contained in the parameterized cell circuit.
5. Generate the actual layout – create the actual layout for the parameterized cell circuit, using the information which has been obtained and processed in the preceding four steps. This is the main part of the parameterized cell development, where the actual geometry is created for the layout of the parameterized cell integrated circuit.

Note that these steps in the development of a parameterized cell directly correspond to the five different class methods which will have to be developed:

1. define the parameters for the cell – `defineParamSpecs()`
2. process the parameters – `setupParams()`
3. generate the topology – `genTopology()`
4. size the devices – `sizeDevices()`
5. generate the actual layout – `genLayout()`

We can then represent this class inheritance and our five class methods using the following Python code structure:

```
class MyTransistor(DloGen):

  def defineParamSpec(cls, specs):
    # code to define parameters

  def setupParams(self, params):
    # code to process parameter values

  def genToplogy(self):
    # code to generate layout topology

  def sizeDevices(self):
    # code to size different devices

  def genLayout(self):
    # code to generate transistor layout
```

The rest of this section of the Tutorial will develop the Python code for these five class methods, illustrating how the Python API can be easily used to develop new parameterized cell designs. After this section of the Tutorial has been finished, the complete design for a MOS transistor parameterized cell will have been developed.

Defining Parameters

The first step in the design process is to decide which aspects of the transistor should be parameterized, so that the designer can set these parameters to customize the generated layout. The first set of parameters which we will use will be the parameters which were used in the design of the transistor unit:

- Transistor type – `nmos` or `pmos`
- Oxide type – `thin` or `thick`
- Width – width of this transistor
- Length – length of this transistor

In addition to these four parameters, we would also like to make the number of `fingers` an option, so that the layout can be optimized for wide transistors. The value of this `fingers` parameter can range from a minimum of 1 to any maximum value. For the purposes of this tutorial, we will arbitrarily set this maximum value to be 100. The Python code which defines this `fingers` parameter can be written as follows:

```
specs('fingers', 1, 'Number of fingers',
          RangeConstraint(1, 100, USE_DEFAULT))
```

We would also like to make the generation of Bar objects for routing purposes be another parameter which can be set by the user of our parameterized cell. In this case, the only allowable values for this `bars` parameter will be `True` or `False`, so that this Boolean parameter can be written using the following Python code:

```
specs('bars', True)
```

This line of Python code specifies that the 'bars' parameter is a Boolean parameter, which has a default value of `True`. Thus, unless the user of this parameterized cell explicitly sets this 'bars' parameter to `False`, bars will automatically be generated as part of the final transistor layout. In a similar fashion, we can use a single line of Python code to define the 'substrateContact' parameter, which is used to indicate whether a contact ring structure should be generated around the final transistor layout, as follows:

```
specs('substrateContact', False)
```

This line of Python code specifies that the 'substrateContact' parameter is a Boolean parameter, which has a default value of `False`. Thus, unless the user of this parameterized cell explicitly sets this `substrateContact` parameter to `True`, no contact ring will be generated as part of the final transistor layout.

The Python code for the `defineParamSpecs()` method which defines all of the parameters for our parameterized cell, with their allowed values, can be written as follows:

```
@classmethod
def defineParamSpecs(cls, specs):

  tranType = 'pmos'
  oxide = 'thin'
  width = specs.tech.getMosfetParams(tranType,
                                     oxide, 'minWidth')
  length = specs.tech.getMosfetParams(tranType,
                                      oxide, 'minLength')
```

# define all parameters for this transistor parameterized cell

```
  specs('width', width, ' device width',
         RangeConstraint(width, 10*width, USE_DEFAULT))
  specs('length, length, 'device length',
         RangeConstraint(length, 10*length, USE_DEFAULT
  specs('tranType', 'pmos', 'MOSFET type (pmos or nmos)',
         ChoiceConstraint(['pmos', 'nmos']))
  specs('oxide', 'thin', 'Oxide (thin or thick)',
         ChoiceConstraint(['thin', 'thick']))

  specs('fingers', 1, 'Number of fingers',
         RangeConstraint(1, 100, USE_DEFAULT))

  specs('bars', True)
  specs('substrateContact', False)
```

Note that the special Python `@classmethod` syntax on the line preceding the `defineParamSpec` function definition is required. This special syntax is used to allow these parameter specifications to be verified before creating any class objects. This approach also makes it easier to derive other classes from this transistor unit class.

Processing Parameters

Now we need to define the `setupParams()` method, which corresponds to the second step in the process of developing a parameterized cell, which processes the parameter values entered by the user. As was the case for the MOS transistor unit parameterized cell example, the width and length values entered by the user may need to be adjusted, since these values depend upon both the transistor type and the oxide type entered by the user.

The set of parameters which are specified by the user is passed to this `setupParams()` method as the `params` parameter, which is a `ParamArray` object, which consists of an array of parameter settings. We should save all of the parameter values specified by the user into local class variables, as follows:

```
def setupParams(self, params):
  self.width = params['width']
  self.length = params['length']
  self.oxide = params['oxide']
  self.tranType = params['tranType']
  self.fingers = params['fingers']
  self.bars = params['bars']
  self.substrateContact = params['substrateContact']
```

We should also recalculate the required minimum width and length values, based upon the transistor type and oxide type entered by the user. In addition, we should also make sure that these width and length values lay on manufacturing grid points, by `snapping` to the nearest manufacturing grid point. This can be done as follows:

```
self.width = max(self.width,
                 self.tech.getMosfetParams(self.tranType,
                           self.oxide, 'minWidth'))

self.length = max(self.length,
                  self.tech.getMosfetParams(self.tranType,
                            self.oxide, 'minLength'))

grid = Grid(self.tech.getGridResolution())
self.width = grid.snap(self.width, SnapType.ROUND)
self.length = grid.snap(self.length, SnapType.ROUND)
```

We should also specify which layer should be used to create the bars, as well as the preferred routing layers which should be used when performing any routing operations.

Note that these layer definitions will only be used, if the user of this parameterized cell sets the parameter to generate bars in the final layout. It is good programming practice to save these specifications in the form of local class variables, as shown in the following Python code fragment:

```
self.barLayer = Layer('metal4')
self.gateRouteLayer = Layer('poly1')
self.drainRouteLayer = Layer('metal3')
self.sourceRouteLayer = Layer('metal3')
self.strapLayer = Layer('metal1')
```

We also want to make sure that the transistor fill layer is the highest layer of any of these routing layers. This is determined by the layer mask number information stored in the technology file. This information can be accessed through the use of the `isAbove()` method on the `Layer` class, as shown in the following Python code fragment:

```
self.xtorFillLayer = Layer('metal1')

if self.gateRouteLayer.isAbove(self.xtorFillLayer):
  self.xtorFillLayer = self.gateRouteLayer

if self.drainRouteLayer.isAbove(self.xtorFillLayer):
  self.xtorFillLayer = self.drainRouteLayer
```

```
if self.sourceRouteLayer.isAbove(self.xtorFillLayer):
  self.xtorFillLayer = self.sourceRouteLayer
```

In addition, we need to determine which well layer will be used when a contact ring is generated to enclose the final transistor layout. Note that this well layer definition will only be used, if the user of our parameterized cell sets the parameter to generate the contact ring. This can be done as shown in the following Python code fragment:

```
if self.tranType == 'pmos':
  self.wellLayer = Layer('nwell')
else:
  self.wellLayer = Layer('pwell')
```

This well layer will then be used when the contact ring gets generated to enclose the final transistor layout.

The Python code for this `setupParams()` method which processes all of the parameter values for our parameterized cell can then be written as follows:

```
def setupParams(self, params):

# save parameter values using class variables
  self.width = params['width']
  self.length = params['length']
  self.oxide = params['oxide']
  self.tranType = params['tranType']
  self.fingers = params['fingers']
  self.bars = params['bars']
  self.substrateContact = params['substrateContact']

  # readjust width and length, as minimum values may be different
  self.width = max(self.width,
                  self.tech.getMosfetParams(self.tranType,
                            self.oxide, 'minWidth'))
  self.length = max(self.length,
                   self.tech.getMosfetParams(self.tranType,
                            self.oxide, 'minLength'))

  # also snap width and length values to nearest grid points
  grid = Grid(self.tech.getGridResolution())
  self.width = grid.snap(self.width, SnapType.ROUND)
  self.length = grid.snap(self.length, SnapType.ROUND)

  # save layer values using class variables
  self.barLayer = Layer('metal4')
  self.gateRouteLayer = Layer('poly1')
  self.drainRouteLayer = Layer('metal3')
  self.sourceRouteLayer = Layer('metal3')

  # ensure that transistor fill layer is highest routing layer
  self.xtorFillLayer = Layer('metal1')
  if self.gateRouteLayer.isAbove(self.xtorFillLayer):
    self.xtorFillLayer = self.gateRouteLayer
  if self.drainRouteLayer.isAbove(self.xtorFillLayer):
    self.xtorFillLayer = self.drainRouteLayer
  if self.sourceRouteLayer.isAbove(self.xtorFillLayer):
    self.xtorFillLayer = self.sourceRouteLayer

  # define the well layer to be used for contact ring generation
  if self.tranType == 'pmos':
    self.wellLayer = Layer('nwell')
  else:
    self.wellLayer = Layer('pwell')
```

Generating Topology

The third step in our process to generate the parameterized cell for our transistor is to generate the topology for the layout which will be generated. Recall that this step was one which was not considered during the design of the transistor unit, since it only consisted of a single device. In this case where we are designing a complete transistor, the user of the parameterized cell can set the `fingers` parameter to create a transistor with multiple fingers. For each of these fingers, we will use a single instance of the transistor unit.

Since each finger of our transistor will consist of a single transistor unit instance, we should use a class variable to store these transistor units. This is necessary, since other methods in our parameterized cell (such as the `sizeDevices()` and `genLayout()` methods) will need to access these transistor units. These transistor units can be saved using a standard Python list. In addition, these transistor units will also be saved as a Grouping object; this will be convenient when bars are created. We should also provide names for these different instances of the transistor unit. The naming scheme which we will use is to number each finger, using either `MP` (for `pmos` type transistors) or `MN` (for `nmos` type transistors) as a base name.

In order to create an instance of our transistor unit, it is necessary to use the Instance class from the Python API. This Instance class will be used to create the instances of our transistor unit parameterized cell which was created during the first phase of this Tutorial. This can be done using the following Python code:

```
defaultParams = ParamArray()

Instance("MyTransistorUnit", defaultParams,
          ['D','G','S','B'], 'MP1')
```

The second parameter is used to specify the parameter values which should be used when this transistor unit is created. In order to simplify this Tutorial, we will just use all default values for the parameters for the transistor unit. This can be done by simply creating a ParamArray object, without specifying any parameter values, as shown above in the first line of Python code. The third parameter in the Instance method call is an ordered list of the names for the terminals to be used when this transistor unit is created.

We just use the standard terminal names for the drain, gate, source, and bulk terminals. Finally, the last parameter is the instance name which should be used to name the instance of the transistor unit parameterized cell which gets created.

Thus, the Python code for our `genTopology()` method can be written as follows:

```
def genTopology(self):

  if self.tranType == 'pmos':
    baseName = 'MP'
  else:
    baseName = 'MN'

  # create grouping for transistor units
  transistorStack = self.getGrouping('transistorStack')

  # create transistor unit for each finger of the transistor
  self.Units = []
  defaultParams = ParamArray()
  for i in range(self.fingers):
    name = baseName + str(i)

  # create transistor unit for this finger, and save it
    self.Units.append(Instance(MyTransistorUnit, defaultParams,
                           ['D','G','S','B'], name))

  # also save this transistor unit in the Grouping object
    transistorStack.add(self.Units[i])

  # also create the terminals for this transistor
  self.addTerm('S', TermType.INPUT_OUTPUT)    # source terminal
  self.addTerm('D', TermType.INPUT_OUTPUT)    # drain terminal
  self.addTerm('G', TermType.INPUT)           # gate terminal
  self.addTerm('B', TErmType.INPUT_OUTPUT)    # bulk terminal
  self.setTermOrder(['D','G','S','B'])
```

Sizing Devices

Now that we have completed the third step in the creation of the parameterized cell for this transistor, we need to consider the fourth step, the sizing of the devices which make up our transistor design. Recall that this step was one which was not considered during the design of the transistor unit, since it only consisted of a single device. In this case where we are designing a complete transistor, the user of the parameterized cell can set the `fingers` parameter to create a transistor with multiple fingers. Recall that fingers are used to layout wide transistors; wider transistors require more fingers to be used. Thus, the total width which is specified and used for our transistor needs to be divided up and used by each of these different fingers. Thus, we need to resize the width value for each of the transistor units which will make up our transistor design. This can be done by simply dividing the total transistor width by the number of fingers, using the following Python statement:

```
unitWidth = self.width/self.fingers
```

Once we have calculated the proper width for each of the transistor units for each finger of our transistor, we then need to apply this value to each transistor unit. This can be done by using the `setParams()` method for each instance of our transistor unit. We simply need to create a new `ParamArray` object which contains this new width value, and then pass this new `ParamArray` object to the `setParams()` method for each of these transistor unit instances.

Thus, the following Python code can be used to size the devices for our transistor parameterized cell:

```
def sizeDevices(self):

  unitWidth = self.width/self.fingers
  unitParams = ParamArray(tranType = self.tranType,
                          width = unitWidth,
                          length = self.length,
                          oxide = self.oxide,
                          xtorFillLayer = self.xtorFillLayer)
  for unit in self.Units:
    unit.setParams(unitParams)
```

Generating Layout

Now that we have completed the fourth step in the creation of the parameterized cell for this transistor, we need to consider the fifth and final step, the actual generation of the layout for this transistor. This layout is generated through the development of the Python code for the `genLayout()` method. This is a method which uses all of the information that has been gathered so far to draw the desired rectangles and shapes on the different layers to generate the actual layout for the parameterized cell design. In the case of the MOS transistor, we need to generate the individual transistor units, one for each finger of the transistor. In addition, we also need to generate the layout for the bars, as well as the contact ring which would enclose the final design layout. The generation of bars and the contact ring would only be done, if the user had selected these options and set the parameters for the parameterized cell to make use of these options. And as a final step, we also need to create the pins for each of the devices in our transistor parameterized cell design.

This `genLayout()` method can be written as one single method, but given the number of steps which would be required to generate this layout for the transistor, this would likely require many individual Python statements. Alternatively, we could structure this method to call other individual Python functions, which would perform each of the individual steps required to generate the layout. In order to illustrate the type of program structure that would best be used when developing other PyCells, we will follow this latter approach of using several different Python functions which get called from this `genLayout()` method. Thus, the Python code for the `genLayout()` method can be structured as follows:

```
def genLayout(self):

  # place the different transistor units, one for each finger
  self.stackUnits()

  # if bars should be generated, create bars and required routing
  if self.bars:
    self.createBars()
    self.createRouting()

  # create pins for each of the devices in this transistor
  self.createPins()

  # create possible contact ring around the final layout
  if self.substrateContact:
    self.createRing()
```

This small amount of Python code can then be used to define the required `genLayout()` method. Of course, this is a small amount of Python code, since it simply calls five other Python functions to perform the actual work. However, this approach very clearly shows the operations which are required to generate the actual layout. It is now just a matter of developing the Python code for these five lower-level functions.

Placing Transistor Units

The first Python function which we need to develop is the one which places the individual transistor units, one for each finger of the transistor. These different fingers will simply be placed one finger above each previous finger, as is suggested by the name which was chosen for this Python function, `stackUnits`. In addition to placing these transistor units, it will also be necessary to `flip` every other transistor unit, so that the sources and drains for each transistor unit are properly located next to each other. This flipping process can be performed by the `mirrorX()` method provided by the Python API. The placement of each transistor unit can be handled by using the explicit `place()` method provided by the Python API.

There is one subtle point which we also need to consider when placing these transistor units. Namely, when we place these transistor units for multiple fingers, then we need to merge the source or drain contacts for each finger. This requires us to determine the distance which should be used when placing these transistor units, so that the sources or drains will be properly overlapped. The easiest way to handle this is to write a separate Python function to calculate this overlap distance, as follows:

```
def getMergeOverlap(self, tranUnitInstance, drainFlag):
  # calculate the distance which transistor unit must be moved
  # to merge the drain or source with another transistor unit.
  if drainFlag:
    pinBox = tranUnitInstance.findInstPin('D').getBBox()
  else:
    pinBox = tranUnitInstance.findInstPin('S').getBBox()

  # compare pin bounding box with transistor unit bounding box;
  # adds minimum extension to get overlap for merge operation.
 tranBox = tranUnitInstance.getBBox()
 ext1 = pinBox.bottom - tranBox.bottom
 ext2 = tranBox.top - pinBox.top
 overlap = pinBox.getHeight() + 2.0 * min(ext1, ext2)
 return(overlap)
```

Thus, the Python code for this `stackUnits()` function can be written as follows:

```
def stackUnits(self):

  # flip every other unit, to properly locate sources and drains
  flip = True
  for unit in self.Units:
    if flip:
      unit.mirrorX()
    flip = not flip

  # now place transistor units, each one above the previous one;
  # adjust placement for contact overlap for multiple fingers.
  reference = self.Units[0]
  offset = - self.getMergeOverlap(reference, True)
  for i in range(1, self.fingers):
    place(self.Units[i], NORTH, reference, offset)
    reference = self.Units[i]
```

Creating Bars

The next two Python functions which need to be developed are the Python functions which handle the generation of bars and their associated routing. As discussed earlier, these functions will only be used if the user of our parameterized cell sets the `bar` parameter to True. Fortunately, the Python API already provides a `Bar` class, which can be used to create the required bar objects; this is much easier than simply creating bar rectangles ourselves. In addition, this `Bar` class also provides a number of class methods, which can be used to perform additional operations with these bar objects.

After these `Bar` objects are created, then it will be necessary to properly place them in our layout. Note that there will be three `Bar` objects created, one for the transistor source, one for the transistor drain and one for the transistor gate. These bars could then be placed at a number of different locations in the generated layout for our transistor. However, it would make sense to locate these bars in a regular fashion; some bars could be located to the left of the transistor, and some could be located to the right of the transistor. For example, the gate bar could be placed to the right of the transistor, while the source and drain bars could be located to the left of the transistor. This bar ordering placement could become a parameter for our parameterized cell. (This could be done by using a string to define the order in which the bars should be placed. For example, the string `G-DS` would indicate that the gate bar would be placed to the right of the transistor, while the drain bar would be placed to the left of the transistor, and the source bar would be placed to the left of the drain bar). However, for the purposes of this Tutorial, we will just make use of a fixed bar placement: the gate bar to the right of the transistor, the drain bar to the left, and the source bar to the left of the drain bar.

We now generate these `Bar` objects, which is done by means of the `Bar` class provided by the Python API. In order to create each of these bars, it is first necessary to calculate the width and length for the bar rectangle. The width of the bar will be defined using the minimum width DRC rule for the bar layer. The length will be the height of the complete set of transistor units, which we already defined with the `transistorStack` grouping. These width and length values will be used to define the two diagonal points of the bounding box for the bar rectangle. Once this bar rectangle has been defined, then the `Bar` object can be easily created.

In order to place these `Bar` objects into the proper relative positions, it is necessary to consider the different constructs which will affect this placement operation. Since we will later generate a straight-line `Route` to connect these source, drain and gate bars to the corresponding source, drain and gate pins, we should plan ahead and make sure that these bars are spaced far enough apart to allow room for the contacts which will be generated. The straight-line `Route` will generate contacts for each bar to connect the `Bar` layer to the routing layer. Thus, in order to properly place these bars, we can generate temporary contacts, which will then be removed after the placement operation has been completed. In addition, we should also make sure that these `Bar` objects lie on manufacturing grid points, so that DRC errors will not be generated.

Note that we also construct a gate contact, in addition to the gate `Bar` object; this is done to make it easier to reuse this transistor parameterized cell in other designs. This gate contact should be placed right next to the `Bar` object which is created for the gate. This can now easily be expressed using the following Python statements:

```
def createBars(self):

  transistorStack = self.getGrouping('transistorStack')
  w = self.tech.getPhysicalRule('minWidth', self.barLayer)
  l = transistorStack.getBBox().getHeight()

  # construct bars for the transistor source, drain and gate
  self.sourceBar = Bar(self.barLayer, NORTH_SOUTH, 'S',
                       Point(0,0), Point(w,l))
  self.drainBar = Bar(self.barLayer, NORTH_SOUTH, 'D',
                       Point(0,0), Point(w,l))
  self.gateBar = Bar(self.gateRouteLayer, NORTH_SOUTH, 'G',
                       Point(0,0), Point(w,l))

  # also create gate contact to enable easier design reuse
  self.gateContact = Contact(self.gateRouteLayer, self.barLayer,
                       'G', NONE, NONE, NONE,
                        Point(0,0), Point(w,l))

  # create temporary contacts to help the smart place method
  self.sourceBar.addContact(self.sourceRouteLayer, Point(w,l))
  self.drainBar.addContact(self.drainRouteLayer, Point(w,l))
  self.gateBar.addContact(self.gateRouteLayer, Point(w,l))

  # smart place the gate bar to right of the transistor,
  # and the source and drain bars to the left side
  fgPlace(self.drainBar, WEST, transistorStack)
  fgPlace(self.sourceBar, WEST, self.drainBar)
  fgPlace(self.gateBar, EAST, transistorStack)

  # delete temporary contacts, since the bars have been placed
  self.sourceBar.clearContacts()
  self.drainBar.clearContacts()
  self.gateBar.clearContacts()

  # also snap bars to nearest grid points
  grid = Grid(self.tech.getGridResolution())
  self.sourceBar.snap(grid, SnapType.ROUND)
  self.drainBar.snap(grid, SnapType.ROUND)
  self.gateBar.snap(grid, SnapType.ROUND)

  # also place the gate contact to the right of gate bar
  place(self.gateContact, EAST, self.gateBar, 0,
        self.gateRouteLayer)
```

It should be noted just how easily we could use the Python API to create and place these three different bars for the transistor. Only one Python statement was needed to create the bars, and only one Python statement was needed to `smart place` them in the proper positions, automatically taking into account the relevant design rules.

Creating Routing

The next Python function which we need to develop is the one which creates the routing between the transistor source, drain and gate and the corresponding bar objects. In this case, we would like to construct a straight-line route between the source, drain and gate pins for each finger of our transistor and the corresponding source, drain and gate bar objects. The Python API provides a `Route` class, which can be used to construct various types of routes between two different pins in a design. Of these different types of routes, the straight-line route additionally provides the ability to connect a pin with a bar object. Fortunately, this capability is exactly what we need to create the routing for these three different bar objects. In fact, this can be accomplished using only a single Python statement to create this pin to bar straight-line connection. This can be expressed using the following Python code for our `createRouting()` function:

```
def createRouting(self):

  # create a straight-line route between the pins and bars
  # for source, drain and gate for each transistor finger

  for i in range(self.fingers):
    Route.StraightLine(self.Units[i].findInstPin('S'),
                       self.sourceBar, self.sourceRouteLayer)
    Route.StraightLine(self.Units[i].findInstPin('D'),
                       self.drainBar, self.drainRouteLayer)
    Route.StraightLine(self.Units[i].findInstPin('G'),
                       self.gateBar, self.gateRouteLayer)
```

It should again be noted how easy it was to use the Python API to construct these straight-line routes for each of the fingers of our transistor. A single Python statement can be used to generate the straight-line route, and this Route class method will automatically use the appropriate design rules to ensure that this route meets minimum width rules.

Creating Device Pins

The next Python function which we need to develop is the one which creates the pins for each of the devices in this transistor. There are really two cases which need to be considered when these device pins are created. If the user of our parameterized cell had set the option to generate bars, then we need to ensure that the bar layer would be the preferred routing layer for these device pins. In the case where bars are not generated, then we should simply generate these device pins from the instance pins which were automatically created when each transistor unit was created.

In the case where bars were generated, then we can create these source, drain, and gate pins, and specify that the bar layer be the preferred pin routing layer. This can be done using an option in the `addPin()` method, as follows:

```
self.addPin('S', 'S', self.sourceBar.getBBox(), self.barLayer)
```

This Python statement will create the device pin for the source, for the source terminal (both being named 'S'), and the fourth parameter will set the bar layer to be the preferred routing layer for this device pin. Note that the third parameter will specify the bounding box for the bar rectangle, so that this bar rectangle will be connected to this device pin.

In the case where bars were not generated, then we simply need to create the required device pins. This can be done by making use of the instance pins which were automatically created when we created these individual transistor units. These device pins can also be created by using the `addPin()` method, but we need to pass the instance pin for each transistor unit as a parameter, as shown in the following Python code statement:

```
self.addPin('S', 'S', self.Units[0].findInstPin('S').getBBox())
```

Note that this `findInstPin()` method is defined for the Instance class, and is used to obtain the instance pin from the individual transistor unit.

Thus, we can put together these two different cases into the following Python function. Notice that we simply use a Python for-loop to obtain each of the individual transistor units which were generated. In this case, we use the for-loop index to create a unique name for each of these individual device pins.

```
def createPins(self):

  # create the pins for each of the devices in this transistor
  if self.bars:
    # if bars were created, create pins from bars
    self.addPin('S','S', self.sourceBar.getBBox(), self.barLayer)
    self.addPin('D','D', self.drainBar.getBBox(), self.barLayer)
    self.addPin('G','G', self.gateBar.getBBox(), self.barLayer)
  else:
    # otherwise, create pins from transistor units
    for i in range(self.fingers):
      self.addPin('S' + str(i), 'S',
                  self.Units[i].findInstPin('S').getBBox(),
                  self.xtorFillLayer)
      self.addPin('D' + str(i), 'D',
                  self.Units[i].findInstPin('D').getBBox(),
                  self.xtorFillLayer)
      self.addPin('G' + str(i), 'G',
                  self.Units[i].findInstPin('G').getBBox(),
                  self.xtorFillLayer)
```

Creating Contact Ring

The final Python function which we need to develop is the Python function which generates a contact ring to isolate the final layout generated for this transistor. As discussed earlier, this function will only be used if the user of our parameterized cell sets the `substrateContact` parameter to True. Fortunately, the Python API already provides a `ContactRing` class, which can be used to create the required contact ring. Note that this is much easier than attempting to create the required rectangles and contacts. In addition, this `ContactRing` class will construct the contact ring using the applicable minimum spacing design rules for each layer, so that the contact ring will not have any DRC errors. In addition, this `ContactRing` class also provides a number of class methods, which can be used to perform additional operations with this contact ring.

Note that when this contact ring is created, it will automatically be placed to enclose the layout which was generated for our transistor design. In fact, this contact ring will be placed as close as possible to the existing layout, without violating applicable minimum spacing design rules.

In order to create a contact ring, we need to specify the two layers which should be used to create the four contacts which make up the contact ring object. In addition, any additional layers which should be used in the construction of these contacts should also be specified. In our case, we will specify the proper implant layer which should be used to complete these contacts. As an additional option in the construction of this contact ring object, we will specify the proper well layer to be used as a fill layer, on which an enclosure rectangle will be generated to enclose this contact ring.

Thus, the Python code for this `createRing()` function to create this contact ring object can be expressed as follows:

```
def createRing(self):

  # determine implant layer to be used for constructing contacts
  if self.tranType == 'pmos':
    impLayer = Layer('nimp')
  else:
    impLayer = Layer('pimp')

  # now create the contact ring structure
  ContactRing(Layer('diff'), self.strapLayer, 'B',
              [impLayer], 0, 0, [self.wellLayer])
```

Note that the creation and placement of this contact ring to isolate the layout generated for our transistor design basically required only a single Python statement. This Python API `ContactRing` creation method not only created this contact ring using the applicable design rules for the current technology, but it also placed this contact ring so that it would enclose the generated layout without violating any design rules.

Viewing the Generated Layout

Now that we have developed the complete Python source code for our MOS transistor parameterized cell, it is necessary to view the layout which will be generated. This will allow us to verify that this Python source code produces the expected results.

As was discussed earlier for the case of the transistor unit design, there is a two-step process involved in viewing this generated layout. The first step is to compile the Python source code for our parameterized cell design into an OpenAccess library. The second step is to use either the PyCell Explorer or WingIDE tool to view the layout which gets generated for different parameter settings.

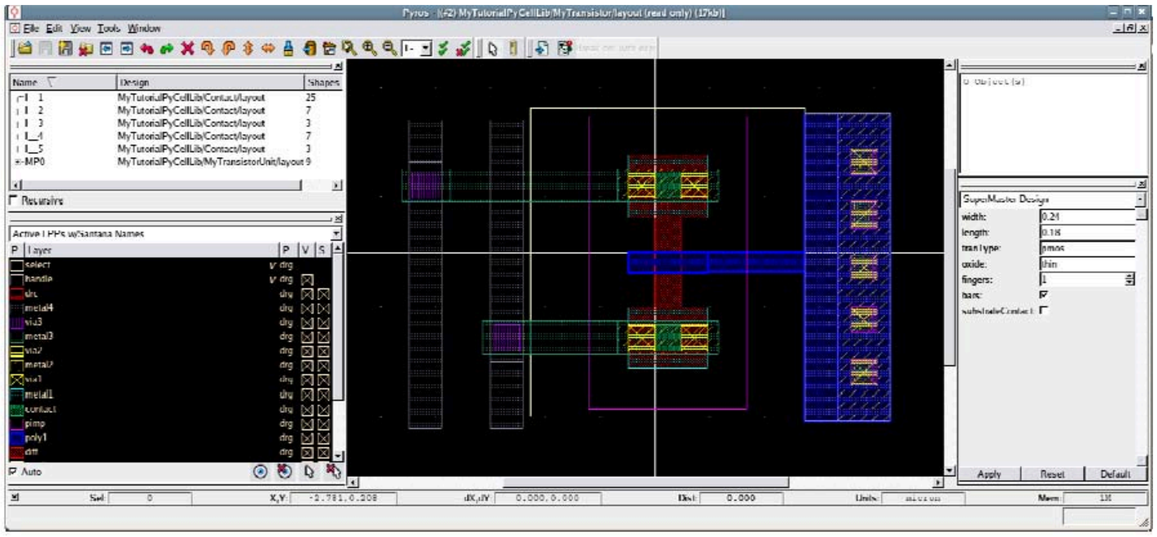
In order to compile the Python source code for our parameterized cell into an OpenAccess library, the `cngenlib` stand-alone utility program can be used. This command-line utility provides an option to display the parameterized cells after they are built, using the viewer tool.

For example, the following command-line invocation of this `cngenlib` utility will create the OpenAccess library for our parameterized cell design:

```
cngenlib --create --view \
  --techfile $CNI_ROOT/tech/cni130/santanaTech/Santana.tech \
  pkg:MyTutorialPyCells MyTutorialPyCellLib \
  ~/MyTutorialPyCellLib
```

The Python source code for the parameterized cells developed in this tutorial can be found in the $CNI_ROOT/tutorial/MyTutorialPyCells directory; this directory should be copied into the user's home directory, before running this `cngenlib` command. Also make sure that the directory ~/MyTutorialPyCellLib does not already exist.

If this `cngenlib` command is successfully executed, then the viewer should display the MOS transistor parameterized cell as shown in the following screen shot:


Pyros - [(2) MyTutorialPyCellLib/MyTransistor/layout (read only) (17kb)]
File Edit View Tools Window
Recursive
Active LPPs w/Santana Names
Auto
0 Object(s)
SuperMaster Design
Apply
Reset
Default

Name	Design	Shapes
1	MyTutorialPyCellLib/Contact/layout	25
2	MyTutorialPyCellLib/Contact/layout	7
3	MyTutorialPyCellLib/Contact/layout	3
4	MyTutorialPyCellLib/Contact/layout	7
5	MyTutorialPyCellLib/Contact/layout	3
MP0	MyTutorialPyCellLib/MyTransistorUnit/layout	9

width:	0.24
length:	0.18
tranType:	pmos
oxide:	thin
fingers:	1
bars:	✓
substrateContact:	☐

Sel: 0 X,Y: -2.781,0.208 dX,dY: 0.000,0.000 Dist: 0.000 Units: micron Mem: 18

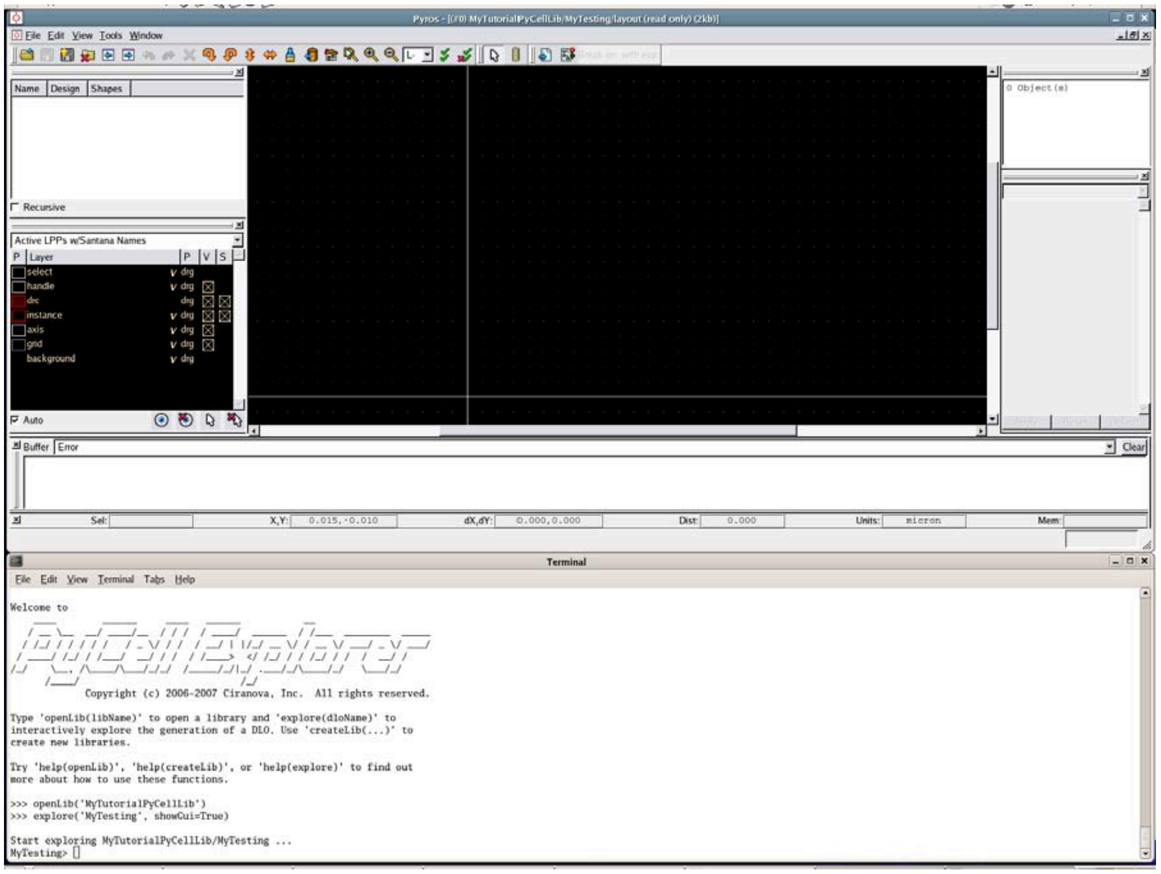
Note that when this parameterized cell is created, it is created using default parameter values which were specified by the code in the `defineParamSpecs()` class method.

Also note that when this `cngenlib` utility is executed, the Python interpreter will be invoked to compile the Python source code. If there are any errors in this Python source code, then the `cngenlib` utility will fail and generate one or more error messages. These errors must be corrected before the OpenAccess library will be created. The parameterized cell will only be displayed in the viewer tool, if there are no errors.

Note that in addition to this `cngenlib` utility which generates the OpenAccess library for our parameterized cell design, there are also other tools which can be used to view the generated layout. The `pyros` viewer tool can be invoked directly to view any of the parameterized cells in this library. Another approach is to use the PyCell Explorer tool, which can easily create different instances of our parameterized cell. This PyCell Explorer tool can be used in conjunction with the viewer tool, so that as Python code is executed, the layout which gets generated is displayed in the viewer tool. In addition, the WingIDE tool can be used to debug a Python parameterized cell, and to interactively step through the Python code to see the layout which gets generated at each step. The use of this WingIDE debugging tool is covered in the section of this tutorial covering the PyCell Studio product.

We will now show how the PyCell Explorer tool can be used to create different instances of our parameterized cell design, and how these instances are displayed in the viewer tool. The Python interpreter `cnpy` is used to invoke this PyCell Explorer tool:

```
cnpy

>> import cni.exp
>> openLib('MyTutorialPyCellLib')
>> explore('MyTesting', showGui=True)
```



```
Welcome to

            Copyright (c) 2006-2007 Ciranova, Inc.  All rights reserved.

Type 'openLib(libName)' to open a library and 'explore(dloName)' to
interactively explore the generation of a DLO. Use 'createLib(...)' to
create new libraries.

Try 'help(openLib)', 'help(createLib)', or 'help(explore)' to find out
more about how to use these functions.

>>> openLib('MyTutorialPyCellLib')
>>> explore('MyTesting', showGui=True)

Start exploring MyTutorialPyCellLib/MyTesting ...
MyTesting> []
```

The `import cni.exp` command imports the Python code for the PyCell Explorer tool, while the `openLib()` command opens up the 'MyTutorialPyCellLib' parameterized cell library which we just created using the `cngenlib` utility. Finally, the `explore()` command opens up a new design for us to use to create instances of our parameterized cell. Note that this command also has an option `showGui` to bring up the viewer tool.

At this point, we are now ready to create instances of our parameterized cell design, by using the Python API `Instance` command. For example, an instance of our MOS transistor which uses default parameter values can be created as follows:

```
p = ParamArray()
connDict = {'S':'S', 'D':'D', 'G':'G', 'B':'B'}
tran1 = Instance('MyTransistor', p, connDict, 'tran1')
```

The first parameter for this Instance command specifies the library name and cell name for the parameterized cell, while the second parameter specifies the array of parameter values which should be used when this parameterized cell is created. The third parameter is a Python dictionary which defines the connectivity which should be used for this instance, while the last parameter is the name to be used for this instance of our parameterized cell.

Using the PyCell Explorer, we can easily write Python code which will create different instances of our transistor unit, each of which has different parameter values. This approach allows us to quickly test different combinations of parameter values to make sure that our parameterized cell is generating the expected layout for different parameter settings. For example, we can create MOS transistors with bars, and without bars. In the case where bars are specified, there should also be routing between the bar contacts and the transistor source, drain and gate. The following Python code will create two different transistors, one with bars (the default), and one without bars:

```
p = ParamArray()
connDict = {'S':'S', 'D':'D', 'G':'G', 'B':'B'}

p['bars'] = True
tran1 = Instance('MyTransistor', p, connDict, 'tran1')

p['bars'] = False
tran2 = Instance('MyTransistor', p, connDict, 'tran2')
```

In addition, we can create transistors with multiple fingers, as well as with contact rings. The following Python code will create two transistors, one with multiple fingers, and one with a contact ring enclosing the transistor; all other parameters will have default values.

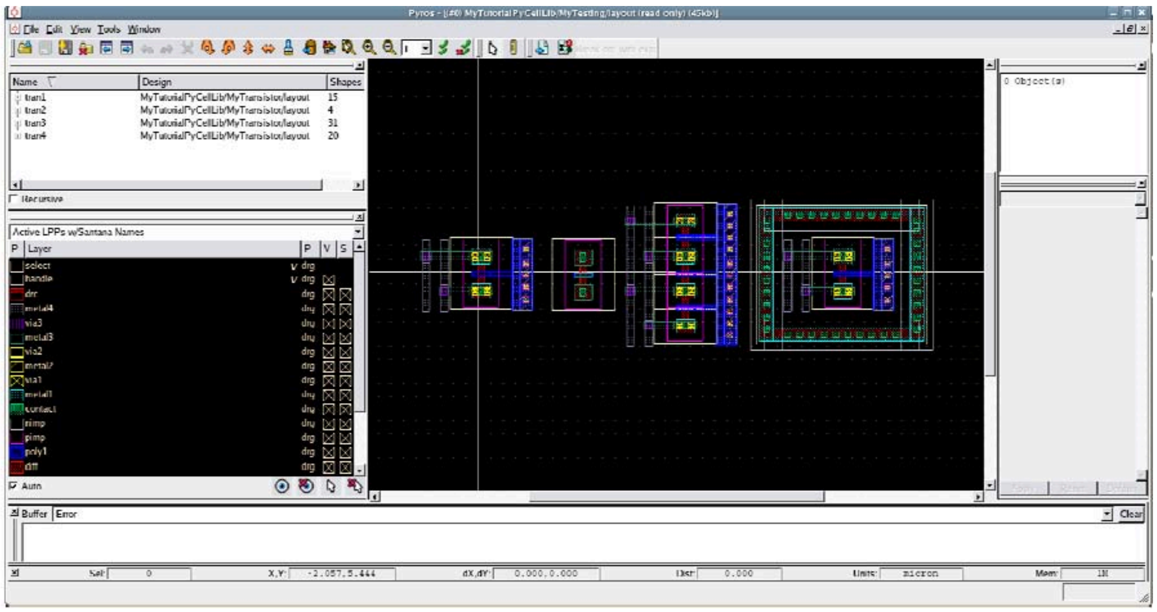
```
p['bars'] = True
p['fingers'] = 3
tran3 = Instance('MyTransistor', p, connDict, 'tran3')

p['fingers'] = 1
p['substrateContact'] = True
tran4 = Instance('MyTransistor', p, connDict, 'tran4')
```

In addition, we can also use the Python API functions to `smart place` these instances in specified relationships to each other. For example, the following Python code will place these transistors next to each other:

```
fgPlace(tran2, EAST, tran1)
fgPlace(tran3, EAST, tran2)
fgPlace(tran4, EAST, tran3)
```

We can then look at the layout which gets generated for each of these transistor instances in the viewer tool, and visually verify that the generated layout is correct. For example, it is easy to see that bars and the associated routing get generated as expected. For the case of multiple fingers, we can also see that the correct number of fingers get generated, and that the source and drain contacts are overlapped. In a similar fashion, it is very easy to see that a contact ring gets generated to completely enclose and isolate the transistor instance.

If the above Python source code has been entered and successfully executed, then the viewer tool should look like the following, after using the `Zoom Home` toolbar command:


Pyros - [#0) MyTutorialPyCellLib/MyTesting/layout (read only) (45kb)]
File Edit View Tools Window
Recursive
Active LPPs w/Santana Names
Auto
Buffer
Error
Clear

Name	Design	Shapes
tran1	MyTutorialPyCellLib/MyTransistor/layout	15
tran2	MyTutorialPyCellLib/MyTransistor/layout	4
tran3	MyTutorialPyCellLib/MyTransistor/layout	31
tran4	MyTutorialPyCellLib/MyTransistor/layout	20

This graphical display makes it very clear whether the proper layout is being generated for the different combinations of parameter settings for our parameterized cell design.

Note that using this approach with the PyCell Explorer, it is very easy to write Python code which will test various combinations of parameter settings for the parameterized cell. In addition, the immediate graphical feedback from the viewer tool allows possible layout generation problems to be detected. For example, the Python language has a built-in `random` module which can be used to generate random values for different parameter settings. Thus, it is possible to generate fairly complete test suites for a parameterized cell using this PyCell Explorer tool.

Verifying the Generated Layout

Now that we have viewed the layout which will be generated for our MOS transistor parameterized cell, the issue of physical verification should also be briefly discussed. Although we can visually inspect the layout which is generated for different parameter settings, this approach will only allow us to find obvious errors in the generated layout. One approach is to run one of several stand-alone physical verification programs, which are available from various EDA vendors, such as Calibre (Mentor Graphics), Assura (Cadence) or Hercules (Synopsys). Another approach is to make use of the convenient interactive DRC program which is already provided in the viewer tool. This interactive program is intended for use with parameterized cells (versus full-chip layout verification), and the results of running DRC are graphically presented in the viewer tool. Thus, this is a very convenient way to verify the layout which gets generated for a parameterized cell.

Note that the higher-level objects in the Python API are already `DRC correct by construction`. This is handled by the `Geometry Engine` which underlies these higher-level design objects, as well as all of the `design rule aware` class methods, such as the `fgPlace() smart place` method, which was discussed earlier in this Tutorial. This interactive DRC program makes use of this same `Geometry Engine` technology to provide a convenient means to perform design rule checking. Although these higher level objects will be constructed according to the design rules specified in the technology file for the process which is being used, it is still necessary to perform design rule checks on parts of the layout which are manually generated, or constructed through the Python API, without making use of these higher-level design objects.

In order to run this interactive DRC program, simply click on the double-check toolbar icon in the viewer tool. This will execute the DRC program as a separate thread (for performance reasons), and if any DRC errors are found, then special DRC markers will be generated on the Santana DRC layer. These error markers will be generated next to the geometric shapes which contain the DRC errors, so that it will be easier to find and debug these DRC errors.

In order to see how this DRC program works with the MOS Transistor which we have created in this tutorial, invoke the PyCell Explorer as before, and then create a single instance of our MOS Transistor (which uses default values), as follows:

```
p = ParamArray()
connDict = {'S':'S', 'D':'D', 'G':'G', 'B':'B'}
tran1 = Instance('MyTransistor', p, connDict, 'tran1')
```

Now simply click on the double-check toolbar icon in the viewer tool. This will execute the DRC program, and display a dialog with the results. If all of the Python source code was correctly entered (or you used the sample Python code which was shipped with this Tutorial), then there should not be any DRC rule checking errors generated when you run the DRC program.

Note that the Transistor Unit which was created earlier in the Tutorial did have some DRC checking errors. Please refer to the section entitled `Verifying the Generated Layout` for the Transistor Unit example, to see how design rule checking errors are generated and handled by the `pyros` viewing tool.