Using the Python API to Develop Process Portable PyCells

L-2016.06

Copyright and Proprietary Information Notice

Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at http://www.synopsys.com/Company/Pages/Trademarks.aspx.
All other product or company names may be trademarks of their respective owners.

Third-Party Links

Any links to third-party websites included in this document are for your convenience only. Synopsys does not endorse and is not responsible for such websites and their practices, including privacy practices, availability, and content.

Synopsys, Inc.
690 E. Middlefield Road
Mountain View, CA 94043
www.synopsys.com

CONTENTS

Introduction

The Python API is a powerful Python-based programming environment for developing parameterized cells and cell generators. This Python API is composed of six major class groups, which together contain over 60 specialized classes, and almost 700 class methods for the development of parameterized cells. However, in order to most effectively make use of the power and capability of this Python API, there are some general guidelines and principles which should be followed during the development of Python source code for a parameterized cell.

This Python API provides a powerful programming environment which can be used to develop parameterized cells which are Design Rule Correct by Construction. The same Python source code can also be used to generate parameterized cell layouts for different process technologies. This can all be done using a relatively small amount of Python source code for each of the parameterized cells in a library.

This paper briefly describes the general principles and guidelines which should be used to develop process portable PyCells™ using the Python API. Guidelines are also given for the creation of Santana technology files which support process portable PyCell development. In addition to describing these conceptual guidelines, several Python source code fragments are used to illustrate how these guidelines and principles are applied in practice. The careful use of these guidelines will allow the PyCell developer to create parameterized cells which can then be migrated between different process technologies.

General Guidelines and Principles of PyCell Programming

Although the Python API contains hundreds of class methods designed specifically for the development of parameterized cells, there is a small subset of class objects and class methods which are central to the power of the Python API. The two most important parts of the Python API are the “FG” methods and the “Smart Object” classes.

The FG methods are a small number of methods which automatically handle design-rule correct placement or spacing operations for physical components in the layout. These methods make use of the entire placement and spacing related design rules which have been defined in the technology file. Thus, it is not necessary to look up and make use of individual design rules when placing components in a layout design. Instead, these FG methods utilize all applicable spacing related rules for all relevant layers to determine the proper design rule correct spacing. This makes it much easier to correctly space components in a layout design. In addition, these FG methods play an important role in making it possible to compile a single Python PyCell for different process technologies.

The Smart Objects are classes which provide basic layout design objects for use by the PyCell developer. These Smart Objects include various types of contact objects (Contact, AbutContact, ArrayInstContact) as well as a ContactRing object. In addition, the Bar, RoutePath and MultiPath objects are provided for routing between different components in a layout design. These Smart Objects are completely design rule aware, so that whenever one of these Smart Objects is used in a design, it will consider all applicable design rules. For example, the ContactRing will consider all applicable minimum spacing and enclosure design rules for all relevant layers when it is created.

There are four fundamental principles of process portable PyCell programming:

- Using FG methods
- Avoid Specific Design Rule Lookups
- Using “Smart Objects”
- Importance of Ordering Layout Construction Steps

Note that all of these principles should be used together; these principles are not listed in any particular order. In fact, they are inter-related, in that they all are necessary to easily and effectively create PyCell designs which design rule correct by construction.

Programming for Process Portability and Migration

The Python API contains hundreds of class methods designed for developing parameterized cells, so that the PyCell developer can develop parameterized cells in many different ways. For example, there is an extensive set of methods for technology database access, so that the PyCell developer can easily reference specific design rule values. However, this approach makes it much harder to develop process portable parameterized cells. The Python API also allows the developer to separate layout intent from process specific details, by using the FG methods and the smart objects. These FG methods and smart objects automatically consider all relevant design rules, so that it becomes easier to migrate a parameterized cell design from one process technology to another one. For example, as the underlying design rules in the technology file change, then these smart objects will automatically construct themselves according to different design rules, to generate the proper layout.

When we say that a parameterized cell becomes “process portable”, we mean that the design for the PyCell is robust against process changes defined in the technology file. In particular, through the use of these FG methods and smart objects, the PyCell should be migratable using the same process node (or line width) from several different foundries. In addition to these FG methods and smart objects, the use of Object Oriented Programming (OOP) allows the PyCell developer to more easily handle any architectural differences between process technology nodes. In summary, the use of the FG methods

and the smart objects makes the PyCell designs much more robust against any process changes and much more portable between process technologies.

Using FG Methods

The FG methods should always be used, whenever there are any spacing or placement operations required in the design of a parameterized cell. These FG methods will consider all applicable spacing related design rules defined for all layers, in order to determine the proper design rule correct spacing for components in a layout design. In particular, these methods make use of all design rules defined in the technology file, which report DRC errors using the "SPACING" operation. Note that a given technology file will typically contain numerous design rules related to spacing; in addition, different technology files will generally have different numbers and types of spacing design rules.

Since these FG methods automatically consider all of these different spacing related design rules to determine the required minimum spacing, they can determine the proper minimum spacing for any technology file. This important capability allows the same Python PyCell source code to be more portable between different technology files. Instead of the PyCell developer attempting to cope with variations between technology files, the same Python source code can simply be reused for each technology file. It is always best to let the FG methods perform any "heavy lifting" for spacing operations.

The two most important FG methods are the following:

(1) **fgPlace**() – places component in layout design rule correct

(2) **fgMinSpacing**() – calculates design rule correct spacing distance for component

(3) **fgEnclose**() – generates design rule correct enclosing rectangles

The **fgPlace**() method can be used to place one component relative to a reference component in the layout; this placement will be design correct by construction, using all spacing design rules specified in the technology file. The **fgMinSpacing**() method is very similar to **fgPlace**(), except that instead of performing the placement of the component, it instead returns the calculated minimum spacing value distance between components. That is, **fgPlace**() has the same functionality as **fgMinSpacing**() followed by the explicit **place**() method. The **fgEnclose**() method is used to generate all enclosing rectangles around a specified component; these enclosing rectangles are generated to be design rule correct by construction. This method is typically used to easily create implant or well layers for a parameterized cell.

How to Avoid Using Design Rule Lookups

The Python API allows the PyCell developer to easily access the technology database to find the specified value for a particular design rule. For example, the minimum width or spacing values for design objects on a metal layer can be easily accessed. However, the goal when creating any parameterized cell should be to avoid any specific design rule lookups, unless absolutely necessary. There are many reasons for this goal. One key factor is that many design rules need to be considered. For example, even to calculate minimum spacing values, several design rules are usually involved, including the more complex conditional design rules used for wide metal spacing values. Note that design rules in the technology file can be described using different approaches; procedural design rules can be used, as well as simple declarative name-value pairs. In addition, the number (and complexity) of design rules is steadily increasing, with each new generation of process technology. The task of simply managing all of these different design rules is becoming a major challenge for the PyCell developer.

Another issue with using specific design rule lookups is that technology files may differ in terms of the design rules which are used to express all of the spacing rules. Besides different design rule names, the technology file may use different and/or additional design rules. As a result, it becomes more difficult to write PyCell source code which can be used with different technology files. By using the FG methods and the Smart Objects, any such differences in design rules are automatically handled for the PyCell developer.

What FG Methods Can and Cannot Do

Although the goal is to avoid any specific design rule lookups in the design of a parameterized cell, it is sometimes necessary to use specific design rule lookups. For example, the FG methods can not consider all possible design rules defined in the technology file; these FG methods are only concerned with spacing design rules. More specifically, the FG methods can only make use of design rules which are defined using the “SPACING” or “ENCLOSURE” operations. Thus, if it is necessary to consider other types of design rules, it will be necessary to query the technology database for different design rule values. For example, there are specific design rules which are used to specify the minimum enclosed area for shapes on metal layers. These particular design rules are specified using the “HOLES” operation, so that these design rules will not be utilized by the FG methods. As another example, it may be necessary to obtain minimum width values for shapes on different layers when specifying (or validating) parameter values for a parameterized cell. For example, the width of a poly resistor would need to be at least as large as the minimum width for the poly layer.

Using “Smart Objects”

The Smart Objects should be used in the construction of layout for a parameterized cell design. For example, if a contact ring is needed, then the Smart Object ContactRing should be used. If for some reason, the Smart Object is missing some specialized piece of functionality, then a new class can be derived. These Smart Objects raise the level of abstraction, so that the resulting Python code is easier to write, easier to read and easier to understand. For example, a contact ring can be created with just a single line of Python code, which is easily written by the original PyCell developer, and can be quickly read and understood by other PyCell developers.

Since these Smart Objects automatically consider all relevant design rules when they are created or modified, they can be used to produce more portable PyCell code. Any changes in the technology file are automatically reflected by the Smart Object. As an example, the Contact will make use of all relevant minimum width, minimum spacing, minimum extension, minimum enclosure and minimum area design rules. In addition, if design rules are defined in the technology file for minimum via enclosure, minimum via spacing for neighbors or minimum number of cuts for via widths, then these design rules will also be applied.

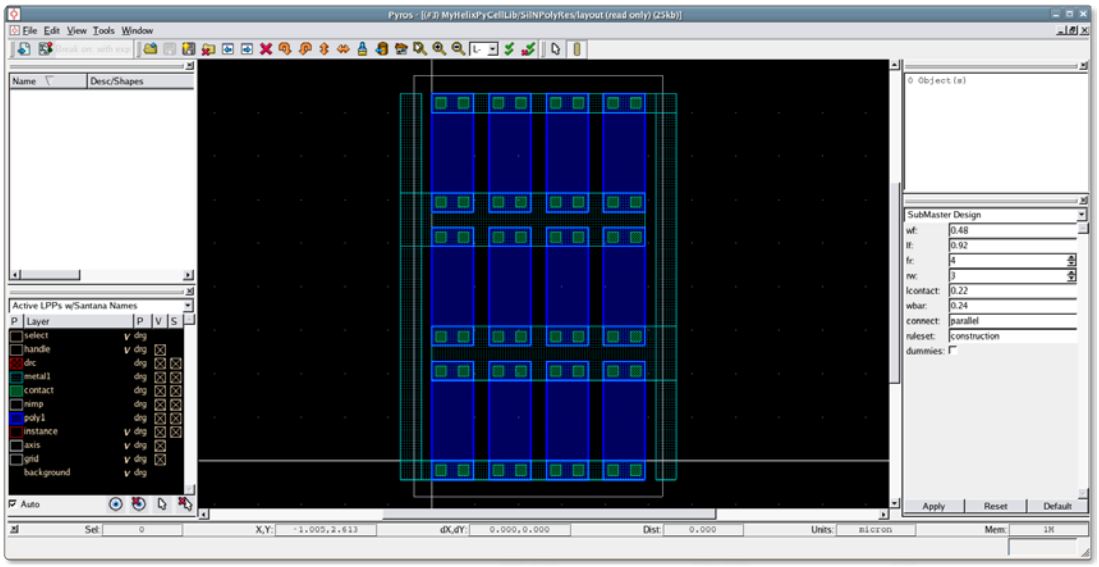
A related benefit of using these Smart Objects is that these are existing higher-level layout design components, which have been used with many different technology files, as well as many different PyCell designs. Thus, these higher-level components will not have the errors that similar components will have which are created and used for the first time by the PyCell developer. Although the PyCell developer can create their own Smart Objects, it is better to derive a new class from one of these existing Smart Objects, and extend the large amount of functionality already contained in these classes.

Importance of Ordering Layout Construction Steps

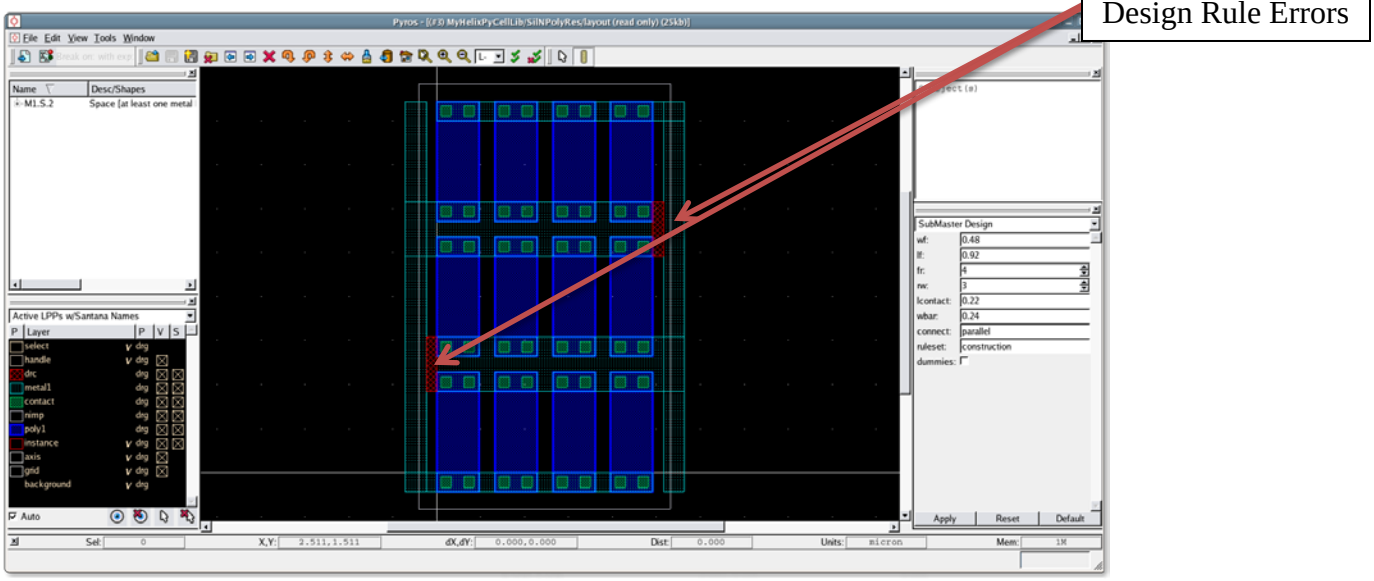
As already described, the FG methods are very powerful, and can be used to create PyCell designs which are portable across multiple technology files. However, it is important to bear in mind that these FG methods can only operate with the information which is passed to them when they are called. In particular, the calculated minimum spacing values will be a function of the current state of the layout design when the particular FG method is called. If additional components or shapes are added to the layout after the FG method has been called, then this information will not have been considered when spacing calculations were performed. Thus, it is important for the PyCell developer to carefully visualize the different steps involved in generating the layout for a parameterized cell design.

For example, suppose that we want to construct the layout for a multi-fingered resistor parameterized cell design. As part of this layout, we want to place Bars at either end, to make it easier for the routing tool to connect to the terminals for this resistor. In addition,

we will use the Python API to connect together the different fingers of this resistor, either as a series or parallel connection. If the fingers of this resistor are connected in parallel, then this resistor parameterized cell would look like the following:

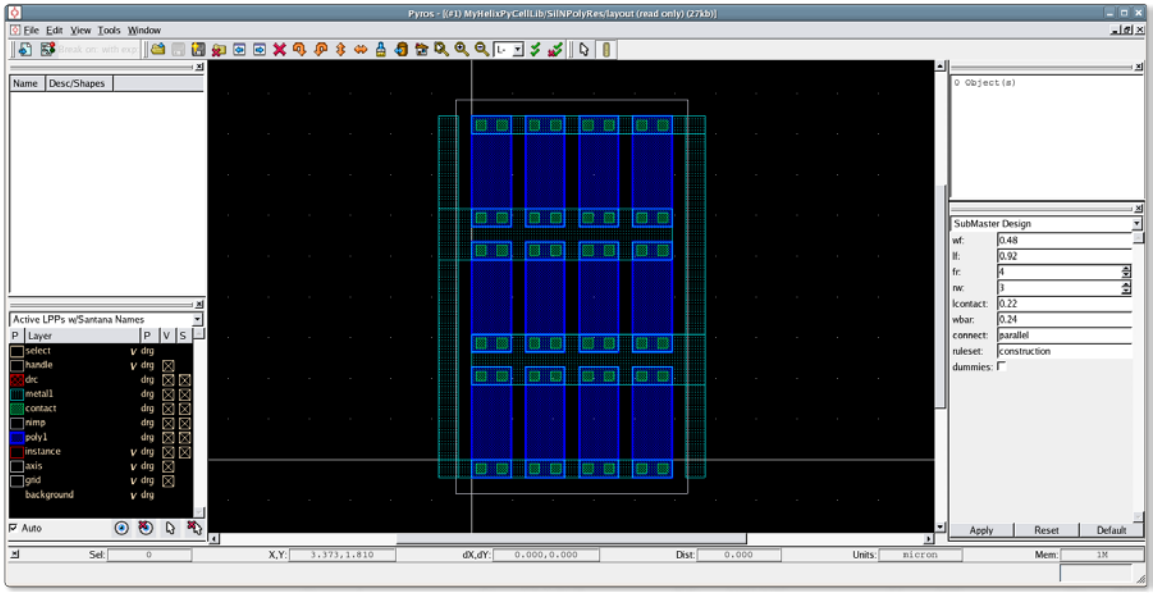


In order to place these Bars at either end of the layout, we can use the **fgPlace**() method to properly space them from the resistor fingers. However, it is important to carefully consider when this **fgPlace**() method should be called in the sequence of steps used to create the layout. We could place these Bars right after the individual resistor fingers have been created, before these fingers have been connected together. However, this approach could possibly lead to problems. The issue is that in the process of routing together these resistor fingers, it may be possible to create large metal shapes used for Mrouting, so that wide metal design rules may be violated. For example, the following screen shot shows the results of running the DRC program when these Bars were placed before the fingers were connected together:



In this case, the metal routing rectangles which were created to connect the individual fingers actually created wide metal regions. Hence, the spacing between these Bars and the resistor fingers is not large enough to account for these wide metal design rules.

Thus, a better approach would be to call the **fgPlace**() method to place these Bars after the individual resistor fingers have been connected together. If this is done in the PyCell source code, then the spacing between the Bars and the metal routing rectangles will automatically be larger to account for the wide metal which was generated, as shown in the following screen shot:



The key point to remember is that the PyCell designer needs to carefully think about the sequence of steps involved in the generation of layout for the parameterized cell.

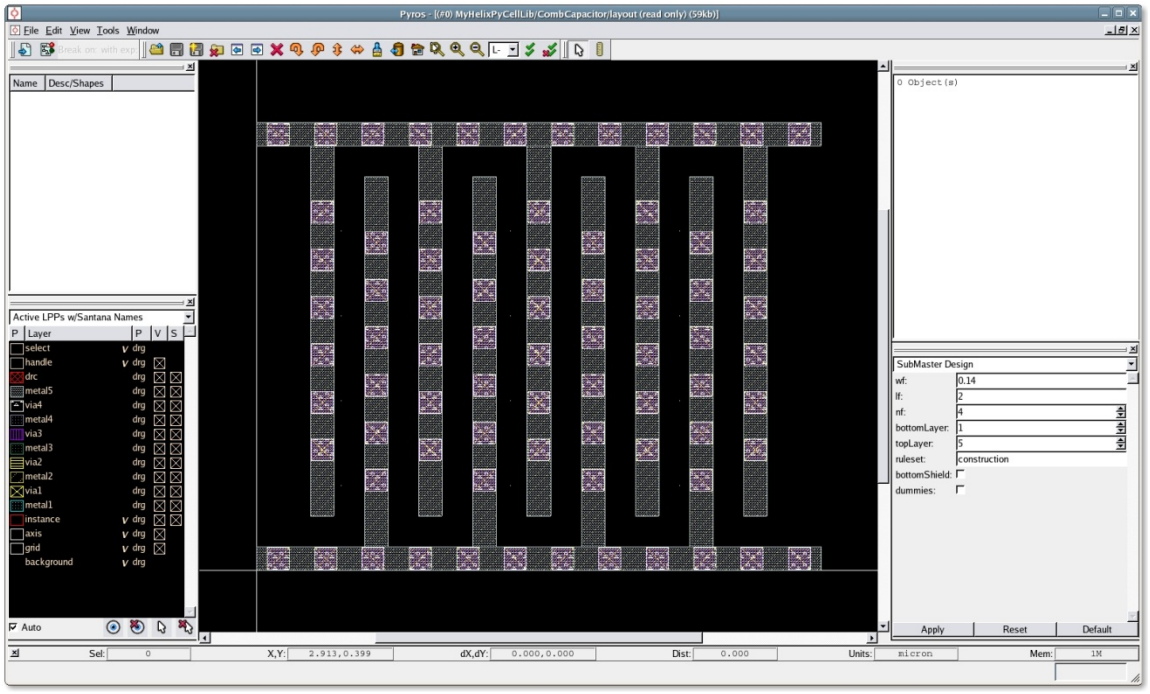
Another approach which can be used with the FG methods is to create dummy objects which can be used to correctly calculate the proper minimum spacing value. Thus, instead of calling **fgPlace**() after all of the routing for the resistor fingers has been completed, dummy routing shapes could be created before calling **fgPlace**(), and then these dummy shapes would be deleted from the generated layout. The basic idea is that it is important to give these FG methods as much information as possible. In fact, there is an “env” environment parameter, which allows the PyCell developer to inform the FG method about other shapes in the layout design which should be considered when calculating spacing values.

No matter which of these approaches is used, the PyCell developer should always carefully think through the sequence of steps involved in generating the layout, in order to produce a more portable parameterized cell design.

Python Source Code Example

In this section, we briefly discuss the design of a parameterized cell, and then use several Python source code fragments to illustrate the general principles of PyCell Programming which have been presented in this paper. Note that complete Python source code for this parameterized cell is available as part of the PyCell Studio™ download.

Let's now consider the basic design for a metal comb capacitor parameterized cell. This comb capacitor is composed of several interdigitated metal fingers, which are spaced together as closely as possible, in order to maximize the fringe capacitance which is generated by these metal fingers for the comb. This parameterized cell would have parameters such as: number of fingers, width of each finger, length of each finger, bottom metal layer number and top metal layer number. In addition, an option could be provided to generate a shield plate on the bottom metal layer. This comb capacitor parameterized cell would look like the following screen shot:



The main design challenge for the PyCell developer is to use the values for these parameters entered by the user to calculate the design rule correct spacing between the individual fingers of this comb capacitor. Once this has been determined, then the individual fingers of the comb capacitor can be created and properly placed in the generated layout for the parameterized cell. In addition, the PyCell developer also needs to figure out how to best generate the layout for the bottom shield plate, when this option has been selected by the user. We can make use of the power and capability of the FG methods to determine the proper design rule correct spacing for the fingers of this comb capacitor. It is simply a matter of creating a sample finger on each metal layer, and then

using **fgMinSpacing**() to determine the minimum spacing for that metal layer. Note that in order to generate a very regular layout structure, we will determine the largest spacing value for any of these metal layers, and then use that spacing distance for all metal layers when constructing the comb capacitor layout structure.

In order to use **fgMinSpacing**(), it is first necessary to create temporary fingers for the comb capacitor. This can be done by making use of the pre-defined AbutContact Smart Object. This is a special Contact object, which is used when one or more contacts will be abutted. In our case, we will abut the fingers of the comb capacitor to the body of the comb, so this is the best Smart Object for our design. Note that for the top metal layer, no AbutContact object is necessary, since there is no higher metal layer which needs to be connected to the top metal layer. In this case, we simply use a Bar Smart Object, in place of this AbutContact object.

This calculation of the required spacing between the fingers of our comb capacitor can be performed using the following Python code fragment. Note that a simple Python method was previously written to determine the metal layers which should be used for our comb capacitor; this information is stored using the list class variable `self.metalLayers`.

```
def calculateFingerSpacing(self):
    w = self.width
    l = self.length
    self.fingerSpacing = 0
    for i in range(len(self.metalLayers)):
        layer = self.metalLayers[i]
        # if not top layer, create vias to above metal layer
        if layer != self.metalLayers[-1]:
            contact1 = AbutContact(layer, self.metalLayers[i+1],
                                   routeDir1=NORTH_SOUTH,
                                   routeDir2=NORTH_SOUTH,
                                   abutDir=NORTH,
                                   point1=Point(0,0),
                                   point2=Point(w,l))
            contact2 = AbutContact(layer, self.metalLayers[i+1],
                                   routeDir1=NORTH_SOUTH,
                                   routeDir2=NORTH_SOUTH,
                                   point1=Point(0,0),
                                   point2=Point(w,l))
            spacing = fgMinSpacing(contact1, EAST, contact2)
            contact1.destroy()
            contact2.destroy()
            if self.spacing > fingerSpacing:
                self.fingerSpacing = spacing
        else:
            # since no vias required, just use Bars
            bar1 = Bar(layer, NORTH_SOUTH, 'B1',
                       Point(0,0), Point(w,l))
            bar2 = Bar(layer, NORTH_SOUTH, 'B2',
                       Point(0,0), Point(w,l))
            spacing = fgMinSpacing(bar1, EAST, bar2)
            bar1.destroy()
            bar2.destroy()
```

```
            if spacing > self.fingerSpacing:
                self.fingerSpacing = spacing
```

Note that after this class method has been called, the variable `self.fingerSpacing` will contain the minimum design rule correct spacing between fingers which should be used when the fingers for our comb capacitor are created and placed in the generated layout. Also note that only a very small amount of Python PyCell source code was required to make this somewhat complicated spacing calculation. This is possible, since we are making use of the power of the **fgMinSpacing**() method, as well as the AbutContact. This AbutContact object will automatically handle the required spacing for the vias on each of the metal layers. Also note that in the case of the top metal layer, instead of using a specific design rule lookup to determine the proper metal spacing between fingers, we used a Bar object, in conjunction with **fgMinSpacing**() to measure the required spacing. Thus, this short Python code fragment clearly illustrates the first three Principles of PyCell programming which were previously discussed.

Once the proper spacing for the fingers of our comb capacitor has been determined, then it is a relatively simple matter to properly place these fingers to create the comb capacitor layout. This can be done as shown in the following Python source code fragment:

```
def createFingers(self):
    # generate fingers for each comb on each metal layer
    for i in range(len(self.metalLayers)):
        # ignore bottom layer, if it was used to create a shield
        if self.bottomShield and i == 0:
            continue
        layer = self.metalLayers[i]
        # generate fingers for terminal1 comb
        for j in range(self.fingers+1):
            w = self.width
            l = self.length + self.fingerSpacing
            bar1 = Bar(layer, NORTH_SOUTH, 'B1',
                       Point(0,0), Point(w,l))
            # if this is not top layer, create vias
            if layer != self.metalLayers[-1]:
                contact1 = AbutContact(layer,
                                       self.metalLayers[i+1],
                                       routeDir1=NORTH_SOUTH,
                                       routeDir2=NORTH_SOUTH,
                                       abutDir=NORTH,
                                       point1=Point(0,0),
                                       point2=Point(w,l))
            # abut this bar just below terminal1 Bar
            bar1.abut(SOUTH, self.term1Bar)
            bar1.alignEdge(WEST, self.term1Bar)
            # place this finger to the right of previous finger
            bar1.moveBy(self.width + self.fingerSpacing, 0)
            bar1.moveBy(j*(2*self.width+2*self.fingerSpacing), 0)
            if layer != self.metalLayers[-1]:
                contact1.alignLocation(Location.UPPER_LEFT, bar1)
```

Note that the generation of the fingers for the second terminal2 comb is almost identical to the code shown for the first terminal1 comb; it has been omitted for brevity. In this sample code, we construct an AbutContact as before, but then we explicitly position this Smart Object using the **moveBy**() method, along with the `self.fingerSpacing` value, which was previously computed using **fgMinSpacing**().

It should be noted that due to the power and capability of the FG methods and the Smart Objects, we just basically needed to use two Python for-loops to perform the generation of the layout for our comb capacitor parameterized cell. There is of course, more Python code than these two fragments in the complete source code for this example, but these two fragments are the critical elements in the generation of the layout for our comb capacitor parameterized cell.

One other point which should be mentioned is that the Python source code for this comb capacitor parameterized cell also illustrates another of the fundamental principles of PyCell programming. Namely, the principle that direct design rule lookups should be avoided, unless necessary. In the case that the bottom metal layer is used as a shield layer, then it is necessary to extend the fingers of each comb, so that these fingers are connected to each comb body, to create the shield. In this particular case, we need to make sure that the metal areas enclosed by these extended fingers do not violate design rules. Thus, in this case, we need to use an explicit design rule lookup, since this type of design rule cannot be expressed in terms of spacing operations. This is shown in the following Python source code fragment:

```
# check for minimum metal enclosed area rules;
# this specific design rule check is required,
# since FG methods do not handle "HOLES" operations.
# This is only necessary for the bottom shield case.

if self.bottomShield:
    layer = self.metalLayers[0]
    if self.tech.physicalRuleExists('minEnclosedArea', layer):
       area = self.tech.getPhysicalRule('minEnclosedArea', layer)
       w = self.fingerSpacing
       l = self.length + 2 * self.fingerSpacing
       if (area > (w * l)):
           box1 = Box(Point(0,0), Point(w,l))
           box1.expandForMinArea(EAST, area)
           self.fingerSpacing = box1.getWidth()
```

Note that if this minimum metal enclosed area design rule will be violated, then we use special methods from the Python API to expand the spacing between the fingers of the comb capacitor.

Process Portable Technology Files

In addition to the general programming guidelines which have been presented for the development of process portable PyCells, there are also some additional guidelines for the development of technology files. Although each technology file is specific to a specific process technology, it needs to be written in such a way that it supports process portable PyCell development. This section describes the general principles and guidelines which should be applied to the development of technology files.

There are several areas which should be considered when writing technology files which can be used for process portable PyCell development, summarized as follows:

- Use same names for any common layers
- Use same names for same design rules
- Use same names and layers for any special device-specific rules
- Use device contexts for any special device constructions
- Use required design rule names used by “smart objects”
- Use SPACING and ENCLOSURE operations to be used by FG methods

Note that all of these areas should be considered; they are all important considerations in the creation of Santana technology files which help promote the development of process portable PyCells. Each of these areas will now be discussed.

First recall that different technology files can use different names for the same layer in a process technology. To allow the same PyCell code to access these layers in a uniform fashion, these layers should have the same Santana layer names in the different technology files. For example, in the case of resistors, a Resist Protection Oxide layer is used to construct oxide rectangles which are used to prevent the formation of silicide. These “silicide blocks” are used to effectively increase the resistance of the resistor. This layer is typically named in different ways by different process technology files; for example, “RPO”, “SAB”, etc. A single layer name for this purpose should be selected and then used by each technology file.

Also recall that different technology files can possibly use different names for the same design rule for a process technology. For example, the minimum spacing value for a particular layer could be specified using names such as “minSpacing”, “minSpace”, “minimumSpace” or “minimumSpacing”. To allow the same PyCell code to access these design rules in a uniform manner, design rules with the same function should select and use the same design rule name in each of the different technology files.

An additional consideration is that when certain types of devices are constructed as shapes on certain layers, then different device-specific minimum design rules will be used for these devices (versus the usual minimum design rules for that layer). For example, when a resistor is constructed on a polysilicon or diffusion layer, then the minimum width value on the poly or diffusion layers may be required to be larger when these layers are used to construct resistors. In such cases, a common name should be used for these device-specific design rules (eg: 'minResistorWidth'), so that the PyCell code can explicitly check for any device-specific design rules in a portable fashion.

The optional device context feature of the Santana technology file allows special design rule values to be used when constructing a specific device; this allows for the use of device-specific design rules in a portable fashion. For example, a special device context can be defined to be used when NWell resistors are constructed. This device context specifies different width, spacing and enclosure design rule values which should be used for the NWell layer, when a resistor device is constructed on the NWell layer. As an example, the following design rule substitutions for this device context can be defined:

```
deviceContext("N-Well Resistor" nwr
;( rule-ID       substitute-ID )
;( ------------  ------------- )
 ( NW.W.1        NWR.W.1       ) ;min resistor width
 ( NW.S.2        NWR.S.1       ) ;min resistor spacing
 ( NW.EN.2       NWR.EN.1      ) ;nwell enclosure of diff
); "N-Well Resistor" device context
```

In such cases, the technology file should define and make use of such device contexts, as it allows the PyCell code to construct special devices in a portable fashion.

As discussed earlier in this document, two of the fundamental principles of process PyCell programming involve the use of the FG methods, as well as the use of the “Smart Objects”, when PyCell code is developed. It is very important that the information in the Santana technology file be written in such a way that it supports the use of these two important PyCell process portable development principles.

As regards the use of the “Smart Objects”, recall that each of these objects in the Python API makes use of explicit design rule lookups to access specific design rule values, which are critical in their construction and modification. For example, the Contact object will automatically consider all applicable via spacing rules. Thus, the Santana technology should be written so that the required common design rule names are used for all of these specific design rules: 'minWidth', 'minSpacing', ‘minAdjacentViaSpacing’ and ‘minLargeViaArrayCutSpacing’ (for cut layers), 'minArea' (for interconnect layers) and 'minExtension' and ‘minDualExtension’ (for interconnect layers over corresponding cut layers). Note that the use of these specific design rule names is discussed in the last section of the Santana Technology File reference manual. In addition, any conditional rules should be written using the same names for the relevant parameters. For example, this currently includes the ‘minExtension’ and ‘minDualExtension’ conditional design rules for the width of the enclosing shape.
This approach helps ensure that these “Smart Objects” will incorporate and make use of all applicable design rule values.

With regard to the use of the FG methods, when DRC operations are written in the technology file, they should return DRC errors whenever possible using the SPACING or ENCLOSURE DRC operations. The reason for this is that the **fgPlace**() method will only consider DRC operations which report errors using the SPACING operation. In a similar fashion, the **fgEnclose**() method will only consider DRC operations which report errors using the ENCLOSURE operation. Thus, design rules in the technology file should be written to provide these FG methods with as much information as possible.

Conclusions

The Python API provides a powerful programming environment, which allows the PyCell developer to create advanced PyCells, which are portable between different process technologies. The knowledge built into the “FG” methods and the “Smart Objects” about design rules allows the PyCell developer to more easily generate layout for a parameterized cell design which is free of explicit references to design rules. In addition, by following the principles outlined in this paper for portable PyCell source code and technology files, then the PyCell developer can create portable parameterized cell designs, which can then be bound to different technology files.

One important side benefit of following the general principles presented in this paper is that the resulting Python source code for the parameterized cell design is more compact, more readable, more understandable and easier to maintain than source code which contains explicit references to process design rules. This approach also makes the resulting parameterized cell source code more portable between process technologies.

Note that all of the concepts discussed in this paper have been put into practice, through the development of a process independent PCell library. This library is part of the Interoperable PCell Library (IPL) project which is briefly described on the web site. In addition, the www.iplnow.com web site is the home site for this IPL project. This IPL project allows an integrated circuit designer to use the same parameterized cell library with tools from many different EDA vendors, as well as work with process technologies from multiple foundries. In addition to the PyCell Studio tools which were used to create this IPL, the actual Python source code for all of the parameterized cells in this library is also available. These library cells include active devices such as MOSFETs and differential pairs, and passive devices such as resistors, capacitors and inductors. This Python source code can be examined to better understand the principles and concepts discussed in this paper.