PyCell™ EDA Tool Integration Guidelines

Version 2022.06-SP2, December 2022

SYNOPSYS®

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 https://www.synopsys.com/company/legal/trademarks-brands.html.
All other product or company names may be trademarks of their respective owners.

Free and Open-Source Licensing Notices

If applicable, Free and Open-Source Software (FOSS) licensing notices are available in the product installation.

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.

www.synopsys.com

Feedback

Contents

1

Introduction

In order to increase design productivity and provide ease-of-use, most EDA interactive layout editing tools allow parameterized cells to have features such as stretch handles and auto-abutment capabilities. A stretch handle for a PCell allows a designer to graphically stretch and resize different shapes in the generated layout, and have the corresponding parameter values automatically updated. Thus, stretch handles provide a means for graphical manipulation of PCell parameter values. The auto-abutment feature allows the designer to overlap two parameterized cell instances, and then have the generated layout automatically adjusted to share structures, to minimize total layout area. These two PCell instances would then appear to be merged in the layout. For example, this auto-abutment feature would allow the source and drain diffusion to be merged and shared between two abutted MOSFET transistor PCell instances. These two features are very useful editing capabilities which are typically provided by most EDA interactive layout design editing tools.

Note that these stretch handle and auto-abutment editing features are defined and supported by the application layer above the OpenAccess database, but make use of OpenAccess data model constructs (properties assigned to layout shapes) to represent stretch handles and auto-abutment capabilities. In this way, the PyCell™ developer can easily provide the information which is needed by the EDA interactive layout editing tool to support these editing features.

In order to simplify the implementation of these features in the OpenAccess environment, has defined protocols which can be used to support both the stretch handle and auto-abutment features. These two protocols provide the basic communication between the PyCell developer and the EDA tool developer, to enable stretch handles and auto-abutment for PyCells. These protocols are implemented using predefined string properties for shapes which are stored in the OpenAccess database. The values for these string properties are specified by the PyCell developer, and are then used by the EDA interactive tool developer. When the designer modifies these shapes in the PyCell layout in the interactive editing environment, then the EDA tool can use these property values to re-instance the PyCell in response to stretching or abutment actions.

This document describes the interface between the PyCell developer and the EDA tool developer, with focus on the requirements for the EDA tool developer. Note that the API requirements for the PyCell developer are more fully described in the Python API Reference manual, in the section entitled “Stretch Handles and Auto-Abutment”.

2

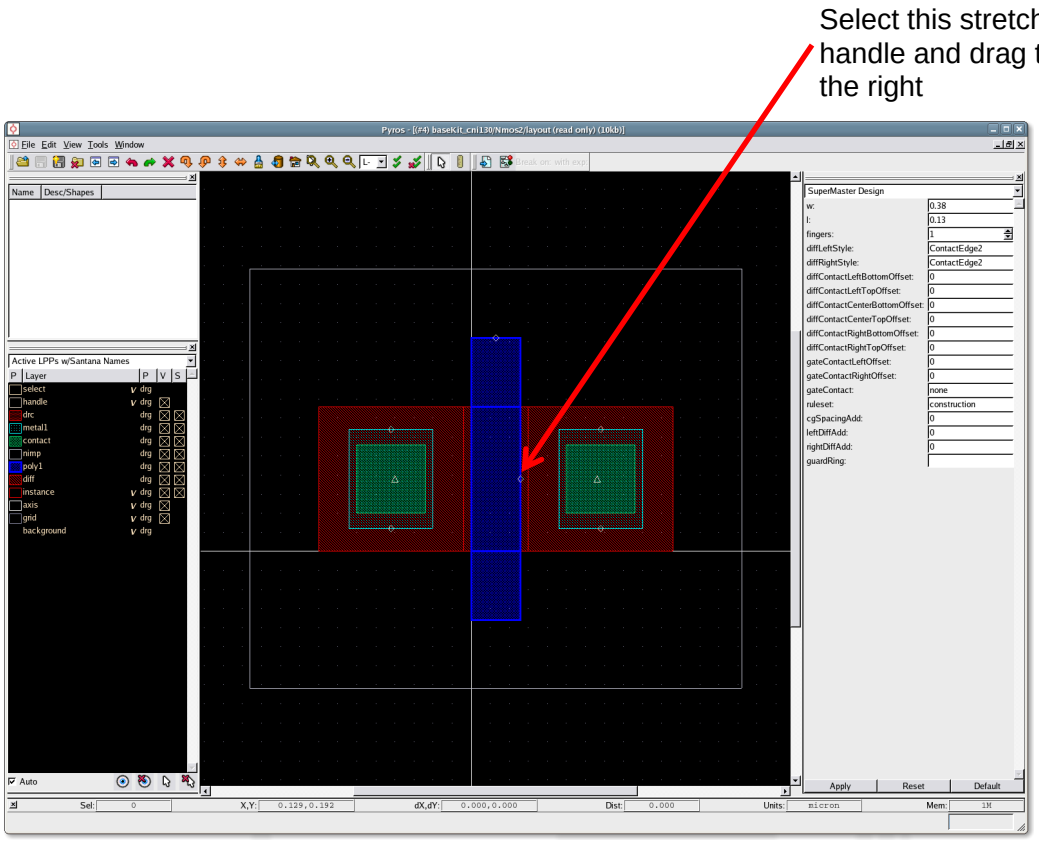
Stretch Handles

Stretch handles provide a convenient means for the user of the EDA interactive layout editing tool to graphically manipulate parameter values for parameterized cells. This is done by stretching and resizing different shapes in the PyCell layout. This graphical stretching operation causes the associated parameter values to change, and the PyCell code is then re-evaluated to create the corresponding submaster. This stretch handle is defined within the source code for the PyCell, but the graphical representation is manipulated within the EDA interactive editing tool.

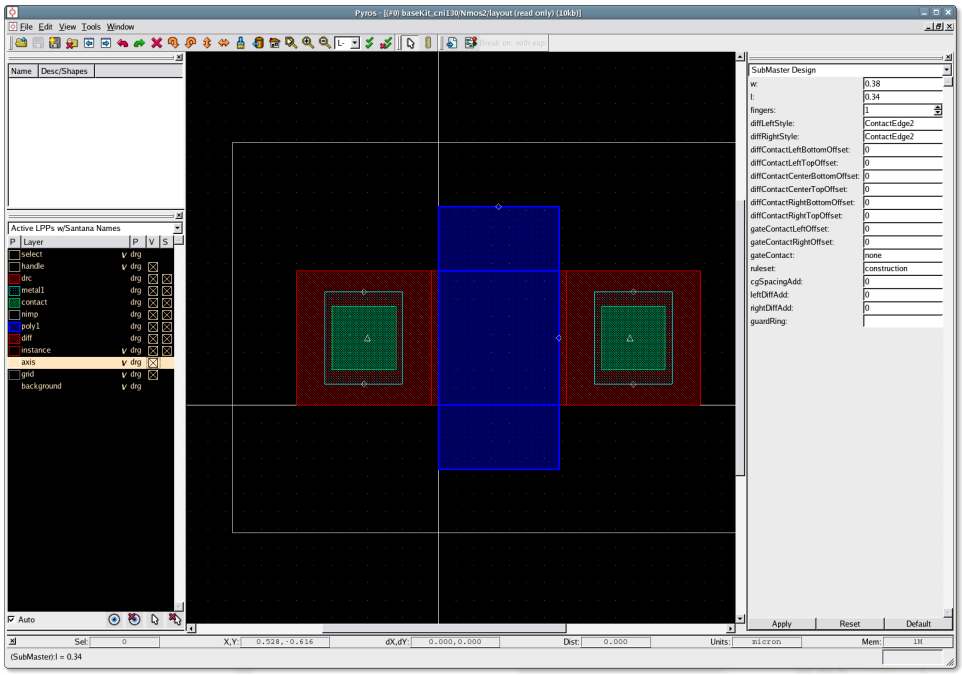
Feedback

As an illustration, the following screen shots show the use of stretch handles in the Pyros™ layout viewer to resize the length of a MOSFET transistor:

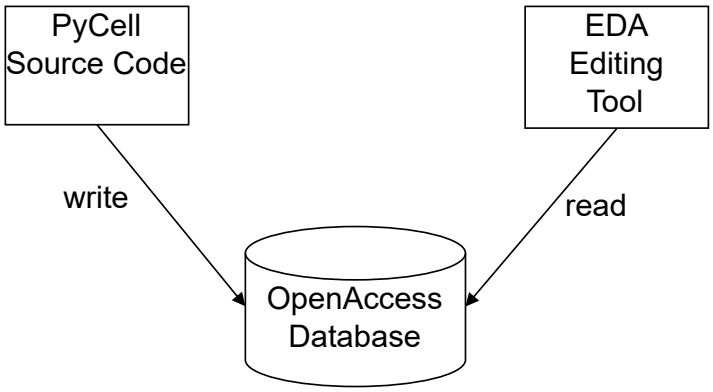
1. MOSFET transistor with default parameter values; stretch poly gate rectangle to the right to increase the length parameter:



2. MOSFET transistor with stretched poly gate poly rectangle; length parameter value has been changed from the default 0.13 to 0.34 by this stretch handle operation:



This stretch handle feature is implemented by having the PyCell developer store information about stretch handles as properties on associated layout shapes (or instances or current design object) in the OpenAccess database, which are then read and interpreted by the EDA interactive editing tool. This is as illustrated in the following diagram:



It is assumed that the EDA interactive layout editing tool already provides the capability of stretch handles in some fashion. This EDA interactive layout editing tool should provide a graphical mechanism whereby the designer can graphically use the mouse to select a

shape in the layout for a parameterized cell and then drag this geometric stretch handle to resize the shape as desired. The EDA editing tool would then use the new dimensions of the shape to update the corresponding parameter value(s) for the parameterized cell. Note that the parameterized cell can also associate stretch handles with instances or the current design object, as well as with shapes in the generated layout.

The following is the general sequence of steps which are required by the EDA tool developer to support stretch handles in their interactive editing environment:

1. Obtain all shapes in the PyCell layout which have the *pycStretch* property. This string property stores all of the information for any stretch handles which have been defined by the PyCell developer for this layout shape. Note that only shapes defined at the top-level of the PyCell instance should be examined for this *pycStretch* property. In addition, check to see if any instances (or the submaster design) have this *pycStretch* property.
2. Parse this *pycStretch* string property value to extract all stretch handle information.
3. Assign all extracted stretch handle information to the geometric stretch handle which is already provided by the EDA interactive editing tool.

These stretch handles are stored as string-valued properties on the shapes (or instances or design object) which are intended to be directly stretched or resized in the EDA layout editing tool. This pre-defined string-valued property is named *pycStretch*, and is used to store and represent the following information concerning a stretch handle:

- **name** – name for stretch handle; should be unique literal string within the PyCell. If the stretch handle is associated with an instance, then this should be the instance name; for the submaster design, this name is ignored.
- **stretchType** – `relative` or `absolute`; `relative` is increment measured relative to the center of the shape, while `absolute` is increment measured according to the absolute x and y directions.
- **direction** – direction of measurement (NORTH_SOUTH or EAST_WEST).
- **parameter** –name of the PyCell parameter which is being modified by user.
- **minVal** – minimum allowed value for the parameter being modified.
- **maxVal** – maximum allowed value for the parameter being modified.
- **location** –location point for the graphical stretch handle on the layout shape; this point should be specified using the Location class, such as Location.UPPER_CENTER.
- **userScale** – scale factor used to multiply the change in parameter value.
- **userSnap** – resolution value which should be used for snapping the parameter value.
- **key** – the name used as a key to specify values for multi-valued parameters.

- **shapeRef** – the name of referenced shape inside an instance. This item will be stored only when a stretch handle is associated with a reference to a shape in a hierarchical PCell. For example, for a stretch handle associated with a reference to a shape named "gate" inside an instance named "fet", "gate" will be stored on the "fet" instance object.

This *pycStretch* string property associated with the shape (or instance or current design object) in the generated layout is represented using the following string format:

```
name:val; stretchType:val, direction:val, parameter:val, minVal:val,
 maxVal:val,
                location:val, userScale:val, userSnap:val, key:val
```

That is, each colon-separated element of the string specifies the name and value for one of the defining characteristics of a stretch handle. These name value pairs are separated from one another by commas, while the stretch handle name is separated by a semicolon.

Note that the minimum and maximum parameter values can be one of three different types: integer, floating-point or string values. If string values are used, then these strings should represent numerical values, such as described for the Python API **Numeric** class. For example, a string value such as `1.3u` would represent the floating point number 1.3e-6, where `u` is a pre-defined scaling factor corresponding to microns.

Note that multiple stretch handles may be associated with a single shape. For example, multiple stretch handles could be associated with the poly gate rectangle shape for a MOSFET transistor, so that the designer could stretch this poly gate in both directions, so that both the length and width of the transistor could be modified using stretch handles.

In addition, stretch handles should not overlap nor be located at the same coordinates.

It should also be noted that a stretch handle can be associated with more than one PyCell parameter value. That is, one stretch handle operation for the stretch handle can cause changes in multiple parameter values at once. For example, the designer could stretch a rectangle so that the stretching operation would cause multiple parameter values to be updated; these parameter value changes would be proportional to the distance stretched.

Stretch handles can also support parameters which have multiple values. For example, independent stretch handles can be defined for each Source/Drain contact for a transistor. Parameters with multiple values are handled through the use of the **key** keyword. When this **key** keyword is None, then the PyCell parameter specified by the **parameter** keyword is a single-valued parameter. However, if this ***key*** keyword is any other string value, then it is the name of one of the multiple values for the multi-valued parameter. The value for this multi-valued parameter is defined as a comma separated string of key names and values. For example, the value for a multi-valued parameter used to specify the top and bottom offset values for a transistor contact could be expressed as `top:0.10, bottom:0.15`, where the colon character is used to separate the key names and values.

In order to improve the performance of the EDA interactive layout editing tool, note that all of the shapes (or instances) which provide stretch handles will be stored using an

oaGroup object, which is also named *pycStretch*. The EDA tool should first check for this group object on the PyCell master, and if it exists, then the EDA tool can directly access all shapes in the layout which are assigned stretch handle properties. In addition, the EDA editing tool should also check the submaster design for the *pycStretch* property, since stretch handles can be attached to design objects, and design objects cannot be stored using an oaGroup. Alternatively, all of the shapes (or instances) in the generated layout can be searched by the EDA tool for these stretch handle properties. Note that only top-level shapes which are defined at the top-level of the PyCell instance should be checked for stretch handle properties.

Although each EDA interactive layout editing tool will be different, the following is the general sequence of steps which should be followed in order to support this protocol for stretch handles. It is assumed that the EDA tool already supports graphical stretch handles, and that PyCell parameters can be assigned to these graphical stretch handles:

1. Check all of the PyCell masters in the design for the *pycStretch* group object
2. If this *pycStretch* group is defined for a PyCell master, then obtain list of top-level shapes (or instances) which have the *pycStretch* property.
3. Also check the submaster design for the *pycStretch* property (as design objects cannot be stored in an oaGroup).
4. If *pycStretch* property is defined, then parse *pycStretch* property string
5. For each handle defined in property string, extract all ten items from property string
6. Calculate corresponding PyCell parameter value(s) for the stretch handle
7. Assign all extracted values for the stretch handle to geometric stretch handle

The calculation of the PyCell parameter value described in step 5 above can be performed using the following formula:

```
parameter = max( minVal, min( maxVal,
              parameter + round( (userScale * stretchIncrement) /
 userSnap) * userSnap))
```

Note that the values of the minVal, maxVal, userScale and userSnap variables in this formula can be obtained from the values stored in the *pycStretch* property string. If these values are represented as string values, then they should first be converted to the corresponding floating-point values, before being used in this formula. The value of the stretchIncrement variable in this formula is obtained by the EDA tool developer, by measuring the distance which the designer stretched the shape for the stretch handle.

This sequence of steps can also be expressed using the following Python-like pseudo-code, where it is assumed that a function named `parseString` can be used to parse the property string assigned to the *pycStretch* property, and then extract the different stretch handle characteristics which are stored in this property string.

Feedback

```
# Assign stretch handles to all PyCell submasters in design

# It is assumed that the EDA editing tool provides graphical
# stretch handles, to which PyCell parameters can be assigned.

for PyCell masters in design:

   if PyCell master has "pycStretch" group object:

     # obtain list of top-level shapes (or instances)
     # which have assigned "pycStretch" property

     for shape (or instance) with "pycStretch" property:

       # parse "pycStretch" property string for each shape

       for handle in shape.props['pycStretch']:

         (name, stretchType, direction,
          minVal, maxVal, parameter, location,
          userScale, userSnap, key) = parseString(handle)

          # Define function to calculate parameter values;
          # note that additional code will be required to
          # convert parameter values to string values, as
          # well as for multi-valued PyCell parameters.

          userFunc = max(minVal, min(maxVal,
                      parameter + round((userScale *
                      stretchIncrement)/ userSnap) * userSnap))

          # now assign these values to geometric stretch handle
          assignHandleToParameter(parameter,
                                  location,
                                  direction,
                                  stretchType,
                                  userFunc)
```

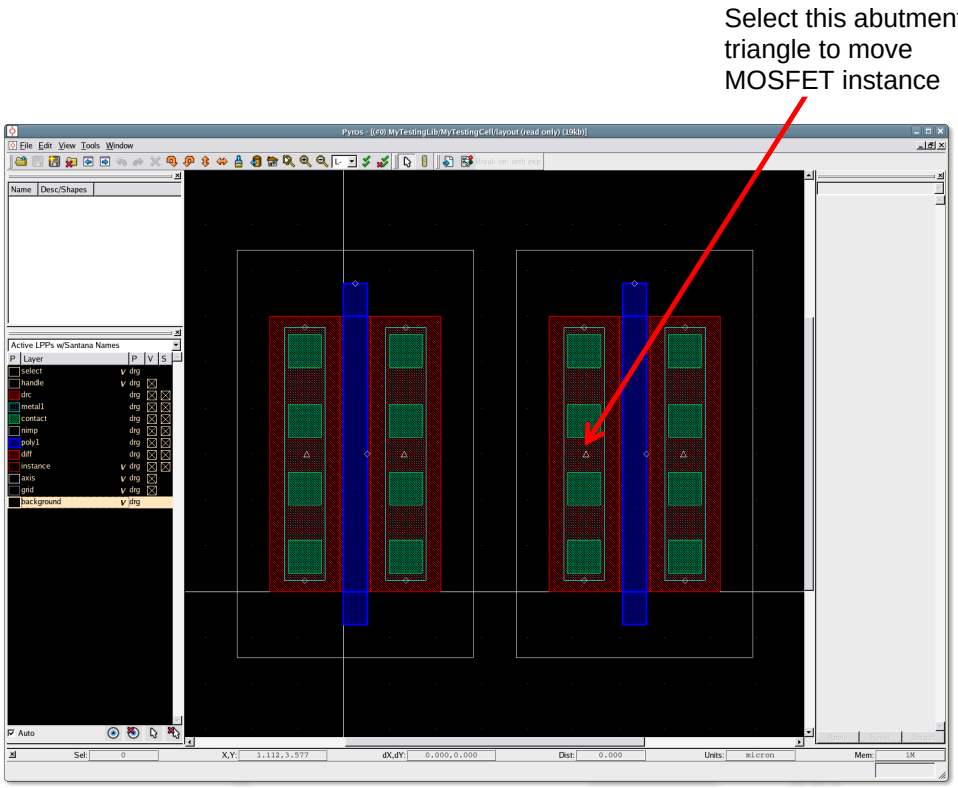
3

Auto-Abutment

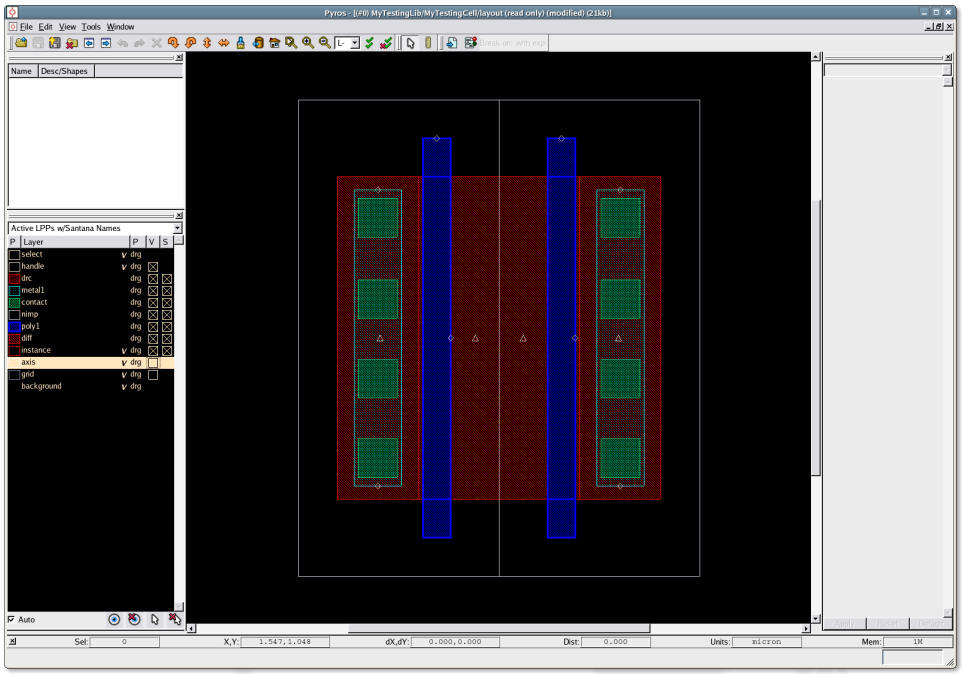
Abutment is the process by which EDA interactive editing tools modify PCell instances and place these instances in order to minimize design layout area. When two such instances are abutted, the layout is modified, so that these instances appear to be merged. For example, if two MOSFET transistor instances are abutted in an interactive editing tool, then the source of one transistor can be merged with the drain of the other transistor, so that the source and drain diffusion can be shared between the two transistor instances.

As an illustration, the following screen shots show the use of auto-abutment in the Pyros layout viewer to merge the source and drain diffusion for two different MOSFET transistor instances:

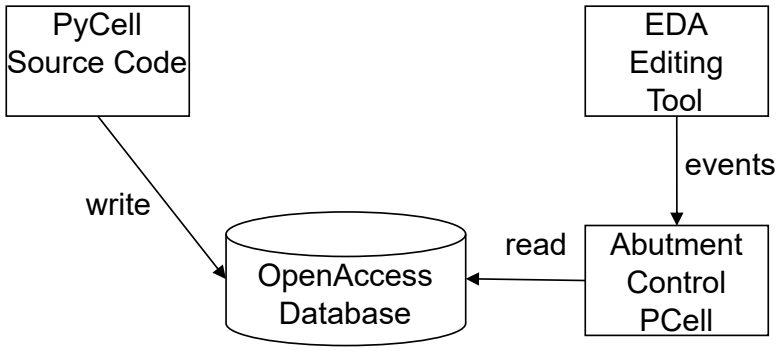
1. Two MOSFET transistor instances with the same dimensions; design rule correct spacing between these two instances:



2. Two MOSFET transistor instances with abutted source and drain diffusion; second instance has been moved to overlap source and drain pin diffusion shapes:



This auto-abutment feature is implemented by having the PyCell developer store abutment information as properties on associated layout shapes in the OpenAccess database. When the EDA tool user performs abutment editing operations, the EDA editing tool generates abutment events to the Abutment Control PCell, which reads the properties stored in the OpenAccess database, to perform the requested abutment operations. This is as illustrated in the following diagram:

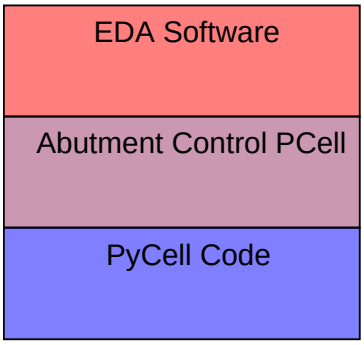


The following is the general sequence of steps which are required by the EDA tool developer to support this auto-abutment feature in their interactive editing environment:

1. Check for abutment situation initiated by EDA tool user. When such abutment situations are detected, then define abutment event for abutment control PCell.
2. Create abutment control PCell, and set parameters as required, using abutment event from step #1. This abutment control PCell will perform requested abutment operations, and appropriately set PyCell instance parameters to properly place PyCell instances.
3. Examine return status value from abutment control PCell, using *pycResult* string property which is stored as property for abutment control PCell. Generate any required error messages for the EDA tool user, using error string stored in *pycResult* property.

In order to handle the interface between the interactive EDA layout editing tool and the PyCcell, has defined an auto-abutment protocol. This abutment protocol allows the PyCell author to provide information about abutment parameters and their values for different abutment situations. This PyCell information is then stored using OpenAccess properties for shapes which can be abutted in the PyCell layout. Note that these properties are only stored on top-level shapes which are defined at the top-level of the PyCell instance.

This abutment protocol is provided by means of an intermediate control layer between the EDA software and the PyCell source code. This control layer is implemented by means of an abutment control PCell. This abutment control PCell does not generate any layout geometries, but instead controls the abutment process for the different instances in the layout. This intermediate control layer can be illustrated as follows:



The EDA tool specifies one of several different predefined abutment events for this abutment control PCell, and it then reads the required abutment information stored as properties on the abutted top-level pin shapes, and then sets the abutment related parameters for the PyCell instances to perform the abutment operation. The EDA software simply needs to create a submaster of this abutment control PCell, and then it can be used to modify abutment parameter values for PyCell instances in the design. Note that a standard abutment control PCell is provided by as part of the PyCell Studio™ download. In addition, EDA tool developers can also write their own abutment control PCells.

Note that the Python PyCell source code for this supplied abutment control PCell is installed in the $CNI_ROOT/pylib/cni/vpcells/abut.py file location. In addition, an example of a PyCell which supports this auto-abutment feature is the Nmos2 cell which is provided as part of the IPL (Interoperable PCell Library) library; this Nmos2 PyCell source code is also installed in the $CNI_ROOT/quickstart/MyPyCellLib_cni130 directory location.

It is assumed that the EDA interactive layout editing tool already has the capability to detect possible abutment situations. These would be situations where the designer has graphically overlapped pin shapes from different PyCell instances. When the EDA interactive editing tool detects such an abutment situation, then it should perform the following tests to determine the parameter values which should be used to create the abutment control PCell:

- Are these overlapping pin shapes connected to the same net in the design?
- Does the *pycAbutClass* property have the same value for these different pin shapes?

After these tests are made, then the EDA interactive editing tool should create the abutment control PCell submaster. Note that OpenAccess allows for the direct creation of submasters, without the need to create an instance in the design. If an instance of this abutment control PCell is created, then it should be deleted, after abutment is completed.

This abutment control PCell will make use of the design and instance names to locate the PyCell instances, and then modify the instance parameters to create valid abutment configurations. If abutment is not valid for the specified instances, then it will calculate the necessary offset distance between the instances.

Although each EDA interactive layout editing tool will be different, the following is the general sequence of steps which should be followed when an abutment situation is detected to support this protocol for auto-abutment:

1. Check to see if overlapping pins are connected to same net in design
2. Check that *pycAbutClass* pin shape property is same for these top-level pin shapes
3. Set up list of parameter values for abutment control PCell
4. Create submaster (or instance) of abutment control PCell
5. Check return status from abutment control PCell using *pycResult* property

The following are the parameter values which will need to be set up for this abutment control PCell for step 3 in the above list; these are the parameter values which should be set when the abutment control PCell submaster (or instance) is created.

- **library** – library name for PyCell being abutted
- **cell** – cell name for PyCell being abutted
- **view** – view name for PyCell being abutted

- **inst1** – name of first PyCell instance; this is the PyCell instance which is moving
- **inst2** – name of second PyCell instance; this is the PyCell instance which is stationary
- **shape1** – name for first instance pin shape; stored as *pycShapeName* pin property value
- **shape2** – name for second instance pin shape; stored as *pycShapeName* pin property value
- **otherPinsOnNet** – number of other connected instance pins on net; values are 1 or 2.
- A value of 1 indicates that there is no need to create a contact to connect to other pin, while a value of 2 indicates that a contact should be created to connect to other pins.
- **abutEvent** – name of abutment event; this string value must be one of `abut`, `unAbut`, `testAbut` or `instAbut`.
- **uniqueString** – a unique string, such as the current date and time; this is used to force submaster evaluation for every abutment event.

The **abutEvent** parameter describes the type of abutment operation which should be performed by the abutment control PCell. The values for this parameter are as follows:

- `abut` – The EDA editing tool has detected a valid abutment operation by the user, and is using the abutment control PCell to change PyCell instance parameters to create valid abutment configurations. The abutment control PCell places the updated PyCell instances; pin shapes are first aligned, and then offset using the spacing distance specified by the *pycAbutRules* property value. If abutment was successfully performed, then *pycResult* will be set to `true`; otherwise, *pycResult* will be set to the appropriate error message.
- `unAbut` – The EDA editing tool has detected an operation to separate currently abutted PyCell instances, and is using the abutment control PCell to restore the prior values of PyCell instance parameters. If this operation was successfully performed, then *pycResult* will be set to `true`; otherwise, *pycResult* will be set to the appropriate error message.
- `testAbut` – The EDA editing tool wants to know whether an instance and shape pair is currently abutted; *pycResult* will be set to `true` if it is abutted and will be set to `false` if it is not abutted; any other return value will be an error.
- `instAbut` – The EDA editing tool wants to know which instances are currently abutted to a specified instance; *pycResult* will be set to `true` and any other return value will be an error. The string property *pycInsts* will be set to a list of PyCell instance names, separated by commas.

Note that different abutment events will require different parameters to be set. The `abut` event requires all parameters to be set, while the `unAbut` event does not require **shape1**,

shape2 and **otherPinsOnNet** to be set. The `testAbut` abutment event does not require **inst2**, **shape2** and **otherPinsOnNet** to be set. The `instAbut` abutment event does not require **shape1**, **inst2**, **shape2** and **otherPinsOnNet** to be set.

In order to more fully illustrate the use of the abutment control PCell by the EDA layout editing tool, several scenarios will be discussed. Assume that there are three PyCell instances (I1, I2 and I3), which are initially abutted in a straight-line fashion (I1 to I2 and I2 to I3). There is also another PyCell instance named I4, which is initially not abutted to any other instances. The following scenarios all use this initial abutment state for all four of these PyCell instances.

- Scenario 1: The user wants to abut I4 to I1, so that I4 is moving and I1 is stationary
  1. EDA tool uses `instAbut` event to find out what I4 is abutted to
  2. Abutment control PCell indicates that I4 is not abutted to anything
  3. EDA tool uses `abut` event to abut I4 to I1
  4. Abutment control PCell performs abutment
- Scenario 2: The user wants to abut I3 to I4, so that I3 is moving and I4 is stationary
  1. EDA tool uses `instAbut` event to find out what I3 is abutted to
  2. Abutment control PCell indicates that I3 is abutted to I2
  3. EDA tool uses `unAbut` event to separate I3 from I2
  4. Abutment control PCell performs unabutment
  5. EDA tool uses `abut` event to abut I3 to I4
  6. Abutment control PCell performs abutment
- Scenario 3: The user wants to abut I2 to I4, so that I2 is moving and I4 is stationary
  1. EDA tool uses `instAbut` event to find out what I2 is abutted to
  2. Abutment control PCell indicates that I2 is abutted to I1 and I3
  3. EDA tool uses `unAbut` event to separate I2 from I1
  4. Abutment control PCell performs unabutment
  5. EDA tool uses `unAbut` event to separate I2 from I3
  6. Abutment control PCell performs unabutment
  7. e) EDA tool uses `abut` event to abut I2 to I4
  8. Abutment control PCell performs abutment

- Scenario 4: The user wants to abut I3 to I1, so that I3 is moving and I1 is stationary
  1. EDA tool uses `instAbut` event to find out what I3 is abutted to
  2. Abutment control PCell indicates that I3 is abutted to I2
  3. EDA tool uses `unAbut` event to separate I3 from I2
  4. Abutment control PCell performs unabutment
  5. EDA tool uses `abut` event to abut I3 to I1
  6. Abutment control PCell performs abutment

Although each EDA layout editing tool is different, the general sequence of steps for performing abutment can be expressed with the following Python-like pseudo-code.

This sample pseudo-code illustrates the steps used to interface with the abutment control PCell for the `abut` event used in the preceding four scenarios. Similar pseudo-code would be used for the other three abutment events handled by the abutment control PCell.

```
# first check for abutment situation

if abutment situation detected:

  if pins not connected to same net:
    return

  if abutClass property not same for each top-level pin shape:
    return

  # set up abutment event type
  abutEvent = "abut"
  # set up parameter list for abutment control PCell

  # get Lib/Cell/View from PyCell instances being abutted

  library = inst1.getLibName()
  cell = inst1.getCellName()
  view = inst1.getViewName()

  # get names for instances and pin shapes
  inst1Name = inst1.getName()
  inst2Name = inst2.getName()
  shape1Name = shape1.props['pycShapeName']
  shape2Name = shape2.props['pycShapeName']

  # put all of these values into parameter list
  paramList = [ ("library", library),
                ("cell", cell),
                ("view", view),
                ("inst1", inst1Name),
                ("inst2", inst2Name),
```

```
                ("shape1", shape1Name),
                ("shape2", shape2Name),
                ("otherPinsOnNet", 1),
                ("abutEvent", abutEvent),
                ("abutManagement", "auto"),
                ("uniqueString", getTime()]

  # now create instance of abutment control PCell
  PCellObj = openCell("cnVPCellLib",
                      "abutControl",
                      "layout")

  PCellInst = createPCellInst(PCellObj, paramList)

  # check return result from abutment control PCell

  if PCellInst.props['pycResult'] != 'true':
    # print error message for user
    print "Error detected - %s" % PCellInst.props['pycResult']

  # delete instance of abutment control PCell
  PCellInst.destroy()

  return

# Can also directly create submaster for abutment control PCell,
# without needing to create instance of this PCell. In this case,
# there is then no need to remember to delete PCell instance.

# this is done by passing parameter array when PCell design is
# opened; this will then cause submaster to be created as needed.

# now directly create submaster for abutment control PCell
PCellObj = openCell("cnVPCellLib",
                    "abutControl",
                    "layout",
                     paramList)
```

Although it is not required for the EDA tool developer to directly access and process the abutment related properties which are stored on top-level pin shapes which may be abutted, the following description is included to described the underlying implementation of the abutment control PCell. This information may be useful, if the EDA tool developer decides to write their own abutment control PCell.

There are a number of predefined properties which are stored on each pin shape which may be abutted in the interactive EDA editing tool. These properties specify all of the information required to perform abutment for PCell instances, as follows:

- *pycShapeName* – string property on each pin shape which can be abutted; specifies a unique name for the shape, and is assigned by the PyCell developer. Note that this string property is only stored on top-level shapes defined at the top-level PCell instance.
- *pycPinSize* – float property on each pin shape which can be abutted; specifies width of the pin shape. This width information is not derived from the pin shape, to avoid any orientation dependent interpretation of width dimensions.
- *pycAbutClass* – string property on each pin shape which can be abutted; used to restrict the valid abutting PyCells. This property can be checked by the interactive EDA editing tool to ensure that two pin shapes from different PyCell instances can be abutted.
- *pycAbutRules* – string property on each pin shape which can be abutted; specifies how the PyCell parameters are modified for different abutment events detected by the interactive EDA editing tool.
- *pycAbutDirections* – string property on each pin shape which can be abutted; specifies valid abutment direction(s) for the abutting PyCell. This is a comma-separated list of one or more abutment direction strings: `top`, `bottom`, `left` or `right`.

The *pycAbutRules* string property is the most complicated of these abutment properties, and is used to encode the abutment information for the PyCell. This string represents a list of lists of association lists, and each association list is composed of three sublists:

- Name of the abutment condition and associated design rule spacing
- PyCell parameter names and values used when PyCell is being moved for abutment
- PyCell parameter names and values used when PyCell is stationary during abutment

The name of the abutment condition is set using the `_name` keyword, and the associated design rule spacing value is set using the `_spacing` keyword. The names for these abutment conditions are predefined, and should be one of the following:

- noAbut – no abutment
- abut2PinBigger – abutment with 2 device connections, bigger pin width
- abut3PinBigger – abutment with 3 device connections, bigger pin width
- abut2PinEqual – abutment with 2 device connections, same pin width
- abut3PinEqual – abutment with 3 device connections, same pin width

- abut2PinSmaller – abutment with 2 device connections, smaller pin width
- abut3PinSmaller – abutment with 3 device connections, smaller pin width

The associated design rule spacing value (using the `_spacing` keyword) for each of these abutment conditions should be a floating point value.

The second and third of these association sublists describe the PyCell parameters and values for the moving PyCell instance and the stationary PyCell instance. Each key-value pair in each of these sublists specifies a PyCell parameter name and its associated value.

This *pycAbutRules* string property associated with the shape in the generated layout is represented using the following string format:

```
_name:val, _spacing:val; param1:val1, param2:val2, param3:val3,
  param4:val4; param5:val5, param6:val6, param7:val7, param8:val8
```

That is, each colon-separated element of the string specifies the name and value pair, which can either be the name or spacing for the abutment information, or one of the PyCell parameter names and values. Note that the information in each of these three different association sublists is separated by a semicolon. In the case where there is more than one abutment condition to be described, then the caret (`^`) character should be used to separate each of these abutment condition strings.

4

Appendix

Note that in the case where the interactive EDA editing tool is the Cadence Virtuoso layout editor, the support for the stretch handle and auto-abutment features is already provided by Synopsys. In particular, the SKILL file cniUtils_encrypted.il (which is installed in the $CNI_ROOT/partners/cds/cniUtils_encrypted.il file location) should be loaded and used by the Cadence Virtuoso layout editor. This file contains the additional SKILL code which is required to implement these features in the Cadence Virtuoso environment; this SKILL code uses the abutment control PCell to provide support for the auto-abutment feature.

All of the required additional auto-abutment properties are automatically set by the **autoAbutment()** function available in the Python API; the *abutFunction* property is set by passing the function name string using the *function* parameter.

The additional properties on layout shapes which are required to be set for the Cadence Virtuoso layout editor are the *abutAccessDir*, *abutClass*, *abutFunction*, *abutDirections*, *vxlInstSpacingDir* and *vxlInstSpacingRule* properties. These additional properties are required to be defined by the PyCell source code. In addition, the *abutFunction* property specifies the name of a SKILL function which should be called when an abutment event is detected by the Virtuoso layout editor. This function (`cniAbut`) is already provided by in the cniUtils_encrypted.il SKILL file used by Virtuoso.