Python API Reference Manual

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

This reference manual documents the Python API (Application Programming Interface) which is used to create parameterized cells within the Santana design environment. This Santana design environment makes use of the popular open-source Python programming language and the OpenAccess EDA database to provide a highly productive design environment for creating parameterized cells for analog layout design purposes. This Python API provides a large number of classes and methods which are specialized for layout designs. By using these Python classes to provide powerful, high-level layout design abstractions, the Santana design environment is extremely productive.

The Python open-source programming language provides a highly productive object-oriented programming environment, which can easily be extended. The Santana system extends these object-oriented programming capabilities by providing class definitions for objects which very closely correspond to objects in physical design layouts. For example, layers, shapes and other geometric objects are easily represented in this extended object-oriented programming environment. This approach provides a very natural correspondence between the geometric objects used in physical design layout and the objects used in object-oriented software design. This natural correspondence provides a productive design environment for creating parameterized cells. Since these parameterized cells are based upon the Python language, they are given the name `PyCells` (PyCell™ for Python parameterized cell).

The OpenAccess database is a popular EDA (Electronic Design Automation) database system which is being used by a number of different EDA tool companies. Instead of using a proprietary database for EDA designs, the use of an open standard for an EDA database allows chip designers to work on the same chip design using tools from a number of different EDA tool vendors. This approach provides the chip designer with an important means of design portability across the set of different EDA tool suites from various EDA tool companies.

This document assumes that the reader has a working knowledge of the Python programming language, as well as a basic understanding of object-oriented programming concepts (such as classes, methods and class inheritance). Note that a great deal of information about Python can be found at the www.python.org web site. This document does not assume any knowledge of the OpenAccess database system. However, in order to use the methods provided in the Python API to access the underlying OpenAccess data objects, the designer should understand the OpenAccess database system. Information about OpenAccess can be found at the www.si2.org web site.

Feedback

Summary Description

The elements of the Python API (Application Programming Interface) can be separated into three general categories: class objects (with methods), global functions and symbolic constant objects (which can be used with both class objects and global functions).

I. The class objects in the Python API can be organized into the following groups:

1. Basic Geometric Classes: Point, Range, Segment, Box, PointList, Orientation, Direction, Transform.
2. Physical Component Shapes: Shape, Arc, Donut, Dot, Ellipse, Line, Path, PathSeg, Polygon, Rect, Text.
3. Physical Component Related Classes: Instance, InstanceArray, VectorInstance, Contact, AbutContact, ArrayInstContact, Grouping, CompoundComponent, Bar, ContactRing, DeviceContact, MultiPath, RoutePath.
4. Physical Component Reference Classes: PhysicalCompRef, InstanceRef, InstanceArrayRef, VectorInstanceRef, ShapeRef, ArcRef, DonutRef, DotRef, EllipseRef, LineRef, PathRef, PathSegRef, PolygonRef, RectRef, TextRef.
5. Connectivity Classes: Net, BusNet, Term, BusTerm, Pin, InstTerm, InstPin, Topology.
6. Technology Related Classes: Grid, Layer, ShapeFilter, Tech.
7. Parameterized Cell Creation Classes: Dlo, DloGen, Lib.
8. Parameterized Cell Parameter Classes: ParamArray, ParamSpecArray, ChoiceConstraint, RangeConstraint, StepConstraint.
9. Utility Classes: Log, Unique, SnapType, ViewType, Prop, PropSet, ulist.

Most all of these class objects have direct constructors which allow easy creation, along with a large number of methods to operate on these class objects, including equality operators. Note that there are a large number of methods defined for the base DloGen and PhysicalComponent classes, upon which layout designs and layout objects are based. In particular, there are layout manipulation methods, such as abut(), alignEdge(), alignLocation(), place(), fgPlace(), move(), and moveTo(), as well as rotate() and mirror() methods which operate on one or more individual design components.

II. The global functions in the Python API can be organized into the following groups:

1. Technology related: conditionalRuleExists(), getGridResolution(), getPhysicalRule(), physicalRuleExists(), getLayer(), getIntermediateLayers(), uu2dbu(), dbu2uu(), uu2dbuArea(), dbu2uuArea().
2. Layout Manipulation functions: abut(), alignEdge(), alignLocation(), alignEdgeToPoint(), alignLocationToPoint(), getBBox(), place(), fgPlace(), fgAbut(), fgAnd(), fgFill(), fgNot(),

fgOr(), fgSize(), fgXor(), moveBy(), moveTo(), moveTowards(), mirrorX(), mirrorY(), rotate90(), rotate180(), rotate270(), setOrigin().

3. Convenience Functions: makeGrouping().
4. Utility Functions: fgDrc().
5. Database Related Functions: save()

Note that many of the global functions in the Layout Manipulation function group have the same name and similar functionality as methods defined for the PhysicalComponent class. These global functions operate on the current design object, and are provided as a convenience for the Python parameterized cell developer. In addition, most of the global functions in the Technology related group have the same name and similar functionality as methods defined for the Tech technology class, and operate on the technology object associated with the current design object. Note that these global functions can be thought of as a convenient short-hand for the underlying class methods.

III. The symbolic constant objects in the Python API can be organized into twelve groups, which are described as follows:

1. Parameter Failure Action values: ACCEPT, REJECT, USE_DEFAULT.

These symbolic constants are used to specify the action which should be taken when a parameter value entered by the user does not meet the range or choice constraint which has been defined for that parameter.

1. Direction values: NONE, NORTH, NORTH_EAST, EAST, SOUTH_EAST, SOUTH, SOUTH_WEST, WEST, NORTH_WEST, NORTH_SOUTH, EAST_WEST, ANY, CENTER. These symbolic constants are used to specify a direction which can be associated with various geometric elements and operations.
2. Location values: LOWER_LEFT, CENTER_LEFT, UPPER_LEFT, LOWER_CENTER, CENTER_CENTER, UPPER_CENTER, LOWER_RIGHT, CENTER_RIGHT, UPPER_RIGHT. These symbolic constants are used to specify a location which can be associated with various geometric elements and operations.
3. Orientation values: R0, R90, R180, R270, MY, MYR90, MX, MXR90. These symbolic constants are used to specify an orientation operation which can be associated with different geometric elements and operations.
4. SnapType values: CEIL, FLOOR, ROUND and TRUNC. These symbolic constants are used to specify the snapping type which should be used when a layout object is snapped to a manufacturing grid point.
5. ViewType values: MASK_LAYOUT, SCHEMATIC, SCHEMATIC_SYMBOL, NETLIST, HIER_DESIGN. These symbolic constants are used to specify the type of design view which should be used when user and database units are being converted; note that different ViewType values may have different scaling and unit conversion factors.

6. SignalType values: SIGNAL, POWER, GROUND, CLOCK, TIEOFF, TIEHI, TIELO, ANALOG, SCAN, RESET. These symbolic constants are used to specify the type of signal to be associated with a Net object.
7. TermType values: INPUT, OUTPUT, INPUT_OUTPUT, SWITCH, JUMPER, UNUSED, TRISTATE. These symbolic constants are used to specify the type of terminal which can be defined for a Term object.
8. PathStyle values: TRUNCATE, EXTEND, ROUND, VARIABLE. These symbolic constants are used to specify the beginning and ending point styles for Path and MultiPath objects.
9. GapStyle values: DISTRIBUTE, PART_DISTRIBUTE, MINIMUM, MIN_CENTER. These symbolic constants are used to specify the spacing used between rectangles in a field of rectangles.
10. MaterialType values: NWELL, PWELL, NDIFF, PDIFF, NIMPLANT, PIMPLANT, POLY, CUT, METAL, CONTACTLESS_METAL, DIFF, RECOGNITION or UNKNOWN. These symbolic constants are used to specify the type of material which can be used when constructing a mask layer for a chip.
11. There are several symbolic constant values which are used with the AttrDisplay attribute display class. These symbolic constants are summarized as follows:

TextDisplay.Format: NAME, NAME_VALUE, VALUE

DloGen.AttrType: CELL_NAME, CELL_TYPE, LAST_SAVED_TIME,

LIB_NAME, VIEW_NAME

Instance.AttrType: CELL_NAME, IS_BOUND, LIB_NAME, NAME,

NUM_BITS, VIEW_NAME

InstTerm.AttrType: NAME

Net.AttrType: IS_EMPTY, IS_GLOBAL, IMPLICIT, NAME,

NUM_BITS, SIG_TYPE

Term.AttrType: HAS_PINS, NAME, NUM_BITS

Note that the only officially supported Python API features are the ones which are described in this document. Other features and capabilities may be present in the current implementation of this Python API, but are not officially supported.

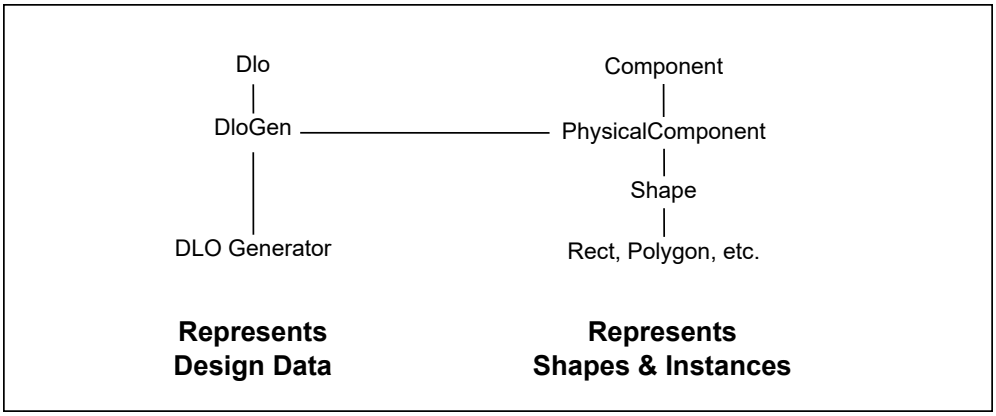
Feedback

Basic Structure

The basic Santana system is built upon a set of base classes, from which the basic design and layout objects are generated through the Python API. These base classes are made accessible through the Python API, but do not have their own creation methods. Instead, other objects which are derived from these base classes can be constructed through the use of the Python API.

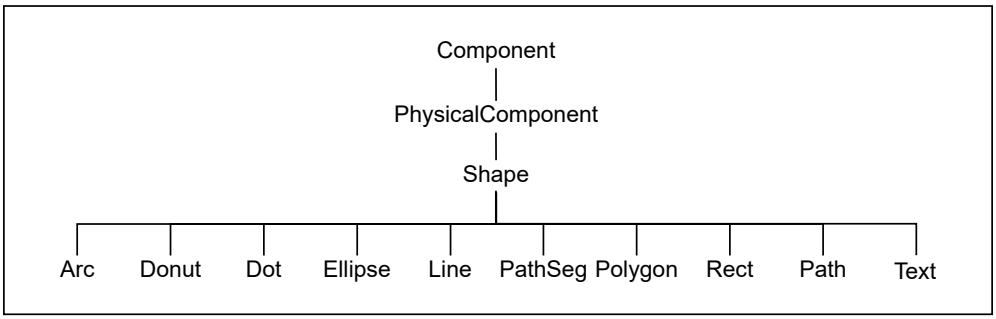
The base class for all components is the Component class, and the base class for all design data is the Dlo class. (The abbreviation `DLO` stands for `Dynamic Layout Object`). These two base classes are then used to construct physical components and generate Dynamic Layout Objects. In particular, the PhysicalComponent class is derived from the base Component class, and the DloGen class is derived from the base Dlo class. The PhysicalComponent class is in turn, the base class for the Shape class, which in turn is the base class for all shapes which are used in physical layout. For example, a rectangle is represented by the Rect class, which is derived from the Shape class, which is derived from the PhysicalComponent class. The DloGen class is the base class for all types of DLO generators. For example, a parameterized transistor layout design could be developed through this Python API, and the resulting DLO generator (parameterized cell) definition would be derived from this DloGen base class.

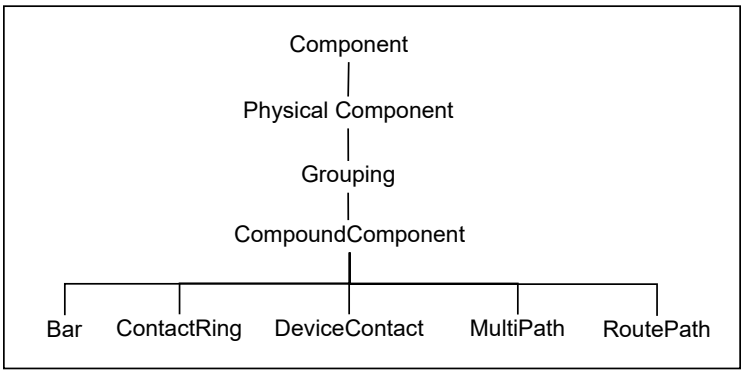
These classes are inter-related, in that the DloGen class is a container class for the Component class. In particular, the Component class object always contains a reference pointer to the DloGen object which owns the particular component object. In addition, the DloGen class object contains a list of all of the PhysicalComponent objects which are contained in the current design object. This can be illustrated using the following general class inheritance diagram:

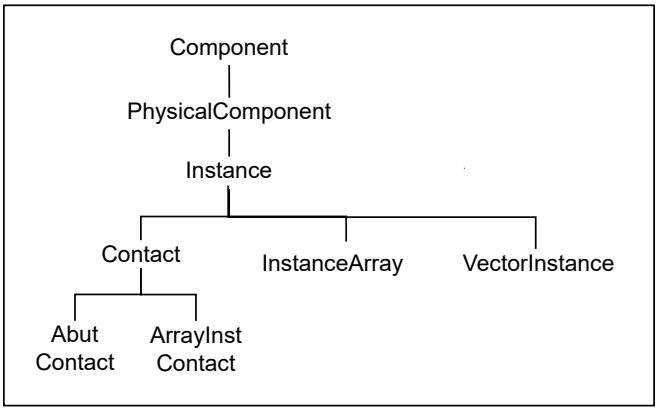


Feedback

In fact, the right hand side of this general class inheritance diagram can be expanded into the complete set of class inheritance diagrams for all of the class objects in the Python API which are derived from this base Component class:







The Python API can be used by the designer to create their own DLO Generator; for example, a library of parameterized cells can be constructed using the Python API. Each of these parameterized cells would be a DLO generator, which would use DloGen as its base class, and would inherit all of the functionality of this DloGen generator base class.

Feedback

Units

There are two different types of units which can be used in physical layout designs, user units and database units. Typical user units used for physical layout designs would be microns, while corresponding database units would be an integral multiple of the user unit chosen, such as 1000 database units per micron user unit. In order to simplify the interface to the Python API, all units used in the Python API are assumed to be specified in terms of user units. For example, the coordinates of a Point object would be specified in terms of user unit values, using the Coord data type in Python.When necessary, there are methods for the DloGen design class and the Tech technology class to convert between user units and database units. Note that the conversion factor which is used to convert between user units and database units is specified within the Santana technology file (as described in the `Technology Related Classes` section).

Python Interpreter

This Python API documentation contains detailed descriptions of the different classes and class methods which are available. As part of this documentation, a large number of Python code examples have been provided, to help illustrate the concepts and proper usage of the different methods contained in each of these classes. In addition to reading these Python code examples, the reader can also directly use these examples as input to the Python interpreter. This Python interpreter (`cnpy`) provides a customized module called `PyCell Explorer`, which can be used to interactively create Python parameterized cell designs. One benefit of using this `PyCell Explorer` module is that it can be directly linked to the Santana viewing tool. Thus, as interactive Python commands are entered by the user and then executed by the Python interpreter, the corresponding physical design objects are displayed in the viewing tool. Please see the Appendix of this document for a detailed description of this `PyCell Explorer` tool.

Python Documentation Conventions

This Python API documentation contains detailed descriptions for the different classes and class methods which are available. Note that in the descriptions of these class methods, the expected types for each parameter are provided. This is unlike standard Python documentation, which does not usually provide such type descriptions (since Python is a dynamically typed language). These type descriptions are provided in this document, in order to make the descriptions of these different parameters for these class methods as descriptive and as clear as possible. The parameter types are displayed using regular fonts, while the parameter names are displayed using italics.

2

BASIC GEOMETRIC CLASSES

There are a number of basic geometric classes which are required to create and manipulate layout objects in a layout design. These basic geometric objects include such familiar objects as points, boxes and point lists. In addition, direction, location, orientation and transform classes are provided to work with these basic geometric objects. These basic geometric classes (and objects) can be summarized as follows:

Point – defines a point in the layout region, defined using x and y coordinate values

Range – defines one-dimensional range of values, defined by pair of coordinate values

Segment – defines a line segment, specified by the two end points

Box – typically used to define a bounding box for a drawn shape in the layout

Direction – class which defines one of several directions, along with methods to manipulate the given direction

Location – class which defines one of several locations

Orientation – class which defines different rotation and mirroring operations, along with methods to manipulate the given orientation

Transform – class which provides two-dimensional transformations, defined by orientation and translation operations

Each of these different basic geometric elements has a number of different methods defined for each of these objects. Note that these geometric objects are typically defined as stand-alone classes, so that they do not have any special class inheritance relationships between them. These geometric objects and their associated methods are described as follows:

Point

Description: This is the basic geometric point class, which defines a single point in the x-y coordinate space which is used to define layout objects. This point object is defined by two values, its x-coordinate value and its y-coordinate value, using the Coord data type in Python.

Note that the standard Python addition and subtraction operators have been overloaded for this Point class. This, the `+` operator can be used to add two points, and the `-` operator can be used to subtract one point from another point. In addition, the combined operators `+=` and `-=` can be used to assign the result of adding or subtracting points.

Creation:

Point(Coord *x*=0, Coord *y*=0) – creates a Point object, using the *x* and *y* coordinate values which are passed as parameter values.

Methods:

areColinearPoints(Point *p1*, Point *p2*, Point *p3*) – returns True if these three points are collinear or coincident, and returns False otherwise. This checking is done using 8-byte integer arithmetic, so that there are no possible round-off errors, as might occur with floating-point arithmetic operations. Note that this method allows these points to lie on any line segments, whereas the "**isBetween**()" method requires points to lie on horizontal or vertical line segments.

Note that this method is defined as a static method, so that it can be called without explicitly creating a Point object. For example, it can be directly called using the syntax "Point.areColinearPoints(*p1*, *p2*, *p3*)".

copy() – returns a copy of this point

getCoord(Direction *dir*) – returns the coordinate value for the specified *dir* direction; the x-coordinate value will be returned for the EAST_WEST direction, and the y-coordinate value will be returned for the NORTH_SOUTH direction. If any other direction is specified, then an exception will be raised (see the description of the `Direction` object later in this section).

getSpacing(Direction *dir*, Point *refPoint*) – returns the relative spacing distance from this Point to a reference point *refPoint*, along the specified direction *dir*. If the specified direction *dir* is not one of the four primary directions (NORTH, SOUTH, EAST, WEST), then an exception is raised.

getX() – returns the x-coordinate value for this point

getY() – returns the y-coordinate value for this point

invalid() – returns a Point object which is the predefined invalid point value in the x-y coordinate space. This is useful when defining possible reference points for other geometric objects (for example, see the description of the Contact class).

Note that this method is defined as a static method, so that it can be called without explicitly creating a Point object. For example, it can be directly called using the syntax `Point.invalid()`.

Feedback

isBetween(Point *a*, Point *b*) – returns True if this point lies between these two other points *a* and *b* on either a horizontal or a vertical line segment, and returns False otherwise. This method can be useful for routing purposes, where rectilinear line segments are required.

isValid() – returns True if this point represents a valid point in the x-y coordinate space, and returns False otherwise.

place(Direction *dir*, Point *refPoint*, Coord *distance*, bool *align*=True) – places this Point relative to a reference point *refPoint*, along the specified direction *dir*. If the optional *align* parameter is True, then this Point will first be aligned with the *refPoint* reference point in the direction orthogonal to the specified direction *dir*. If the specified direction *dir* is not one of the four primary directions (NORTH, SOUTH, EAST, WEST), then an exception is raised.

set(Point *p*)

set(Coord *_x*, Coord *_y*) – sets the point value (or the coordinate value) for this point

setCoord(Direction *dir*, Coord *coord*) – sets the coordinate value for the *dir* direction; the x-coordinate value will be set for the EAST_WEST direction, and the y-coordinate value will be set for the NORTH_SOUTH direction. If any other direction is specified, then an exception will be raised (see the description of the `Direction` object later in this section).

setX(Coord *x*) – sets the x-coordinate value for this point

setY(Coord *y*) – sets the y-coordinate value for this point

snap(Grid *grid*, SnapType *snapType*=None) – snaps this point to the nearest grid point. The *grid* parameter is used to specify the Grid object which should be used to determine the nearest grid point value, while the *snapType* parameter specifies the rounding method which should be used for this snapping operation (see the `Utility Classes` section).

snapX(Grid *grid*, SnapType *snapType*=None) – snaps the x-coordinate of this point to the nearest grid point. The *grid* parameter is used to specify the Grid object which should be used to determine the nearest grid point value, while the *snapType* parameter specifies the rounding method which should be used (see the `Utility Classes` section).

snapY(Grid *grid*, SnapType *snapType*=None) – snaps the y-coordinate of this point to the nearest grid point. The *grid* parameter is used to specify the Grid object which should be used to determine the nearest grid point value, while the *snapType* parameter specifies the rounding method which should be used (see the `Utility Classes` section).

snapTowards(Grid *grid*, Direction *dir*) – snaps this point to the nearest grid point in the specified direction *dir*. The *grid* parameter is used to specify the Grid object which should be used to determine the nearest grid point value. If the specified direction *dir* is not a half-line direction, then an exception will be raised (see the description of the `Direction` object later in this section).

Feedback

toDiagAxes() – returns a Point object which transforms this point specified in orthogonal coordinate axis values to a point in diagonal axes. Diagonal axes are the 45-degree and 135-degree axes. Note that diagonal distances and diagonal coordinate values for these diagonal axes are scaled by the square root of two, in order to retain integer accuracy.

toOrthogAxes() – returns a Point object which transforms this point specified in diagonal coordinate axis values to a point in the usual orthogonal axes. Diagonal axes are the 45-degree and 135-degree axes. Note that diagonal distances and diagonal coordinate values for diagonal axes are scaled by the square root of two, in order to retain integer accuracy.

transform(Transform *trans*) – applies the `transform` *trans* passed as a parameter to this point object (see the description of the `Transform` object later in this section).

Attributes:

In addition to these methods for the Point class, there are also two attributes (or properties) defined for this Point class, described as follows:

x – the value of the x-coordinate for this point

y – the value of the y-coordinate for this point

Note that these **x** and **y** attribute values will return the same values as returned by the "**getX**()" and "**getY**()" methods. In addition, these attributes are settable, so that they can be used to set these values in the same manner as the "**setX**()" and "**setY**()" methods.

Examples:

```
point1 = Point(10.0, 20.0)
point2 = Point(10.0, 40.0)
point3 = Point(10.0, 80.0)
point1.getX()                          # returns 10.0
point1.x                               # also returns 10.0
point1.getY()                          # returns 20.0
point1.y                               # also returns 20.0
point2.isBetween(point1, point3)       # returns True
point2.setX(30.0)
point3.isBetween(point1, point2)       # returns False
point1.isValid()                       # returns True
point4 = point1.INVALID                # create invalid point
point4.isValid()                       # returns False
# check to see if these three points are collinear points
point.areColinearPoints(point1, point2, point3) # returns False
point5 = point3.copy()                 # copy point
point5.isValid()                       # returns True
point5.place(WEST, point3, 5.0)        # place point5 WEST of point3
point5.getSpacing(WEST, point3)        # returns 5.0 spacing value
# perform addition and subtraction operations
point1 = Point(1,2)
point2 = Point(3,4)
point1 + point2                        # returns Point(4,6)
```

Feedback

```
point2 - point1                     # returns Point(2,2)
point1 += Point(3,4)                # returns Point(4,6)
point1                              # point1 is now Point(4,6)
# Convert point to diagonal axes (from orthogonal axes);
# note diagonal coordinate values are scaled by sqrt(2)
point6 = Point(1,1)
point6 = point6.toDiagAxes()
point6                              # point6 is Point(2,0)
# convert point back to orthogonal axes
point6 = point6.toOrthogAxes()
point6                              # point6 is Point(1,1)
```

Range

Description: This is the basic geometric range class which defines a one-dimensional range of values in the x-y coordinate space which is used to define layout objects. This range is defined by the two coordinate values for the range, the left and right values for the range. Note that although these two left and right coordinate values are specified in user units, they are actually stored in database units.

Creation:

There are two different sets of parameters which can be used to create this Range object: 1) the individual coordinate values for the left and right coordinates of the Range and 2) another previously defined Range object. These two different creation calls have the following parameter signatures:

Range(Coord *left*, Coord *right*)

Range(Range *r*)

In the case where no parameter values are specified, then an inverted Range is created, using the value INT_MAX for the left coordinate value and INT_MIN for the right coordinate value. Note that INT_MAX and INT_MIN are symbolic constant names for the maximum and minimum allowable integer values; these names correspond to the Python sys.maxint and –sys.maxint –1 constants.

Methods:

alignEdge(Direction *dir*, Range *refRange*, Direction *refDir*=None, Coord *offset*=None) – moves this range to align in the X-direction (EAST, WEST or EAST_WEST) with a reference range specified by the *refRange* parameter. The alignment directions are specified by the *dir* and *refDir* direction parameters. If either of these directions is not an X-direction, then an exception is raised. If specified, the *offset* value will be added to the distance which is required to align these two ranges.

Feedback

alignEdgeToCoord(Direction *dir*, Coord *coord*) – moves this range to align in the X-direction (EAST, WEST or EAST_WEST) with the specified *coord* parameter. If the *dir* direction is not an X-direction, then an exception is raised.

Note that both of these alignment methods align any center edges in a deterministic fashion; when two solutions are possible, the one where this range is greater than the reference range or coordinate value will always be chosen.

compareTrueCenter() – checks to see if the mathematically defined midpoint (or center) value for this range as represented a real number is the same as the midpoint value that is calculated using the stored integer database units. Note that due to integer truncation, the integer center value can possibly be different from the actual center value. A return value of zero indicates that these two representations provide the same center value, while a non-zero value indicates that they provide different center values. If the return value is greater than zero, the real number center is larger than the integer center; if this return value is less than zero the real number center is smaller than the integer center.

contains(Range *range*, bool *incEnds*=True) – returns True if this range contains the specified *range*. If the *incEnds* Boolean flag is set, then the *range* will be considered to be contained inside this range, if the left and right coordinate values of the *range* parameter are values within this range.

containsCoord(Coord *coord*, bool *incEnds*=True) – returns True if this range contains the specified *coord* coordinate value. If the *incEnds* Boolean flag is set, then *coord* will be considered to be contained inside this range, if has the same value as either the left or right coordinate value for this range.

expand(Coord *coord*) – expands this range by the *coord* coordinate value in both directions; this *coord* coordinate value is added to the right coordinate and is subtracted from the left coordinate of this range.

expandDir(Direction *dir*, Coord *coord*) – expands this range by the *coord* coordinate value in the *dir* direction; if this *dir* direction is not one of the directions defined by the X-axis (EAST, WEST or EAST_WEST), then an exception will be generated.

findAlignEdgeOffset(int *moveEdgeLean*, int *refEdgeLean*) – adjusts edge alignment between two ranges. The *moveEdgeLean* and *refEdgeLean* parameters are set by calling the **compareTrueCenter**() method for the range being aligned and the reference range. Based upon these two parameter values, this method adjusts an edge for alignment; if a center edge is used, this method considers the value of the true real number center.

Note that this method is defined as a static method, so that it can be called without explicitly creating a Range object. For example, it can be directly called using the syntax “Range. findAlignEdgeOffset (*moveEdgeLean*, *refEdgeLean*)”.

fitSubranges(Coord *width*, Coord *space*, Coord *gridSize*, GapStyle *gapStyle*) – determines the number of equally sized subranges which can fit within this range. The *width* parameter specifies the size of each subrange, while the *space* parameter specifies

the spacing which should be used between these subranges. In addition, the *gapStyle* parameter specifies the distribution of the spacing for these subranges within this range (as one of MINIMUM, DISTRIBUTE, MIN_CENTER. The *gridSize* parameter specifies the grid size for each subrange. The return value for this method is a 4-tuple of values: (*counts*, *beginStep*, *step*, *extraGrids*), where *counts* is the number of subranges which will fit, *beginStep* is the distance from the left point of this range for the first subrange, and *step* is the distance between two consecutive subranges. In the case where the *gapStyle* parameter is set to GapStyle.DISTRIBUTE, *extraGrids* returns the extra space which should be distributed among these subranges. Note that these *beginStep* and *step* return values will be negative for an inverted Range.

fix() – checks to see if this range is inverted; if it is inverted, then it will swap left and right coordinates, so that the range will not be inverted, and returns the resulting range.

getCenter() – returns the midpoint of this range, calculated by adding the left and right coordinate values and dividing by two.

getCoord(Direction *dir*) – returns the left or right coordinate value for this range, based upon the *dir* direction. If the direction is WEST, then the left coordinate value is returned; if the direction is EAST, then the right coordinate value is returned. If the direction is EAST_WEST, then the center coordinate for this range is returned; any other direction will cause an exception to be raised.

getLeft() – returns the left coordinate value for this range

getRight() – returns the right coordinate value for this range

getWidth() – returns the width of this range, calculated as the left coordinate value subtracted from the right coordinate value. Note that this calculated width can be negative for an inverted range, and may overflow, if it can not be represented in database units.

hasNoWidth() – returns True if this range has zero width, and returns False otherwise. This will be the case if the left and right coordinates for this range have the same value.

init() – sets this range to the default inverted Range by setting the left coordinate to the maximum positive value (INT_MAX), and the right coordinate set to the maximum negative value (INT_MIN). This inverted range is returned as the result of this method.

intersect(Range *range,* Direction *dir*=None*)* – returns the intersection of this range and the passed parameter *range*. If these two ranges do not intersect, then an inverted range representing the range between these two ranges will be returned. If the *dir* direction is specified, then it specifies which coordinate values should be set; if the direction is WEST, then the left value is changed; if the direction is EAST, then the right value is changed. If the direction is EAST_WEST, then both coordinate values are changed. Any other direction will cause an exception to be raised. If the direction is not specified, then both coordinate values are changed.

Feedback

isInverted() – returns True if this range has negative width. That is, this method returns True if the right coordinate value for this range is less than the left coordinate value, and False otherwise.

isNormal() – returns True if this range has positive width. That is, this method returns True if the right coordinate value for this range is greater than the left coordinate value, and False otherwise.

limit(Coord *coord*) – returns the coordinate which is found by limiting the specified *coord* to be within this range. If the specified *coord* is less than the left coordinate of this range, or greater than the right coordinate of this range, then the left or right coordinate value for this range will be returned; otherwise, the *coord* value will be returned.

merge(Range *range*, Direction *dir*=None) – this range is merged with the passed *range* parameter value, and is returned as the value of this method. Note that the merged range is the overall union of these two range objects, determined using the maximum of the two range dimensions. If the *dir* direction is specified, then it specifies which coordinate values should be set; if the direction is WEST, then the left value is changed; if the direction is EAST, then the right value is changed. If the direction is EAST_WEST, then both coordinate values are changed. Any other direction will cause an exception to be raised. If the direction is not specified, then both coordinate values are changed.

mergeCoord(Coord *coord*) – this range is merged with the passed *coord* parameter.

This is done by using the *coord* coordinate value to expand this range, until this coordinate defines either the left or right coordinate value for this range, which is then returned as the merged range. If this *coord* corodinate value lies within this range, then this range is returned as the merged range.

moveBy(Coord *coord*) – moves this range by the specified *coord* offset coordinate value. The *coord* coordinate value is added to the left and right coordinate values for this range.

overlaps(Range *range*, bool *incEnds*=True) – returns True if this range has any overlap with the passed *range*. If the *incEnds* Boolean parameter is True, then the ranges will be considered to overlap, when the intersection between the two ranges is only the left or right coordinate value.

removeRegion(Range *range*) – removes the region of this range specified by the *range* parameter. This *range* parameter must be a sub-range of this range, with one left or right coordinate which matches a left or right coordinate of this range. If this *range* parameter is not such a sub-range of this range, then an exception is raised.

set(Range *r*, Direction *dir*=None)

set(Coord *left*, Coord *right*) – sets both the left and right coordinate values for this range, using the coordinate values specified by either the *r* Range parameter value or the *left* and *right* coordinate parameter values. If the *dir* direction is specified, then it specifies which coordinate values should be set; if the direction is WEST, then the left value is changed;

Feedback

if the direction is EAST, then the right value is changed. If the direction is EAST_WEST, then both coordinate values are changed. Any other direction will cause an exception to be raised. If the direction is not specified, then both coordinate values are changed.

setCenter(Coord *coord*) – sets the midpoint value for this range

setCoord(Direction *dir*, Coord *coord*)– sets the left or right coordinate value for this range, based upon the specified *dir* direction. If the direction is WEST, then the left coordinate value is set; if the direction is EAST, then the right coordinate value is set. Any other direction will cause an exception to be raised.

setDimension(Coord *coord*, Direction *dir*=None) – sets the width for this range, using the *dir* direction to determine which coordinate values should be changed; if the direction is WEST, then the left value is changed; if the direction is EAST, then the right value is changed. If the direction is EAST_WEST, then both the left and right values are changed. Any other direction will cause an exception to be raised. If the direction is not specified, then both coordinate values are changed.

setLeft(Coord *v*) – sets the left coordinate value for this range

setRight(Coord *v*) – sets the right coordinate value for this range

setWidth(Coord *width*) – sets the width for this range; note that the center (or midpoint) for this range will be the same, after this method is completed.

Attributes:

In addition to these methods for the Range class, there are also two attributes (or properties) defined for this Range class, described as follows:

left – the value of the left coordinate for this range

right – the value of the right coordinate for this range

Note that these **left** and **right** attribute values will return the same values as returned by the "**getLeft**()" and "**getRight**()" methods. In addition, these attributes are settable, so that they can be used to set these values in the same manner as the "**setLeft(**)" and "**setRight**()" methods.**Examples:**

```
range1 = Range(-10.0, 10.0)
range2 = Range(0, 5.0)
range1.getLeft()                    # returns -10.0
range1.getRight()                   # returns 10.0
range2.getLeft()                    # returns 0
range2.getRight()                   # returns 5.0
range1.getCoord(WEST)               # returns -10.0
range1.getCoord(EAST)               # returns 10.0
range1.getCoord(EAST_WEST)          # returns 0
range1.getCenter()                  # returns 0
range2.getCenter()                  # returns 2.5
range1.getWidth()                   # returns 20.0
```

Feedback

```
range2.getWidth()                   # returns 5.0
range1.overlaps(range2)             # returns True
range2.overlaps(range1)             # returns True
range1.contains(range2)             # returns True
range2.contains(range1)             # returns False
range1.containsCoord(5.0)           # returns True
range2.containsCoord(5.0)           # returns True
range2.containsCoord(5.0, False)    # returns False
range1.isInverted()                 # returns False
range1.isNormal()                   # returns True
range1.hasNoWidth()                 # returns False
range1.merge(range2)                # returns range1
range1.intersect(range2)            # returns range2
# Generate subranges which fit inside this range;
# generates 3 subranges, with step size of 1.5
(counts, beginStep, step, extraGrids) =
range2.fitSubranges(1.0, 0.5, 0.1, GapStyle.MINIMUM)
```

Segment

Description: This is the basic geometric segment class, which defines a line segment in the x-y coordinate space which is used to define layout objects. This segment is defined by the two end point values for the line segment, the head point and the tail point.

Creation:

There are two different sets of parameters which can be used to create this Segment object: 1) the individual Point values for the head and tail points of the Segment and 2) another previously defined Segment object. In the case where no parameter values are specified, then an empty Segment is created, using the origin point (0,0) for both the head point and the tail point of the segment.

These two different creation calls have the following parameter signatures:

Segment(Point *head*, Point *tail*)

Segment(Segment *s*)

Methods:

addOffsets(Coord *begin*=0, Coord *end*=0) – modifies this segment, by adding the *begin* offset value to the head point and adding the *end* offset value to the tail point for this line segment. If the offset value is positive, then the head or tail point will be extended; if the offset value is negative, then the head or tail point will be retracted. If the *begin* or *end* coordinate value is non-zero and the head and tail points for this segment are coincident, then an exception is raised.

Feedback

contains(Segment *segment*, bool *incEnds* =True) – returns True, if the specified *segment* is included in this line segment. If the *incEnds* parameter is True, then the end points of this segment will be included; otherwise, the head and tail points will not be considered.

containsPoint(Point *point*, bool *incEnds*=True) – returns True, if the specified *point* is included in this line segment. If the *incEnds* parameter is True, then the end points of this segment will be included; otherwise, the head point and tail point will not be considered.

extrapIntersect(Segment *segment*) – returns the intersection of this line segment and the passed parameter *segment*, by extrapolating each of these line segments into complete lines. A point will be returned, if the two extrapolated lines are intersecting and non-parallel, while a segment will be returned, if the two extrapolated lines are collinear. If the two extrapolated lines are parallel and non-intersecting, then None will be returned.

genJustifySegment(Direction *justify*, Coord *sep*) – returns a justified segment which is constructed from this line segment. The *justify* parameter specifies the direction of the justified segment relative to this segment, and the *sep* parameter specifies the separation between the justified segment and this segment. If the head and tail points for this segment are coincident, then an exception is raised. In addition, an exception will also be raised if the *justify* direction is not one of the directions defined by the X-axis (EAST, WEST or EAST_WEST).

getDeltaX() – returns the difference in the X-coordinate values between the tail end point and the head end point for this line segment. This value represents the X-coordinate component of the slope for this line segment (sometimes called the "run").

getDeltaY() – returns the difference in the Y-coordinate values between the tail end point and the head end point for this line segment. This value represents the Y-coordinate component of the slope for this line segment (sometimes called the "rise").

getDir() – returns the Direction value associated with this line segment; this is the direction generated by moving from the head point to the tail point. If this line segment is orthogonal, then the returned direction will be one of the four primary directions (NORTH, SOUTH, EAST, WEST). Otherwise, the returned direction will be one of the four compound directions (NORTH_EAST, NORTH_WEST, SOUTH_EAST, SOUTH_WEST). In the case where this line segment is empty (the head end point and the tail end point are the same), then the NONE direction value will be returned (see the description of the Direction class).

getHead() – returns the head end point value for this segment

getPosition(double *position*) – returns the Point on this line segment determined by the value of the *position* parameter. If the value of this *position* parameter is zero, then the head point will be returned, while if the value is one, then the tail point will be returned. The returned point is calculated using the formula: head + position * (tail – head). Note that the returned point value is rounded to the nearest database unit, and can possibly be located outside the range of this segment. This will be the case, if the value of the *position* parameter is outside the range of values between zero and one.

getTail() – returns the tail end point value for this segment

hasIntersection(Segment *segment*, bool *incEnds*=True, bool *incParallel*=True) – returns True if this line segment has intersection with the specified *segment* parameter, and returns False otherwise. If the *incEnds* parameter is True, then intersection at an end point of the segment is considered to be a valid intersection. If the *incParallel* parameter is True, then parallel intersection is considered to be a valid intersection.

intersect(Segment *segment*) – returns the intersection of this line segment and the passed parameter *segment*. A point will be returned, if these two segments only intersect in a single point, while a segment will be returned, if these two segments are collinear. If these two segments do not intersect, then None will be returned.

isCoincident() – returns True if the head and tail points for this line segment are coincident, and returns False otherwise.

isHorizontal() – returns True if this line segment is horizontal, and False otherwise.

isOrthogonal() – returns True if this line segment is orthogonal, and returns False otherwise. Note that this segment will be considered to be orthogonal, if it is either horizontal or vertical.

isParallel(Segment *segment*) – returns True if this line segment is parallel to the specified *segment* line segment, and returns False otherwise. Recall that two line segments will be parallel, if they each have the same slope.

isVertical() – returns True if this line segment is vertical, and returns False otherwise.

moveBy(Coord *dx*, Coord *dy*) – moves this line segment by the specified *dx* and *dy* offset coordinate values. The *dx* value is added to the head and tail x-coordinate values, while the *dy* value is added to the head and tail y-coordinate values for this line segment.

reverse() – reverses this segment, by interchanging the head end point value and the tail end point value.

set(Point *head*, Point *tail*) – sets both the *head* end point value and the *tail* end point value for this segment.

setHead(Point *head*) – sets the *head* end point value for this segment

setTail(Point *tail*) – sets the *tail* end point value for this segment

transform(Transform *trans*) – applies the `transform` *trans* passed as a parameter to the head point and tail point of this line segment (see the description of the `Transform` object later in this section).

Feedback

Attributes:

In addition to these methods for the Segment class, there are also two attributes (or properties) defined for this Segment class, described as follows:

head – the value of the head end point for this segment

tail – the value of the tail end point for this segment

Note that these **head** and **tail** attribute values will return the same values as returned by the "**getHead**()" and "**getTail**()" methods. In addition, these attributes are settable, so that they can be used to set these values in the same manner as the "**setHead**()" and "**setTail**()" methods.

Examples:

```
seg1 = Segment(Point(0,0), Point(1,0))
seg2 = Segment(Point(0,0), Point(1,1))
seg3 = Segment(Point(0,0), Point(0,1))
seg1.getHead()                      # returns Point(0,0)
seg1.getTail()                      # returns Point(1,0)
seg1.getDeltaX()                    # returns 1
seg1.getDeltaY()                    # returns 0
seg1.isHorizontal()                 # returns True
seg1.isVertical()                   # returns False
seg2.isHorizontal()                 # returns False
seg2.isVertical()                   # returns False
seg3.isHorizontal()                 # returns False
seg3.isVertical()                   # returns True
seg1.isOrthogonal()                 # returns True
seg2.isOrthogonal()                 # returns False
seg3.isOrthogonal()                 # returns True
seg1.getDir()                       # returns Direction.EAST
seg2.getDir()                       # returns Direction.NORTH_EAST
seg3.getDir()                       # returns Direction.NORTH
seg1.isParallel(seg2)               # returns False
seg1.isParallel(seg3)               # returns False
seg1.isCoincident()                 # returns False
seg2.isCoincident()                 # returns False
seg3.isCoincident()                 # returns False
# check for intersection points between these segments
seg1.hasIntersection(seg2, incEnds=True)     # returns False
seg1.hasIntersection(seg2, incEnds=False)    # returns True
seg1.intersect(seg2)                # returns Point(0,0)
seg1.intersect(seg3)                # returns Point(0,0)
# check for position points on and outside segment1
seg1.getPosition(0.0)               # returns Point(0,0)
seg1.getPosition(0.5)               # returns Point(0.5,0)
seg1.getPosition(1.0)               # returns Point(1,0)
seg1.getPosition(2.0)               # returns Point(2,0)
# check to see if various points are contained by these segments
seg1.containsPoint(Point(0,0))      # returns True
seg2.containsPoint(Point(1,1))      # returns True
```

Feedback

```
seg3.containsPoint(Point(1,2))      # returns False
# end points can be ignored for the "contains()" method
seg1.containsPoint(Point(0,0), False)    # returns False
seg2.containsPoint(Point(1,1), False)    # returns False
# check to see if segments are contained by other segments
seg1.contains(seg2)                 # returns False
seg1.setTail(Point(2,2))
seg1.contains(seg2)                 # returns True
# modify segment3 head and tail points using offsets
 seg3.addOffsets(begin=1, end=3)

# generate justified segment from segment3
seg3.genJustifySegment(justify=EAST, sep=1.0)
# move segment3 in both the X and Y directions
seg3.moveBy(1,1)
```

Box

Description: This is the basic geometric box object, which is used to define the bounding box which is used to define basic shapes for layout design objects. For example, the bounding box for a rectangle object can be used to define the basic shape and properties of the rectangle shape. In a similar manner, the bounding box for an ellipse defines the basic properties for the ellipse shape. Note that a box is said to be inverted, if either the left coordinate value is greater than the right coordinate value, or if the bottom coordinate value is greater than the top coordinate value. An inverted bounding box is used to indicate that there are no valid bounding coordinates for any possible bounding box.

Creation:

There are three different sets of parameters which can be used to create this Box object: 1) all 4 individual coordinate values for the left, bottom, right and top of the Box, 2) the lower left and upper right points for the Box, and 3) another previously defined Box object. In the case where no parameter values are specified, then a large inverted Box will be created, using default coordinate values. For example, a box with the lower left corner at (0, 0) and the upper right corner at (10.0, 10.0) can be defined using either of the forms `b1 = Box(Point(0, 0), Point(10.0, 10.0))` or `b1 = Box(0, 0, 10.0, 10.0).`

These three different creation calls have the following parameter signatures:

Box(Coord *left*=INT_MAX, Coord *bottom*=INT_MAX, Coord *right*=INT_MIN, Coord *top*=INT_MIN)

Box(Point *lowerLeft*, Point *upperRight*)

Box(Box *b*)

Note that INT_MAX and INT_MIN are symbolic constant names for the maximum and minimum allowable integer values; these names would correspond to the Python sys.maxint and –sys.maxint –1 constants.

Feedback

Methods:

abut(Direction *dir*, Box *refBox*, bool *align*=True) – abuts edge of this box to opposite edge of the *refBox* box, so that edge of this box just touches the edge of the *refBox* box. The *dir* direction specifies which edge of this box should be abutted to the *refBox* box.

If the *align* parameter is True, then this box is first aligned with *refBox* in the direction orthogonal to the *dir* direction; if this *align* parameter is False, then no alignment will be performed. If *dir* direction is not one of the four primary directions (NORTH, SOUTH, EAST or WEST), then an exception is raised.

alignEdge(Direction *dir*, Box *refBox*, Direction *refDir*=None, Point *offset*=None) – aligns an edge of this box with an edge of the specified *refBox* reference box, where the edges are specified by the *dir* and *refDir* direction parameters. These edges are specified by either horizontal or vertical directions, while center edges are specified by the NORTH_SOUTH or EAST_WEST directions. If the *refDir* parameter is not specified, then the *dir* direction parameter value will be used for the *refDir* parameter. If specified, the *offset* value will be used as an additional offset to move this box after alignment.

This method performs one-dimensional alignment. Note that both of these directions should be either horizontal or vertical; otherwise, an exception is raised. If either of these directions is not one-dimensional, then an exception will also be raised.

alignEdgeToCoord(Direction *dir*, Coord *coord*) – aligns an edge of this box to the specified coordinate, in the specified *dir* direction. This specified direction should be either an X direction or a Y direction; otherwise, an exception is raised. This method performs the most basic one-dimensional alignment.

alignEdgeToPoint(Direction *dir*, Point *point*) – aligns an edge of this box to the specified point, in the specified *dir* direction. This specified direction should be either an X direction or a Y direction; otherwise, an exception is raised. This method performs one-dimensional alignment.

alignLocation(Location *loc*, Box *refBox*, Location *refLoc*=None, Point *offset*=None) – aligns the specified *loc* location point for this box with the *refLoc* location point for the *refBox* reference box. If the *refLoc* parameter is not specified, then the *loc* location point will be used as the value of this parameter. This box will be moved, so that the *loc* location point coincides with the specified *refLoc* location point for the reference box.

If specified, the *offset* value will be used as an additional offset to move this box after alignment. Note that either of these location parameter values can be any of the nine possible location points for these boxes (see section on the Location class). This method performs two-dimensional alignment.

alignLocationToPoint(Location *loc*, Point *pt*) – aligns the specified *loc* location point for this box with the *pt* point. This box will be moved, so that this *loc* location point coincides with the specified point *pt*. This method performs two-dimensional alignment.

Feedback

Note that all of these align methods are deterministic; they will all choose the same alignment value when any center edges or locations are used. Recall that there is always a possibility of truncation error when using center points which lie on odd multiples of grid points; these methods always ensure that the same value is consistently used.

centerCenter() – returns the center center Point object for this box

centerLeft() – returns the center left Point object for this box

centerRight() – returns the center right Point object for this box

contains(Box *box*, bool *incEdges*=True) – returns True if this box contains the specified Box *box*. If the *incEdges* Boolean flag is set, then the *box* will be considered to be contained inside this box, if it touches any of the edges of this box.

containsPoint(Point *p*, bool *incEdges*=True) – returns True if this box contains the specified point *p*. If the *incEdges* Boolean flag is set, then this point will be considered to be contained inside this box, if it touches any of the edges of this box.

expand(Coord *coord*) – expands this box by the *coord* coordinate value in each direction

expandDir(Direction *dir*, Coord *coord*) – expands this box by the *coord* coordinate value in the specified *dir* direction. Note that the *dir* direction can be any direction; the components of this direction will be used to determine which edges should be expanded.

expandForMinArea(Direction *dir*, AreaType *minArea*, Grid *grid*=None) – expands this box in the specified direction *dir*, so that the area of this Box is at least the value of the *minArea* parameter. The *dir* direction can be any direction; the components of this direction will be used to determine which edges should be expanded. If the direction contains a NORTH or SOUTH component, the height will be expanded; if the direction contains an EAST or WEST component, the width will be expanded. If two opposite edges are specified, then both edges will be expanded. If the *grid* parameter is used, this box will be further expanded in the specified *dir* direction to ensure that it lies on a grid point. This method can be used to check that a Box meets minimum area design rules.

expandForMinWidth(Direction *dir*, Coord *minWidth*, Grid *grid*=None) – expands this box in the specified direction *dir*, so that the height or width is at least as large as the value of the *minWidth* parameter. The *dir* direction can be any direction; the components of this direction will be used to determine which edges should be expanded. If the direction contains a NORTH or SOUTH component, the height will be increased; if the direction contains an EAST or WEST component, the width will be increased. If two opposite edges (such as EAST_WEST) are specified, then both edges will be expanded. If the *grid* parameter is used, then this box will be further expanded in the specified *dir* direction to ensure that it lies on a grid point. This method can be used to check that a Box meets minimum width design rules.

expandToGrid(Grid *grid*, Direction *dir*=None) – expands the edges of this box in the specified direction *dir* to align with the nearest grid point. The *grid* parameter is used to specify the Grid object which should be used to determine the nearest grid point value.

If the direction is not specified, then all four edges of this box will be expanded to grid. Note that this method will always expand the edges of this box towards the outside, so that the SnapType for the specified *grid* will be ignored.

fix() – checks to see if this box is inverted; if it is inverted, then it will swap coordinates, so that the box will not be inverted, and then returns the resulting box.

Note that there are a large number of different ”get” methods which can be used to access various values for this box.

getArea() – returns the area of this box. Note that this method can return a negative area for a box which is inverted. It also may overflow if the resulting area value is too large to represent in database units.

getBottom() – returns the coordinate value for the bottom of this box

getCenter() – returns the coordinates for the center point of this box

getCenterX() – returns the x-coordinate value for the center point of this box

getCenterY() – returns the y-coordinate value for the center point of this box

getCoord(Direction *dir*) – returns the coordinate of this box in the specified *dir* direction. For example, if the *dir* direction value is NORTH, then the top coordinate value for this box will be returned. In the case of a full-line direction (NORTH_SOUTH or EAST_WEST), the center coordinate value will be returned. This *dir* direction should be one dimensional; otherwise, an exception is raised.

getDimension(Direction *dir*) – returns the dimensions of this box in the specified *dir* direction. If this direction is NORTH_SOUTH, then the height of this box is returned, while if it is EAST_WEST, the width of this box is returned. This *dir* direction should be either NORTH_SOUTH or EAST_WEST; otherwise, an exception is raised.

getHeight() – returns the height of this box

getLeft() – returns the coordinate value for the left side of this box

getLocationPoint(Location *loc*) – returns location point specified by the *loc* location

getLocationPoint(Direction *dir*) – returns location point specified by the *dir* direction

getPoints() – returns the list of four points which are defined by the boundary of this box. The returned point list is determined by traversing the four corners of this box in a counter-clockwise fashion, starting at the lower left corner.

getRange(Direction *dir*) – returns the coordinate values for this box in the specified *dir* direction, as a range value. For example, for the NORTH_SOUTH direction, the bottom

and top coordinate range value will be returned. This *dir* direction should be a full-line direction (NORTH_SOUTH or EAST_WEST); otherwise, an exception is raised.

getRangeX() – returns the left and right coordinate values for this box as a range value.

getRangeY() – returns the bottom and top coordinate values for this box as a range value.

getRight() – returns the coordinate value for the right side of this box

getSpacing(Direction *dir*, Box *refBox*) – returns the spacing value between this box and a specified *refBox* reference box in the specified *dir* direction. For example, if the specified direction is NORTH, then this method returns the distance from the bottom of this box to the top of the *refBox* reference box. If the specified direction is not one of the four primary directions (NORTH, SOUTH, EAST, WEST), then an exception is raised.

getTop() – returns the coordinate value for the top of this box

getWidth() – returns the width of this box

hasNoArea() – returns True if this box has zero area. This will be the case if the bottom and top coordinates are equal, or if the left and right coordinates are equal.

init() – sets this box to an inverted box with left and bottom coordinates set to maximum positive values, and the right and top coordinates set to maximum negative values. This inverted box can be useful as a starting point when several boxes are merged.

intersect(Box *box*) – sets this box to the intersection of this box with the passed *box* parameter object, determined using the minimum of the two box dimensions. This intersected box is then returned as the value of this method. Note that if these two boxes do not intersect, then this box will be set to an inverted box and returned as the value.

intersect(Box *box*, Direction *dir*) – sets this box to the intersection of this box with the passed *box* parameter object, in the directions specified by the *dir* Direction parameter. This *dir* Direction should contain one of the four primary directions (NORTH, SOUTH, EAST or WEST), or be the NULL or ANY direction, in which case the intersection will be taken in all four primary directions.

isInverted() – returns True if either the height or width value of this box is negative, and returns False otherwise.

isNormal() – returns True if this box has positive width and height, and False otherwise.

limit(Point *point*) – returns the point which is found by limiting the specified *point* to be within this box. The x coordinate value of the specified *point* is compared with the left and right coordinates of this box, and the y coordinate value is compared with the bottom and top coordinates of this box.

lowerCenter() – returns the lower center Point object for this box

lowerLeft() – returns the lower left Point object (vertex) for this box

Feedback

lowerRight() – returns the lower right Point object (vertex) for this box

merge(Box *box*) – sets this box to the merger of this box with the passed *box* parameter object; this merged box is then returned as the value of this method. Note that the merged box is the overall union of these two boxes, determined using the maximum of the two box dimensions.

merge(Box *box*, Direction *dir*) – sets this box to the merger of this box with the passed *box* parameter object, in the directions specified by the *dir* Direction parameter. This *dir* Direction should contain one of the four primary directions (NORTH, SOUTH, EAST or WEST), or be the NULL or ANY direction, in which case merging will be performed in all four primary directions. This merged box is then returned as the value of this method. Note that the merged box is the overall union of these two box objects, determined using the maximum of the box dimensions in the specified directions.

mergePoint(Point *p*) – this box is merged with the passed Point parameter *p*. This is done by using the x and y coordinate values of this Point parameter to expand this box, until this point defines one of the 4 vertices of a new box, which is returned as the merged box. If point *p* lies within this box, then this box is returned as the merged box.

mirrorX(Coord *yCoord*=0) – mirrors this box about the X coordinate axis. The optional *yCoord* parameter specifies a displacement for the location of the X coordinate axis.

mirrorY(Coord *xCoord*=0) – mirrors this box about the Y coordinate axis. The optional *xCoord* parameter specifies a displacement for the location of the Y coordinate axis.

moveBy(Coord *dx*, Coord *dy*) – moves this box by the specified *dx* and *dy* offset coordinate values. The *dx* value is added to the left and right coordinate values for this box, and the *dy* value is added to the bottom and top coordinate values for this box.

moveTo(Point *destination*, Location *loc*=Location.CENTER_CENTER) – moves this box to the specified *destination* point. The *loc* location is any of the nine possible location points on this box which should coincide with the *destination* point (see section on the Location class).

moveTowards(Direction *dir*, Coord *d*) – moves this box by the specified *d* coordinate value distance, in the specified *dir* direction. If the *dir* direction is an opposing direction (such as EAST_WEST or NORTH_SOUTH), then an exception is raised.

overlaps(Box *box*, bool *incEdges*=True) – returns True if this box has any overlap with the passed *box*. If the *incEdges* Boolean parameter is true, then the boxes will be considered to overlap, if the edges touch.

place(Direction *dir*, Box *refBox*, Coord *distance*, bool *align*=True) – places this box relative to the specified *refBox* reference box in the specified *dir* direction, using the specified *distance* value. If the optional *align* parameter is True, then this box will first be aligned with the reference box in the direction orthogonal to the specified *dir* direction. If this *align* parameter is False, then no alignment operation will be performed before the placement

operation. If the specified direction is not one of the four primary directions (NORTH, SOUTH, EAST, WEST), then an exception is raised.

removeRegion(Box *box*) – removes the region of this box specified by the *box* parameter. This *box* parameter must be a sub-box of this box, with one corner which matches a corner of this box. If the *box* parameter does not have a corner which matches any corner of this box, then an exception is raised.

rotate90(Point *origin*=None) – rotates this box by 90 degrees, in a counter-clockwise direction. If the *origin* parameter is not specified, then the (0,0) point will be used as the origin for this rotation operation.

rotate180(Point *origin*=None) – rotates this box by 180 degrees, in a counter-clockwise direction. If the *origin* parameter is not specified, then the (0,0) point will be used as the origin for this rotation operation.

rotate270(Point *origin*=None) – rotates this box by 270 degrees, in a counter-clockwise direction. If the *origin* parameter is not specified, then the (0,0) point will be used as the origin for this rotation operation.

set(Box *b*) – sets this box to have the top, bottom, left and right values of the Box *b*.

set(Box *b*, Direction *dir*=None) – sets this box to have the values of the Box *b*, in the directions specified by the *dir* Direction parameter. This *dir* Direction should contain one of the four primary directions (NORTH, SOUTH, EAST or WEST), or be the NULL or ANY direction, in which case all four sides of the box will be modified. If the direction is not specified, then all four sides of the box will be modified.

set(Point *lowerLeft*, Point *upperRight*) – sets this box to have the lower left and upper right point values of these two Point parameters *lowerLeft* and *upperRight*.

set(Coord *left*, Coord *bottom*, Coord *right*, Coord *top*) – sets this box to have left, bottom, right and top coordinate values of these *left*, *bottom*, *right* and *top* Coord parameters.

setBottom(Coord *v*) – sets the coordinate value for the bottom of this box

setCenter(Point *point*) – sets the center point for the center of this box

setCenterX(Coord *v*) – sets the x-coordinate value for the center of this box

setCenterY(Coord *v*) – sets the y-coordinate value for the center of this box

setCoord(Direction *dir*, Coord *coord*) – sets this box to have the coordinate value in the specified direction. For example, if the direction is NORTH, then the top coordinate value will be set to the specified value. If the *dir* direction is not one of the four primary directions (NORTH, SOUTH, EAST, WEST), then an exception is raised.

setDimension(Coord *coord*, Direction *dir*) – sets the dimensions of this box in the specified *dir* direction. If this direction is a primary direction (NORTH, SOUTH, EAST or WEST), then only the specified edge is modified; if it is either NORTH_SOUTH or

EAST_WEST, then two opposite edges are modifed. If this direction is NORTH_EAST, NORTH_WEST, SOUTH_EAST or SOUTH_WEST, then two adjacent edges are modified, while for a NULL or ANY direction, all four edges are modified.

setBottom(Coord *v*) – sets the coordinate value for the bottom of this box

setHeight(Coord *height*) – sets the height of this box

setLocationPoint(Location *loc*, Point *pt*) – moves this box, so that the specified point *pt* becomes the *loc* location point for this box. Note that if the location point is one if the CENTER locations, then the position of this box may vary, due to truncation errors.

setRange(Direction *dir*, Range *range*) – sets this box to have the specified coordinate value *range* in the given direction. For example, if the direction is NORTH_SOUTH, then the bottom and top coordinate values will be set to the specified *range* value. If the *dir* direction is not one of the two full-line directions (NORTH_SOUTH or EAST_WEST), then an exception is raised.

setRangeX(Range *range*) – sets this box to have the specified coordinate value *range* for the left and right coordinate values.

setRangeY(Range *range*) – sets this box to have the specified coordinate value *range* for the bottom and top coordinate values.

setRight(Coord *v*) – sets the coordinate value for the right side of this box

setTop(Coord *v*) – sets the coordinate value for the top of this box

setWidth(Coord *width*) – sets the width of this box

snap(Grid *grid*, SnapType *snapType*=None) – snaps lower left point of this box to the nearest grid point. The *grid* parameter is used to specify the Grid object which should be used to determine the nearest grid point value, while the *snapType* parameter specifies the rounding method which should be used.

snapX(Grid *grid*, SnapType *snapType*=None) – snaps the left coordinate of this box to the nearest grid point. The *grid* parameter is used to specify the Grid object which should be used to determine the nearest grid point value, while the *snapType* parameter specifies the rounding method which should be used.

snapY(Grid *grid*, SnapType *snapType*=None) – snaps the bottom coordinate of this box to the nearest grid point. The *grid* parameter is used to specify the Grid object which should be used to determine the nearest grid point value, while the *snapType* parameter specifies the rounding method which should be used.

snapTowards(Grid *grid*, Direction *dir*) – snaps the lower left point of this box to the nearest grid point in the specified direction *dir*. The *grid* parameter is used to specify the Grid object which should be used to determine the nearest grid point value. If the specified direction *dir* is not a half-line direction, then an exception is raised.

Feedback

transform(Transform *trans*) – applies the `transform` *trans* passed as a parameter to this box (see the description of the `Transform` object later in this section). Note that the values returned by the **isInverted**(), **isNormal**() and **hasNoArea**() methods will be the same after this transform is applied; these values are invariant under transformation.

upperCenter() – returns the upper center Point object for this box

upperLeft() – returns the upper left Point object (vertex) for this box

upperRight() – returns the upper right Point object (vertex) for this box

Attributes:

In addition to these methods for the Box class, there are also four attributes (or properties) defined for the Box object, described as follows:

bottom – the coordinate value for the bottom of this box

left – the coordinate value for the left side of this box

right – the coordinate value for the right side of this box

top – the coordinate value for the top of this box

Note that these **bottom**, **left**, **right** and **top** attribute values will return the same values as returned by the "**getBottom**()", "**getLeft**()", "**getRight**()" and "**getTop**()" methods. In addition, these attributes are settable, so that they can be used to set these values in the same manner as the "**setBottom**()", "**setLeft**()", "**setRight**()" and "**setTop**()" methods.

Examples:

```
box1 = Box(0, 0, 10.0, 10.0)
box2 = Box(Point(10.0, 0), Point(20.0, 20.0))
box1.overlaps(box2)             # returns True (edges touch)
box1.overlaps(box2, False)      # returns False (only edges touch)
box1.contains(box2)             # returns False
box2.getWidth()                 # returns 10.0
box2.getHeight()                # returns 20.0
loc = Location.UPPER_CENTER
box2.getLocationPoint(loc)      # returns Point(15,20)
box2.merge(box1)                # box2 is now Box(0,0,20,20)
box2.getTop()                   # returns 20.0
box2.getBottom()                # returns 0.0
box2.getLeft()                  # returns 0.0
box2.getRight()                 # returns 20.0
box2.top                        # returns 20.0
box2.bottom                     # returns 0.0
box2.left                       # returns 0.0
box2.right                      # returns 20.0
box2.getCenter()                # returns Point(10,10)
box2.getCenterX()               # returns 10.0
box2.getCenterY()               # returns 10.0
box2.getCoord(NORTH)            # returns 20.0
```

Feedback

```
box2.getCoord(SOUTH)            # returns 0.0
box2.getHeight()                # returns 20.0
box2.getDimension(NORTH_SOUTH)  # returns 20.0
box2.getCoord(EAST)             # returns 20.0
box2.getCoord(WEST)             # returns 0.0
box2.getWidth()                 # returns 20.0
box2.getDimension(EAST_WEST)    # returns 20.0
box2.lowerLeft()                # returns Point(0,0)
box2.upperRight()               # returns Point(20,20)
box2.isNormal()                 # returns True
box2.isInverted()               # returns False
box2.hasNoArea()                # returns False
box2.getArea()                  # returns 400.0
box2.getPoints()                # returns list of points
box2.set(box1).expand(2)        # can "chain" methods
box3 = Box(0, 0, 10.0, 10.0)
box4 = Box(20.0, 20.0, 30.0, 30.0)
box3.alignEdge(EAST, box4, WEST)
box3.getSpacing(WEST, box4)     # returns 0.0
box4.moveBy(10.0, 0.0)
box3.getSpacing(WEST, box4)     # returns 10.0
box3.place(NORTH, box4, 5.0)
box3.getSpacing(NORTH, box4)    # returns 5.0
```

Direction

Description: There are several different directions which can be defined for use with geometric layout objects in the Santana system. These directions are specified by one of the following constant objects, which are defined as attributes for this Direction class: NONE, NORTH, NORTH_EAST, EAST, SOUTH_EAST, SOUTH, SOUTH_WEST, WEST, NORTH_WEST, NORTH_SOUTH, EAST_WEST, ANY, CENTER.

These Direction objects can be used in conjunction with various functions within the Python API. For example, this Direction object is used to specify the directions to be used for relative geographical placement and alignment functions in the Python API.

A primary direction is one of the four directions NORTH, SOUTH, EAST or WEST. A one-dimensional direction is one of the directions defined by either the X-axis or by the Y-axis; these correspond to the directions NORTH, SOUTH, NORTH_SOUTH and EAST, WEST, EAST_WEST. A half-line direction is a direction defined by a half-line in the X and Y coordinate space; these correspond to the eight directions NORTH, SOUTH, EAST, WEST, NORTH_EAST, NORTH_WEST, SOUTH_EAST or SOUTH_WEST. A full-line direction is a direction defined by a full line in the X and Y coordinate space; these correspond to the directions NORTH_SOUTH or EAST_WEST.

There is no need to `construct` any Direction objects; simply use the Direction constants such as Direction.NORTH or Direction.SOUTH. In addition, as a shorthand notation, the direction name (without quotes) can also be used. For example, the name SOUTH will be the same object as Direction.SOUTH. (Note that these symbolic constant objects are

defined in the `cni.constants` Python module, which would need to be imported into the Python environment, if it is not already present).

Methods:

containsComponent(Direction *dir*) – returns True if the *dir* direction is a component of this Direction, and False otherwise. This *dir* direction component should be NORTH, SOUTH, EAST, WEST or CENTER. Note that Direction.NONE contains no direction components, while Direction.ANY contains all direction components.

extend() – extends a one dimensional Direction (such as NORTH or EAST) to a full-line Direction (such as NORTH_SOUTH or EAST_WEST). If this Direction is not a one dimensional Direction, then an exception is raised.

getJustifiedDir(Direction *justify*) – returns the half-line direction which is perpendicular to this primary Direction, justified relative to the specified *justify* direction. If this *justify* direction is EAST, then the returned direction is a 90 degree clockwise rotation of this Direction; if this *justify* direction is WEST, then the returned direction is a 90 degree counter-clockwise rotation. The *justify* direction should either be EAST or WEST; otherwise, an exception is raised. If this Direction is not one of the four primary directions (NORTH, SOUTH, EAST, WEST0, then an exception is raised.

getMembers() – returns list of all Direction objects defined for the Direction class.

Note that this method is defined as a static method, so that it can be called without explicitly creating a Direction object; for example, `Direction.getMembers()`.

getPrimaryDirs () – returns the list of one or more primary directions which are contained in this Direction.

is1Dimension() – returns True if this Direction is a one dimensional direction defined by either the X-axis or the Y-axis (NORTH, SOUTH, NORTH_SOUTH, EAST, WEST or EAST_WEST) and returns False, otherwise.

isHalfLine() – returns True if this Direction is one of the eight half-line directions (NORTH, SOUTH, EAST, WEST, NORTH_EAST, NORTH_WEST, SOUTH_EAST or SOUTH_WEST) and returns False, otherwise.

isFullLine() – returns True if this Direction is one of the full-line directions (NORTH_SOUTH or EAST_WEST) and returns False, otherwise.

isPrimary() – returns True if this Direction is one of the four primary directions (NORTH, SOUTH, EAST or WEST) and returns False, otherwise.

isXDir() – returns True if this Direction is one of the directions defined by the X-axis (EAST, WEST or EAST_WEST) and returns False, otherwise.

isYDir() – returns True if this Direction is one of the directions defined by the Y-axis (NORTH, SOUTH or NORTH_SOUTH) and returns False, otherwise.

Feedback

mapXDirToYDir() – maps this Direction from a direction defined by the X-axis to a direction defined by the Y-axis. For example, the Direction EAST would be mapped to the Direction NORTH. If this Direction is not a direction defined by the X-axis, then an exception is raised.

mapYDirToXDir() – maps this Direction from a direction defined by the Y-axis to a direction defined by the X-axis. For example, the Direction NORTH would be mapped to the Direction EAST. If this Direction is not a direction defined by the Y-axis, then an exception is raised.

mirrorX() – returns the direction which results when this Direction is mirrored about the X-axis. For example, the Direction.NORTH would be mirrored to the Direction.SOUTH.

mirrorY() – returns the direction which results when this Direction is mirrored about the Y-axis. For example, the Direction.EAST would be mirrored to the Direction.WEST.

opposite() – returns the opposite half-line direction to this Direction. If this Direction is not one of the eight one dimensional directions defined by either the X-axis or the Y-axis, then an exception is raised.

perpendicular() – returns the perpendicular direction to this Direction. Note that this Direction will first be extended to a full-line direction, before determining the perpendicular direction. For example, if this Direction is EAST or WEST, then it will be extended to the Direction EAST_WEST, and the perpendicular Direction NORTH_SOUTH will be returned by this method. If this Direction is not a direction defined by either the X-axis or the Y-axis, then an exception is raised.

rotate90() – returns the direction which results when this Direction is rotated 90 degrees, in a counter-clockwise direction. For example, the Direction.SOUTH would be rotated to the Direction.EAST.

rotate180() – returns direction which results when this Direction is rotated 180 degrees, in a counter-clockwise direction. For example, the Direction.SOUTH would be rotated to the Direction.NORTH.

rotate270() – returns direction which results when this Direction is rotated 270 degrees, in a counter-clockwise direction. For example, the Direction.SOUTH would be rotated to the Direction.WEST.

transform(Transform *trans*) – returns the direction which results when the `transform` *trans* passed as a parameter is applied to this Direction (see the description of the `Transform` object later in this section). For example, the Direction.SOUTH would be transformed to the Direction.EAST, when the *trans* transform is Transform(0, 0, R90).

Examples:

```
d1 = Direction.NORTH
d2 = NORTH                    # same Direction object
d1 == d2                      # returns True
```

Feedback

```
d1.isPrimary()               # returns True
d1.isFullLine()              # returns False
d1.isHalfLine()              # returns True
d1.isXDir()                  # returns False
d1.isYDir()                  # returns True
d1.is1Dimension()            # returns True
d1.extend()                  # returns Direction.NORTH_SOUTH
d1.mapYDirToXDir()           # returns Direction.EAST
d1.opposite()                # returns Direction.SOUTH
d1.perpendicular()           # returns Direction.EAST_WEST
d1.getJustifiedDir(EAST)     # returns Direction.EAST
d1.getJustifiedDir(WEST)     # returns Direction.WEST
d1.getPrimaryDirs()          # returns [Direction.NORTH]
d3 = Direction.NORTH_SOUTH
d3.getPrimaryDirs()          # returns [Direction.SOUTH,
                                       Direction.NORTH]
d3.containsComponent(NORTH)  # returns True
d3.containsComponent(SOUTH)  # returns True
d3.containsComponent(WEST)   # returns False
# also illustrate use of mirror, rotate and transform methods
d4 = Direction.EAST
d4.mirrorX()                 # returns Direction.EAST
d4.mirrorY()                 # returns Direction.WEST
d4.rotate90()                # returns Direction.NORTH
d4.rotate180()               # returns Direction.WEST
d4.rotate270()               # returns Direction.SOUTH
trans = Transform(0, 0, R180)
d4.transform(trans)          # returns Direction.WEST
```

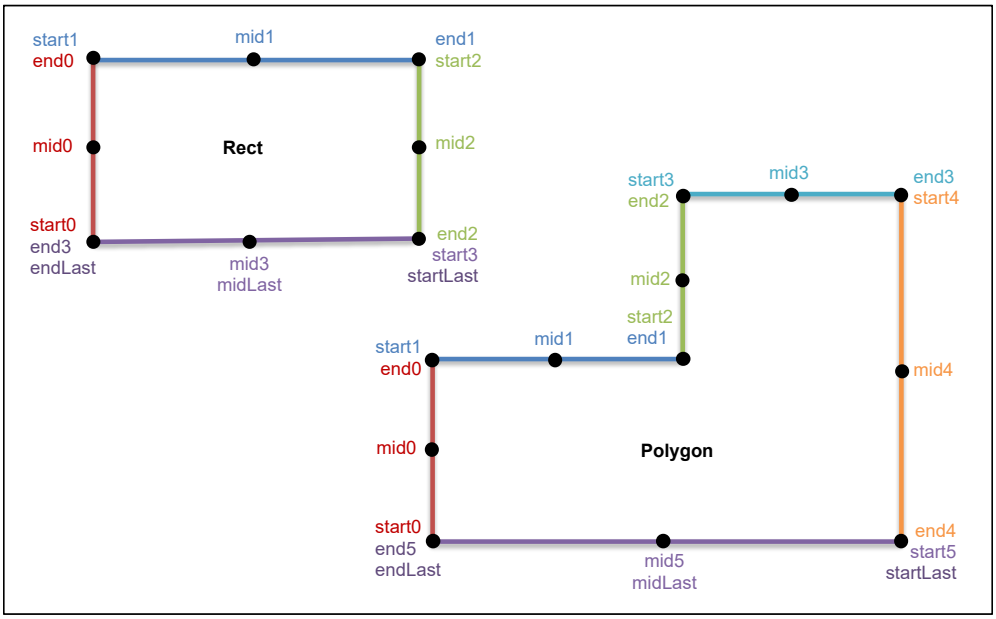
Location

Description: There are several different locations which can be defined for use with geometric layout objects in the Santana system. These locations are specified by one of the following constant objects, which are defined as attributes for this Location class: LOWER_LEFT, CENTER_LEFT, UPPER_LEFT, LOWER_CENTER, CENTER_CENTER, UPPER_CENTER, LOWER_RIGHT, CENTER_RIGHT, UPPER_RIGHT.

For Rect and Polygon objects, the following attributes are reserved names for index locations:

- Each segment has three point locations: start*n* at the beginning, mid*n* in the middle, and end*n* at the end of the segment, where *n* is the segment number.
- The last segment has three additional locations: startLast, midLast, and endLast.

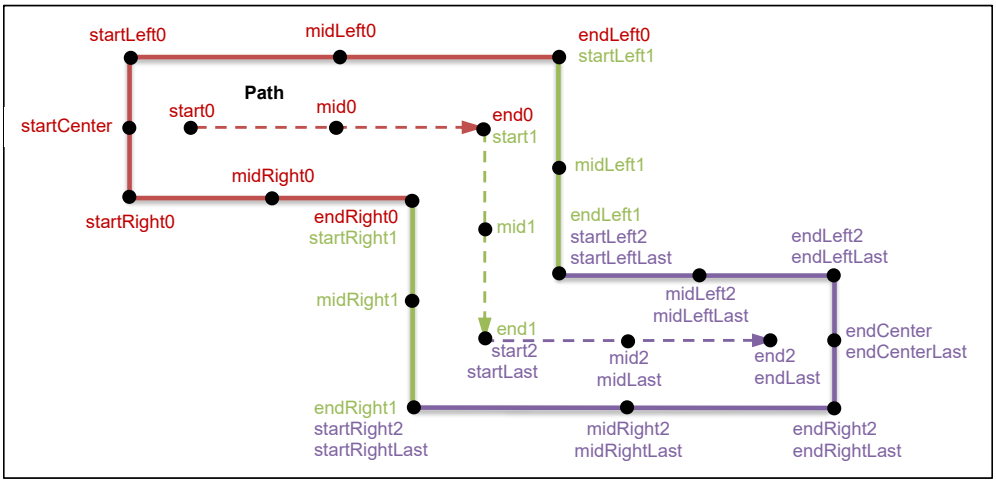
Feedback



Polygon

For Path objects, the following attributes are reserved names for index locations:

- Each centerline segment has three point locations: start*n* at the beginning, mid*n* in the middle, and end*n* at the end of the segment, where *n* is the segment number.
- Each boundary segment has point locations denoted as either Left or Right: startLeft*n*|startRight*n* at the beginning, midLeft*n*|midRight*n* in the middle, and endLeft*n*|endRight*n* at the end of the segment, where *n* is the segment number.
- The start segment has the point location: startCenter.
- The last segment has the point locations: startLast, startLeftLast, startRightLast, midLast, midLeftLast, midRightLast, endLast, endLeftLast, endRightLast, endCenter, and endCenterLast.



These Location objects can be used in conjunction with various functions within the Python API. This Location object is typically used to specify one of nine standard locations on a box, so that it is used to specify the locations for different types of movement and alignment functions, which make use of bounding box operations.

Creation:

There is no need to `construct` any Location objects; simply use the Location constants for the nine standard pre-defined locations, such as Location.LOWER_LEFT or Location.UPPER_RIGHT.

In addition to the standard pre-defined locations, it is possible to create a named custom location by means of a call to setCustomLocation(). This custom location can be used with the alignLocation, alignLocationToPoint, and moveTo methods of PhysicalComponent and PhysicalCompRef.

Location(string *name*) – creates a custom or index location with the specified name. Index locations are created with the reserved names shown above. Note that if the name of a standard location is provided, it does not create a new location and only returns the corresponding pre-defined location.

Methods:

mirrorX() – returns the location which results when this Location is mirrored about the X-axis. For example, the Location.LOWER_LEFT would be mirrored to the Location.UPPER_LEFT.

mirrorY() – returns the location which results when this Location is mirrored about the Y-axis. For example, the Location.LOWER_LEFT would be mirrored to the Location.LOWER_RIGHT.

Feedback

rotate90() – returns the location which results when this Location is rotated 90 degrees, in a counter-clockwise direction. For example, the Location.LOWER_RIGHT would be rotated to the Location.UPPER_RIGHT.

rotate180() – returns location which results when this Location is rotated 180 degrees, in a counter-clockwise direction. For example, the Location.LOWER_RIGHT would be rotated to the LocationUPPER_LEFT.

rotate270() – returns location which results when this Location is rotated 270 degrees, in a counter-clockwise direction. For example, the Location.LOWER_RIGHT would be rotated to the Location.LOWER_LEFT.

transform(Transform *trans*) – returns the location which results when the `transform` *trans* passed as a parameter is applied to this Location (see the description of the `Transform` object later in this section). For example, the Location.LOWER_RIGHT would be transformed to the Location.UPPER_RIGHT, when the *trans* transform is Transform(0, 0, R90).

Examples:

```
loc1 = Location.LOWER_LEFT
loc2 = Location.LOWER_LEFT   # same Location object
loc1 == loc2                 # returns True
# also illustrate use of mirror, rotate and transform methods
loc3 = Location.LOWER_RIGHT
loc3.mirrorX()               # returns Location.UPPER_RIGHT
loc3.mirrorY()               # returns Location.LOWER_LEFT
loc4 = Location.UPPER_RIGHT
loc4.rotate90()              # returns Location.UPPER_LEFT
loc4.rotate180()             # returns Location.LOWER_LEFT
loc4.rotate270()             # returns Location.LOWER_RIGHT
trans = Transform(0, 0, MY)
loc4.transform(trans)        # returns Location.UPPER_LEFT
```

Orientation

Description: In This Orientation class provides the ability to rotate an object or mirror it about the X or Y coordinate axes. In addition, an Orientation object can be defined which performs a combination of rotation and mirroring. The basic orientation operations can be defined using the set of pre-defined symbolic constant objects R0, R90, R180, R270, MY, MYR90, MX, MXR90, which can be defined as follows:

R0 - represents no change in orientation (the identity)

R90 - represents a 90 degree rotation

R180 - represents a 180 degree rotation

R270 - represents a 270 degree rotation

MY - represents mirroring about the Y coordinate axis

MYR90 - represents mirroring about the Y axis, followed by a 90 degree rotation

MX - represents mirroring about the X coordinate axis

MXR90 - represents mirroring about the X axis, followed by a 90 degree rotation

Note that all of these rotations are counter-clock-wise rotations. Also note that mirroring about the Y coordinate axis means to mirror horizontally, while mirroring about the X coordinate axis means to mirror vertically.

There is no need to `construct` any Orientation objects; simply use the Orientation constants such as Orientation.R90 or Orientation.MX. In addition, as a shorthand notation, the orientation name (without quotes) can also be used. For example, the name R90 will be the same object as Orientation.R90. (Note that these symbolic constant objects are defined in the `cni.constants` Python module, which would need to be imported into the Python environment, if it is not already present).

Methods:

concat(Orientation *other*) – concatenates this Orientation object with the Orientation object passed as a parameter, and returns the resulting Orientation object as the return value. This concatenation operation simply applies these two orientation operations in sequence, and then returns the result as another Orientation object.

getRelativeOrient(Orientation *o*) – returns the relative orientation which is required to transform this Orientation object into the orientation specified by the *o* Orientation parameter. This can be thought of as the `inverse concatenation operation`.

Examples:

```
o1 = Orientation.R90
o2 = R90                  # Orientation object with same value
o1 == o2                  # returns True
# test concatenation operation
o3 = o2.concat(o1)        # o1 operation followed by o2 operation
o3 == Orientation.R180    # returns True
# test inverse concatenation operation
O1 = R90
O2 = R270
O3 = o1.getRelativeOrient(o2)
O3 == Orientation.R180    # returns True
```

Transform

Description: The Transform class provides the ability to implement two-dimensional transformations, consisting of orientation changes (rotations and mirroring about the

coordinate axes), translation (offsets in the X and Y directions), and magnification of the X and Y coordinates, with the operations performed in the following order:

1. Rotation/Mirroring
2. Translation
3. Magnification

When rotation operations are performed on an object in the layout design, it is important to note that it may be necessary to first translate the object to the origin of the DLO coordinate system, apply the rotation operation, and then translate the object back to its original location. This would be necessary, because the center of rotation is the origin of the coordinate system, not the center of the object. With this approach, the object will be rotated about the center of the object. Otherwise, the resulting rotation may not produce the expected results. In order to more easily handle this situation, the **rotate**() methods are provided by this Transform class.

Creation:

The Transform object can be directly created using the desired x and y coordinate values for the translation operation, the desired orientation value, and the desired magnification value. The individual x and y coordinate values can be specified, or the corresponding Point object can be used instead. Thus, this Transform object can be created using either of the following forms:

Transform(Coord *x*, Coord *y*, Orientation *o*=R0, double *mag*=1.0)

Transform(Point *offset*, Orientation *o*=R0, double *mag*=1.0)

If these values are not specified, then default values will be generated and used.

Methods:

concat(Transform *transform*) – concatenates this Transform with the Transform *transform* passed as a parameter, and returns the concatenated Transform as the return value. Note that this original Transform object is left unchanged.

invert() – inverts this Transform, and returns the inverted Transform as the return value. When this Transform is concatenated with the returned inverse Transform, then the identify Transform will result. Note that this original Transform object is left unchanged.

mirrorX(Coord *yCoord*=0) – returns a Transform which will set the orientation for the transform to be the MX mirroring orientation; if the *yCoord* parameter is specified, then this value will be used as the offset for applying the transform. Note that this original Transform object is left unchanged.

mirrorY(Coord *xCoord*=0) – returns a Transform which will set the orientation for the transform to be the MY mirroring orientation; if the *xCoord* parameter is specified, then this

value will be used to specify the offset for applying the transform. Note that this original Transform object is left unchanged.

rotate90(Point *origin*=None) – returns a Transform which will set the orientation for the transform to be the R90 rotation orientation; if the *origin* parameter is specified, then this value will be used as the origin for applying the transform. Note that this original Transform object is left unchanged.

rotate180(Point *origin*=None) – returns a Transform which will set the orientation for the transform to be the R180 rotation orientation; if the *origin* parameter is specified, then this value will be used as the origin for applying the transform. Note that this original Transform object is left unchanged.

rotate270(Point *origin*=None) – returns a Transform which will set the orientation for the transform to be the R270 rotation orientation; if the *origin* parameter is specified, then this value will be used as the origin for applying the transform. Note that this original Transform object is left unchanged.

Note that these mirror and rotate methods are all defined as static methods, so that they can be called without explicitly creating a Transform object. For example, they can be directly called using the syntax "Transform.mirrorX**()**" or `Transform.rotate90()`.

Attributes:

In addition to these methods for this Transform class, there are also four attributes (or properties) defined for this Transform class, described as follows:

offset – returns the offset point value for this Transform

orientation – returns the orientation value for this Transform

mag – returns the magnification value for this Transform

xOffset – returns the x-coordinate value of the offset for this Transform

yOffset – returns the y-coordinate value of the offset for this Transform

Examples:

```
t1 = Transform(Point(10.0, 20.0), Orientation.R90)
t1.offset                # returns Point(10,20)
t1.xOffset               # returns 10.0
t1.yOffset               # returns 20.0
t1.orientation           # returns Orientation.R90
t2 = t1.invert()         # returns inverted Transform object
t2.offset                # returns Point(-20,10)
t2.xOffset               # returns -20.0
t2.yOffset               # returns 10.0
t2.orientation           # returns Orientation.R270
# check concatenation with inverted transform gives identity
t3 = t1.concat(t2)
t3.orientation           # returns Orientation.R0
```

Feedback

```
t3.offset                # returns Point(0,0)
t4 = t2.concat(t1)
t4.orientation           # returns Orientation.R0
t4.offset                # returns Point(0,0)
```

In addition to these basic geometric classes, there are also three auxiliary classes, which are used by several classes in the Python API. The Font class is used to specify the font style for a Text object. This Font class is used by the Text basic Shape class, as well as the AttrDisplay class. The PathStyle class is used to specify the path style for the beginning and ending points for a path. This PathStyle class is used by the Path basic Shape class, as well as the MultiPath derived compound component class. The GapStyle class is used to specify how a set of equally sized sub-rectangles contained within a larger enclosing rectangle should be spaced apart from each other. This GapStyle class is used by the Range basic geometric class, the Rect basic Shape class, the MultiPath derived compound component class, as well as the DeviceContact class.

Font

Description: There are nine different font styles which can be used to specify the font which should be used for a text object. These different fonts are specified by one of the following constants, which are defined as attributes for this Font class: EURO_STYLE, FIXED, GOTHIC, MATH, MIL_SPEC, ROMAN, SCRIPT, STICK and SWEDISH.

Note that there is no need to `construct` any Font objects; simply use the Font constants such as Font.GOTHIC or Font.SWEDISH.

Methods:

calcBBox(str *text,* Point *origin*,Coord *height*,Location *location*=Location.UPPER_LEFT, Orientation *orient*=Orientation.R0, bool *overbar*=False) – calculates the bounding box for this Font object for the specified text string.

getMembers() – returns a uniform list of the enumerated members for this Font class.

Note that this method is a static method, so that it can be called without explicitly creating a Font class object. For example, it can be directly called using the syntax `Font.getMembers()`.

Examples:

```
# obtain list of Font class constants
Font.getMembers()

# compare bounding boxes for text strings using different fonts
font1 = Font.STICK
font2 = Font.SCRIPT
font3 = Font.GOTHIC
font1.calcBBox('pin1', Point(0,0), 0.5)
```

Feedback

```
font2.calcBBox('pin1', Point(0,0), 0.5)
font3.calcBBox('pin1', Point(0,0), 0.5)
```

PathStyle

Description: There are four different path styles which can be used to specify the path style for the beginning and ending points for a path. These different path styles are specified by one of the following constants, which are defined as attributes for this PathStyle class: TRUNCATE, EXTEND, ROUND and VARIABLE. The TRUNCATE path style will provide no extension beyond the beginning and ending points, while the EXTEND path style will extend both the beginning and ending points by one half the width of the path. The ROUND path style will generate an octagonal extension which results in three edges which extend both the beginning and ending points by one third width of the path. The VARIABLE path style allows the beginning and ending points to have different extensions; for this path style, the designer provides the desired extension values for the beginning and ending points.

There is no need to `construct` any PathStyle objects; simply use the PathStyle constants such as PathStyle.TRUNCATE or PathStyle.ROUND.

Methods:

Note that there currently are no methods defined for this PathStyle class.

Examples:

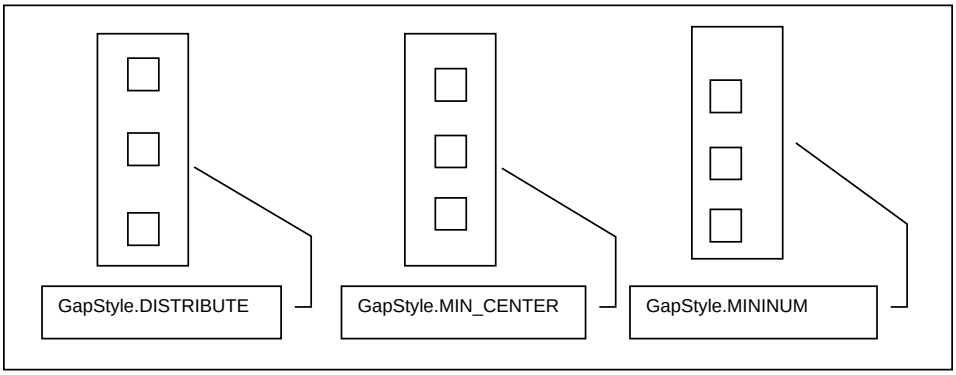
```
ps1 = PathStyle.EXTEND
ps2 = PathStyle.EXTEND          # same PathStyle object
ps1 == ps2                      # returns True
```

GapStyle

Description: There are three different gap styles which can be used to specify the spacing between equally sized sub-rectangles contained inside a larger enclosing rectangle. These different gap styles are specified by one of the following constants, which are defined as attributes for this GapStyle class: DISTRIBUTE, MINIMUM and MIN_CENTER. The DISTRIBUTE value indicates that these rectangles should be spread out as far as possible, while the MINIMUM value indicates that these rectangles should be packed as closely as possible, from the lower left corner of the enclosing rectangle. The MIN_CENTER value indicates that these rectangles should be packed as closely as

Feedback

possible in the center of the larger enclosing rectangle. These different GapStyle values are illustrated in the following diagram:



These GapStyle objects can be used in conjunction with various functions within the Python API. This GapStyle class is typically used when specifying the spacing for a field of via cuts, to connect one layer to another layer in a layout design.

There is no need to `construct` any GapStyle objects; simply use the GapStyle constants such as GapStyle.DISTRIBUTE or GapStyle.MINIMUM.

Methods:

Note that there currently are no methods defined for this GapStyle class.

Examples:

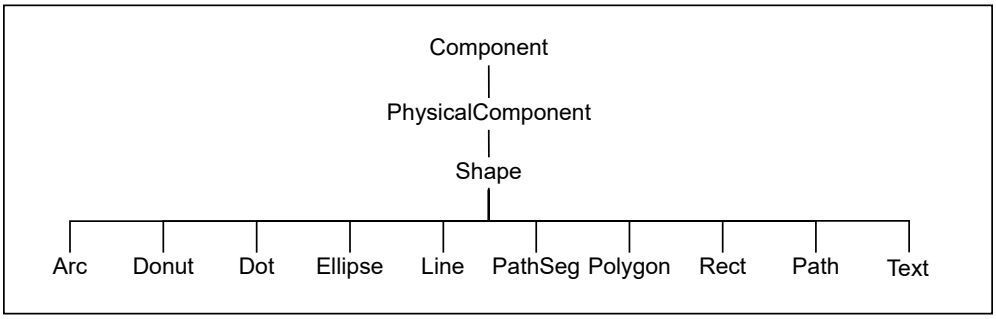
```
gs1 = GapStyle.DISTRIBUTE
gs2 = GapStyle.DISTRIBUTE       # same GapStyle object
gs1 == gs2                      # returns True
```

PHYSICAL COMPONENT CLASSES

The PhysicalComponent class is an abstract base class which is used to derive class objects which represent the physical components or shapes which specify the geometry which can be manufactured in a chip layout. All of these physical components are placed on one or more Layer objects, which represent the layers in the chip layout. The Shape class is derived from this base PhysicalComponent class, and is used to represent the shapes which can be used in physical layout designs. These different Shape objects are constructed on a single Layer object, while the Instance and Grouping class objects can be constructed on multiple Layer objects. This Shape class consists of the following derived different types of shapes: arc, rectangle, polygon, ellipse, donut, line, dot, path, simple path, and path segment. In addition, the text shapes are used to provide viewable text labels on different design objects.

Feedback

Any PhysicalComponent object has a type, which is one of the different types of generic shape objects, and can be represented using the following class inheritance diagram:



There are a large number of useful methods which are defined for the base PhysicalComponent class. Note that all of these methods are inherited by all of the Shape and Instance classes which are derived from this base PhysicalComponent class. The section on the ShapeFilter class in the Chapter These methods are described as follows:

Methods:

Note that many of the methods for the PhysicalComponent class make use of the ShapeFilter class to allow for the selection of one or more layers contained in the physical component. The ShapeFilter class allows the designer to specify a layer,

or a list of layers, which should be considered (or filtered) when performing operations on physical components, such as bounding box or placement calculations. Note that by default, all layers in the physical component will be considered. The description of the ShapeFilter class in the `TECHNOLOGY RELATED CLASSES` section should be consulted for a complete description of the capabilities and uses of the ShapeFilter class.

abut(Direction *dir*, PhysicalComponent *refComp*, ShapeFilter *filter* = ShapeFilter(), bool *align*=True, ShapeFilter *refFilter*=None) – moves this physical component in the given direction *dir*, so that the bounding box for this physical component just touches the bounding box for the *refComp* physical component. This is done using the bounding box for the *comp* physical component as determined by the *filter* ShapeFilter, and the bounding box for the *refComp* physical component as determined by the *refFilter* ShapeFilter. If the *refFilter* is None (the default value), then the *filter* ShapeFilter will also be used for the *refComp* physical component. If the *align* parameter is True, then this physical component is first aligned with the *refComp* physical component in the direction orthogonal to the *dir* direction; if this *align* parameter is False, then no alignment is performed. If the *dir* direction is not one of the four primary directions (NORTH, SOUTH, EAST or WEST), then an exception is raised. If the *refComp* physical component is the same as this physical component, then an exception is raised.

alignEdge(Direction *dir*, PhysicalComponent *refComp*, Direction *refDir*=None, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *refFilter*=None, Coord *offset*=None) – aligns this physical component with the *refComp* physical component, in the given direction *dir*, using the

bounding boxes for these two physical components as determined from the layers specified by the *filter* ShapeFilter parameter. This is a one-dimensional alignment operation, based upon the specified direction for each component. If the *refDir* parameter is not specified, then it will be assigned the same value as the *dir* parameter. If the *refFilter* parameter is not specified, then it will be assigned the same value as the *filter* parameter. This physical component is moved in the proper direction, until the specified edge of its bounding box is aligned with the specified edge of the bounding box for the *refComp* physical component. If the *offset* parameter is specified, then the value of this *offset* parameter will be added to create the total displacement for this one-dimensional alignment operation. Note that the only valid directions are the primary direction values NORTH, SOUTH, NORTH_SOUTH, EAST, WEST or EAST_WEST. If any other direction values are specified for the *dir* or *refDir* parameters, an exception is raised. If the *refComp* physical component is the same as this physical component, then an exception is raised.

alignEdgeToPoint(Direction *dir*, Point *point*, ShapeFilter *filter*=ShapeFilter()) – aligns this physical component with the *point* in the given direction *dir*, using the bounding box for this component as determined from the layers specified by the *filter* ShapeFilter parameter. This is the most general form of one-dimensional edge alignment, which is used when it is difficult to generate a reference bounding box for alignment operations.

alignLocation(Location *loc*, PhysicalComponent *refComp*, Location *refLoc*=None, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *refFilter*=None, Point *offset*=None) – aligns this physical component with the *refComp* physical component, for the specified locations, using the bounding boxes for these two physical components as determined from the layers specified by the ShapeFilter parameters. This is a two-dimensional alignment operation, based upon the specified location for the bounding box of each component. If the *refLoc* parameter is not specified, then it will be assigned the same value as the *loc* parameter. If the *refFilter* parameter is not specified, then it will be assigned the same value as the *filter* parameter. This physical component is moved as necessary, until the specified location point for its bounding box is the same point as the specified location point for the *refComp* bounding box. If the *offset* parameter is specified, then this point will be added to create the total displacement for this two-dimensional alignment operation. Note that this location alignment operation may produce overlapping physical components, based upon the specified bounding box location points. Also note that any of the nine different location points for a bounding box, or a custom location point set by setCustomLocation(), are valid values for the *loc* and *refLoc* parameters. If the *refComp* physical component is the same as this physical component, then an exception is raised.

alignLocationToPoint(Location *loc*, Point *point*, ShapeFilter *filter*=ShapeFilter()) – aligns this physical component with the specified *point*, for the specified location, using the bounding box for this physical component as determined from the layers specified by the *filter* ShapeFilter parameter. This is the most general form of two-dimensional location alignment, which can be used when it is difficult to generate a reference bounding box for alignment operations. Any of the nine different location points for a bounding box, or a custom location point set by setCustomLocation(), are valid values for the *loc* parameter.

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this PhysicalComponent object. Note that this method on the base PhysicalComponent class is effectively a virtual method, which should be overridden by any derived classes based upon this object. The cloning process will necessarily require a detailed understanding of the derived object.

destroy() – destroys this physical component. Note that the underlying OpenAccess object will also be destroyed and removed from the design database.

find(string *name*=) – returns the physical component having this *name* in the current design. This is done by searching through all physical components. If there is no physical component having this name, then None will be returned. If the *name* parameter is the empty string, then the first physical component in the design will be returned.

Note that this method is a static method, so that it can be called without explicitly creating a physical component. For example, it can be directly called using the syntax `PhysicalComponent.find('comp1')`.

fgAbut(Direction *dir*, PhysicalComponent *refComp,* LayerList *abutLayers*, ShapeFilter *filter*=ShapeFilter(), PhysicalComponent *env*=None, bool *align*=True, ShapeFilter *refFilter*=None, dict *options*=None, float *grid*=None) – this method combines the functionality of the **fgPlace**() and **fgFill**() methods. This is done by placing this physical component next to the *refComp* physical component in the given direction *dir*, while simultaneously abutting or overlapping them on the layers contained in the *abutLayers* list of layers. If necessary, new shapes will be created to fill any gaps between these two physical components on the layers contained in the *abutLayers* list of layers. This set of shapes is then used to create a Grouping object, which is the return value for this method. If no shapes were created, then this Grouping object will be empty. If the *refFilter* parameter is not specified, then it will be assigned the same value as the *filter* parameter. If the default *env* parameter is specified, then this additional physical component will be considered when the `smart place` operation is performed. If the *refComp* physical component is the same as this physical component, then an exception is raised.

The user can modify the behavior of the method by means of the *options* parameter.

At the moment, there are seven options available: "infiniteLayers", "envFilter", "ruleIds", "reduceGeometry", "useInitialPlacement", "bboxFilter", and "refBboxFilter".

- The "infiniteLayers" option allows you to specify a layer list for enclosing shapes that are supposed to be created later. The effect of this option is similar to activating a Device Context with these layers specified as the device context layers in the techfile. The geometry engine works as if there were shapes on these layers that enclose all the geometry under consideration. This option can be used if you do not have a suitable Device Context defined in the techfile but know that some enclosing shape will be created later that influences the current construction.

For example, *options* = {'infiniteLayers' : [Layer('pimp'), Layer('nwell')]}

- The "envFilter" option provides a separate shape filtering for the `env` component.
- The "ruleIds" option is a list of strings (rule IDs) that allows the user to restrict the consideration of spacing rules to only those specified in the list. By default, all spacing rules are considered.
- The "reduceGeometry" option is a boolean flag that can be turned on to optimize the run-time when placing large physical components.
- The "useInitialPlacement" option is a boolean flag that prevents the physical components from coming closer than the current spacing even if design rules allow.
- The "bboxFilter" can be used to specify how to calculate the bounding box of the physical component. By default, *filter* is used to calculate the b-box.
- The "refBboxFilter" option can be used to specify how to calculate the bounding box of the *refComp* component. By default, *refFilter* is used to calculate the b-box.

If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

fgAddEnclosingPolygon(Layer *layer,* ShapeFilter *filter*=ShapeFilter(), PhysicalComponent *env*=None, float *grid*=None) – creates an enclosing polygon for this physical component on the desired layer passed as the *layer* parameter. The *filter* ShapeFilter parameter can be used to indicate which layers should be considered when generating this enclosing polygon. The sizing for this enclosing polygon is determined by use of the minimum enclosure rule defined for these layers (two layer rules). If there is no minimum enclosure rule defined for the layer on which the component shape is drawn and the specified *layer* parameter value, then no enclosing polygon will be generated. In addition, if the default *env* parameter is specified, then the layers present for this additional physical component will be considered during the generation of the enclosing polygon. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point. Note that this enclosing polygon is generated by following the contours of this physical component's shape, so that non-rectangular polygons can be generated. This enclosing polygon is used to create a Grouping object, which is the return value for this method.

fgAnd(PhysicalComponent *comp*, Layer *resultLayer*, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *compFilter*=None, float *grid*=None) – performs a logical and operation for this physical component and the *comp* physical component, by selecting those polygon areas which are in both physical components. This calculation is performed using the layers specified in the *filter* and *compFilter* ShapeFilter objects for each of these physical components. The resulting polygon shapes are generated on the *resultLayer* layer. Note that these shape filters will first be used to merge all layers of the two physical components into geometries on the single *resultLayer* layer, and then the Boolean operation will be performed on the merged geometries of the *resultLayer* for both physical components. For example, if the physical component contains diffusion and metal layers, and the *comp* reference physical component contains poly and metal layers, while the *filter*

shape filter is diffusion and the *refFilter* shape filter is poly, then the resulting geometry for the Boolean AND operation will be the MOS gate channel. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point. In addition, these polygon shapes are used to create a Grouping object, which is the return value for this method. If there are no polygon shapes, then this Grouping object is empty.

fgDeriveLayer(string *derivationScript*, Layer *layer*, ShapeFilter *filter*=ShapeFilter(), float *grid*=None) – runs the DRC program to check DRC correctness for this physical component, using the design rules in the technology file which is attached to the current design. The *derivationScript* parameter contains the string of DRC commands to be executed, while the *layer* parameter specifies the layer which should be used to generate DRC error marker geometries. The *filter* ShapeFilter parameter can be used to specify the exact layers which should be used during this DRC run; by default all layers is considered. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

fgEnclose (LayerList *layers*, ShapeFilter *filter*=ShapeFilter(), dict *options*=None, float *grid*=None) – creates rectangles that enclose the physical component on every layer in the given layer list ("layers" parameter). All minimum enclose design rules are considered to produce the DRC correct set of enclosing rectangles. The resulting rectangles also satisfy minimum width rules. Note that any minimum area rules or off-grid rules are not considered as part of this calculation

By default, all the shapes in the physical component are taken into consideration. The *filter* parameter allows you to change the default behavior. For example, *filter* = ShapeFilter(Layer("metal1")) forces the method to disregard all shapes in the physical component except for metal1 shapes.

The user can modify the behavior of the method by means of *options* parameter.

At the moment, the only available option is "infiniteLayers".

- The "infiniteLayers" option allows you to specify a layer list for enclosing shapes which are supposed to be created later. The effect of this option is similar to activating a Device Context with these layers specified as the device context layers in the techfile. The geometry engine works as if there were shapes on these layers that enclose all the geometry under consideration (both input shapes and the enclosing rectangles under construction). This option can be used if you do not have a suitable Device Context defined in the techfile but know that some enclosing shape will be created later that influences the current construction.

For example, options = {'infiniteLayers' : [Layer('pimp'), Layer('nwell')]}

The method returns a Grouping of created enclosing rectangles. The order of the rectangles in the grouping follows the order of the layers in the *layers* input parameter. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

Feedback

fgExtend(PhysicalComponent *comp*, ShapeFilter filter=ShapeFilter(), dict *options*=None, float *grid*=None) – can be called on a single Rect object or on a grouping of Rect objects. The rectangles are extended to satisfy all min extension design rules between the rectangles and other shapes overlapping them (provided by the *comp* parameter).

By default, all the shapes in the *comp* parameter are taken into consideration. The *filter* parameter allows you to change the default behavior. For example, *filter* = ShapeFilter(Layer("metal1")) forces the method to disregard all shapes in the *comp* parameter except for metal1 shapes.

The user can modify the behavior of the method by means of *options* parameter.

At the moment, there are two options available: "direction" and "infiniteLayers".

- The "direction" option allows you to specify list of directions to extend (Direction.ANY by default). For example, options = {'direction' : [Direction.NORTH, Direction.EAST]}
- The "infiniteLayers" option allows you to specify a layer list for enclosing shapes that are supposed to be created later. The effect of this option is similar to activating a Device Context with these layers specified as the device context layers in the techfile. The geometry engine works as if there were shapes on these layers that enclose all the geometry under consideration (both shapes from "comp" parameter and the rectangles to be extended). This option can be used if you do not have a suitable Device Context defined in the techfile but know that some enclosing shape will be created later that influences the current construction.

For example, options = {'infiniteLayers' : [Layer('pimp'), Layer('nwell')]}

If successful, the method returns True. If it cannot produce DRC correct results, it returns False, and does not modify the geometry of the physical component.

This functionality is similar to the fgEnclose method. The essential difference is that fgEnclose creates enclosing rectangles from the scratch, whereas fgExtend modifies existing rectangles in order to make them DRC correct. Another difference is that fgExtend method only tries to fix existing min extension violations of the rectangles under consideration over overlapping shapes. It does not try to enclose the overlapping geometry completely by the rectangles. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

fgFill(PhysicalComponent *comp*, Layer *layer*, Direction *dir*, Coord *divider*, float *grid*=None) – performs a DRC-correct fill operation between this physical component and the *comp* physical component, on the specified *layer* parameter, in the specified *dir* direction. Note that this *dir* value should be one of the primary directions (NORTH, SOUTH, EAST, WEST), or else an exception is raised. The *divider* parameter is used to specify a dividing line, which is used to indicate where the fill operation should take place; this fill operation will only be performed to the left of this dividing line for a vertical direction, or below this dividing line for a horizontal direction. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

Feedback

fgMerge(Layer *resultLayer*=None, ShapeFilter *filter*=None, float *grid*=None) – merges all shapes found on the single layer specified by the *filter* ShapeFilter parameter. If this *filter* ShapeFilter parameter is not specified, then this physical component should only contain a single layer. Note that this method provides the basic functionality of the **fgOr**() method, but for convenience, is specialized for shapes located on a single layer. The resulting merged polygon shapes are generated on the *resultLayer* layer, which will be automatically determined, if this parameter is not specified. In addition, these merged shapes are used to create a Grouping object, which is the return value for this method. An exception will be raised if a single unique layer for this merging operation can not be determined by an analysis of this physical component and the *filter* ShapeFilter parameter. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

fgMinSpacing(Direction *dir*, PhysicalComponent *refComp*, ShapeFilter *filter*=ShapeFilter(), PhysicalComponent *env*=None, bool *align*=True, ShapeFilter *refFilter*=None, dict *options*=None) – returns the minimum “DRC clean” distance between this physical component and another physical component in the specified direction *dir*. The *filter* and *refFilter* ShapeFilter parameters can be used to indicate which layers should be considered in making this minimum distance calculation for these physical components. If the *refFilter* parameter is not specified, then it will be assigned the same value as the *filter* parameter. In addition, if the default *env* parameter is specified, then this additional physical component will be considered during the minimum distance calculation. Note that any text layer shapes will automatically be excluded from consideration when making this minimum distance calculation. If the *align* parameter is True, then this physical component is first aligned with the *refComp* physical component in the given direction *dir*; if this *align* parameter is False, then no alignment is performed. If the *dir* direction is not one of the four primary directions (NORTH, SOUTH, EAST or WEST), then an exception is raised.

The user can modify the behavior of the method by means of *options* parameter.

At the moment, there are eight options available: "infiniteLayers", "envFilter", "ruleIds", "reduceGeometry", "useInitialPlacement", "bboxFilter", "refBboxFilter", and "returnIncrement".

- The "infiniteLayers" option allows you to specify a layer list for enclosing shapes that are supposed to be created later. The effect of this option is similar to activating a Device Context with these layers specified as the device context layers in the techfile. The geometry engine works as if there were shapes on these layers that enclose all the geometry under consideration. This option can be used if you do not have a suitable Device Context defined in the techfile but know that some enclosing shape will be created later that influences the current construction.
- For example, options = {'infiniteLayers' : [Layer('pimp'), Layer('nwell')]}
- The "envFilter" option provides a separate shape filtering for the `env` component.

- The "ruleIds" option is a list of strings (rule IDs) that allows the user to restrict the consideration of spacing rules to only those specified in the list. By default, all spacing rules are considered.
- The “reduceGeometry" option is a boolean flag that can be turned on to optimize the run-time when placing large physical components.
- The "useInitialPlacement" option is a boolean flag that prevents the physical components from coming closer than the current spacing even if design rules allow.
- The "bboxFilter" can be used to specify how to calculate the bounding box of the physical component. By default, *filter* is used to calculate the b-box.
- The "refBboxFilter" option can be used to specify how to calculate the bounding box of the *refComp* component. By default, *refFilter* is used to calculate the b-box.
- The "returnIncrement" option is a boolean flag that changes the meaning of the return value from “minimum DRC correct spacing” to “minimum shift of the component in the placement direction that is necessary to achieve DRC correct spacing.” In this case, the return value can be used later in conjunction with the **moveTowards** method.

fgNot(PhysicalComponent *comp*, Layer *resultLayer*, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *compFilter*=None, float *grid*=None) – performs a logical not operation for this physical component and another physical component, by selecting those polygon areas contained in this physical component which are not contained in the *comp* physical component. This calculation is performed using the layers specified in the *filter* and *compFilter* ShapeFilter objects for each of these physical components. The resulting polygon shapes are generated on the *resultLayer* layer. Note that these shape filters will first be used to merge all layers of the two physical components into geometries on the single *resultLayer* layer, and then the NOT Boolean operation will be performed on the merged geometries of the *resultLayer* for both physical components. In addition, these polygon shapes are used to create a Grouping object, which is the return value for this method. If there are no polygon shapes generated, then this Grouping object will be empty. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

fgOr(PhysicalComponent *comp*, Layer *resultLayer*, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *compFilter*=None, float *grid*=None) – performs a logical or operation for this physical component and another physical component, by selecting those polygon areas which are in either physical component. This calculation is performed using the layers specified in the *filter* and *compFilter* ShapeFilter objects for each of these physical components. The resulting merged polygon shapes are generated on the *resultLayer* layer. Note that these shape filters will first be used to merge all layers of the two physical components into geometries on the single *resultLayer* layer, and then the OR Boolean operation will be performed on the merged geometries of the *resultLayer* for both physical components. In addition, these polygon shapes are used to create a Grouping object,

which is the return value for this method. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

fgPlace(Direction *dir*, PhysicalComponent *refComp*, ShapeFilter *filter*=ShapeFilter(), PhysicalComponent *env*=None, bool *align*=True, ShapeFilter *refFilter*=None, dict *options*=None, float *grid*=None) – performs a `smart place` operation. If the *align* parameter is True, then this physical component is first aligned with the *refComp* physical component in the direction orthogonal to the *dir* direction, using the layer bounding boxes for each component, as defined for each layer in the *filter* and *refFilter* ShapeFilter parameters. Then this physical component is moved in the *dir* direction to have minimal spacing value, according to the current set of design rules (using the technology file currently attached to the current design). If the *align* parameter is False, then no alignment is performed. If the *refFilter* parameter is not specified, then it will be assigned the same value as the *filter* parameter. If the default *env* parameter is specified, then this additional physical component will be considered during the `smart place` operation. Note that any text objects will automatically be excluded from consideration when calculating the distance for this `smart place` operation. If the *dir* direction is not one of the four primary directions (NORTH, SOUTH, EAST or WEST), then an exception is raised. If the *refComp* physical component is the same as this physical component, then an exception is raised.

The user can modify the behavior of the method by means of *options* parameter.

At the moment, there are seven options available: "infiniteLayers", "envFilter", "ruleIds", "reduceGeometry", "useInitialPlacement", "bboxFilter", and "refBboxFilter".

- The "infiniteLayers" option allows you to specify a layer list for enclosing shapes that are supposed to be created later. The effect of this option is similar to activating a Device Context with these layers specified as the device context layers in the techfile. The geometry engine works as if there were shapes on these layers that enclose all the geometry under consideration. This option can be used if you do not have a suitable Device Context defined in the techfile but know that some enclosing shape will be created later that influences the current construction. For example, options = {'infiniteLayers' : [Layer('pimp'), Layer('nwell')]}
- The "envFilter" option provides a separate shape filtering for the `env` component.
- The "ruleIds" option is a list of strings (rule IDs) that allows the user to restrict the consideration of spacing rules to only those specified in the list. By default, all spacing rules are considered.
- The "reduceGeometry" option is a boolean flag that can be turned on to optimize the run-time when placing large physical components.
- The "useInitialPlacement" option is a boolean flag that prevents the physical components from coming closer than the current spacing even if design rules allow.

- The "bboxFilter" can be used to specify how to calculate the bounding box of the physical component. By default, *filter* is used to calculate the b-box.
- The "refBboxFilter" option can be used to specify how to calculate the bounding box of the *refComp* component. By default, *refFilter* is used to calculate the b-box.

If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

fgSize(ShapeFilter *filter*, Coord *sizeValue*, Layer *resultLayer*, float *grid*=None) – expands or shrinks all polygon areas contained in this physical component, according to the *sizeValue* parameter value. If this *sizeValue* parameter is positive, then the polygon edges are expanded outward by that amount. If this *sizeValue* parameter is negative, then the polygon edges are shrunk inward by that amount. The resulting polygon shapes are generated on the *resultLayer* layer. In addition, these polygon shapes are used to create a Grouping object, which is the return value for this method. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

fgXor(PhysicalComponent *comp*, Layer *resultLayer*, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *compFilte*r=None, float *grid*=None) – performs a logical exclusive-or operation between this physical component and another physical component, by selecting those polygon areas which are in either physical component, but not in both physical components. This operation selects all polygon areas which are common to exactly one physical component. This calculation is performed using the layers specified in the *filter* and *compFilter* ShapeFilter objects for each of these physical components. The resulting polygon shapes are generated on the *resultLayer* layer. Note that these shape filters will first be used to merge all layers of the two physical components into geometries on the single *resultLayer* layer, and then the XOR Boolean operation will be performed on the merged geometries of the *resultLayer* for both physical components. In addition, these polygon shapes are used to create a Grouping object, which is the return value for this method. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

getBBox(ShapeFilter *filter*=ShapeFilter()) – returns the bounding box for this physical component on all of the layers defined by the *filter* ShapeFilter parameter. Note that this will be the merged bounding boxes for this physical component on all of these layers.

getCompOwner() – returns the DloGen or Grouping which owns this physical component; by default, this is the current DloGen design object.

getDlo() – returns the Dlo or DloGen design object which contains this physical component.

getLocationPoint(Location *loc*, Point *default*=None) – returns a point corresponding to the *loc* location parameter which can be one of nine standard (pre-defined) locations, or a custom location previously created by calling setCustomLocation(). *default* is an optional

parameter that is returned if the custom location does not exist. When *default* is None, an exception is thrown if the custom location does not exist.

getName() – returns the name for this physical component

getProps() – returns a Python dictionary-like object, which contains any properties which have been defined for this physical component object.

getSpacing(Direction *dir*, PhysicalComponent *refComp*, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *refFilter* = None) – calculates the current distance between this physical component and another *refComp* physical component in the specified *dir* direction. The bounding box for each physical component will be determined using the ShapeFilter which is specified for each physical component. Note that if the *refFilter* ShapeFilter for the reference component is not specified, then the *filter* ShapeFilter for this physical component will be used. The current distance between these two components is determined by measuring the distance between these two bounding boxes.The *dir* direction parameter should be one of the four primary directions (NORTH, SOUTH, EAST or WEST); otherwise, an exception is raised.

keepRelativePosition(IPhysicalComponent *comp*, bool *keep* = True) – keeps the alignment of physical components throughout PyCell evaluation. After calling this method with *keep* parameter = True (default), the relative position of this physical component and *comp* parameter (which can be PhysicalComponent or PhysicalCompRef) is automatically maintained: when one of them moves, another moves accordingly. To terminate this relationship, call this method with *keep* parameter = False. Or to terminate all relationships between this object and all objects, call this method with *keep* parameter = False without *comp* parameter.

makeArray(Coord *dX*, Coord *dY*, unsigned *numRows*, unsigned *numCols*, *baseName*='', *name*='', bool *rowMajor*=True) – creates an array of instances of this physical component; this array is structured as *numRows* rows by *numCols* columns. The

X-distance spacing between components of this array is specified using the *dX* parameter, and the Y-distance spacing between components is specified using the *dY* parameter. Note that these *dX* and *dY* distances are the distances measured from the lower-left corner of one component to the lower-left corner of the next. That is, these distances are absolute distances, not relative offset distances. The *baseName* parameter specifies the base name which is used to name each physical component in the generated array; if the *rowMajor* parameter is True, each array element name has the format "<baseName>_<rowNumber>_<colNumber>". If the *rowMajor* parameter is False, each array element name has the following formats:

- If *numRows* > 1 and *numCols* > 1: <baseName>.<colNumber>.<rowNumber>
- If *numRows* = 1 and *numCols* > 1: <baseName>.<colNumber>
- If *numRows* > 1 and *numCols* = 1: <baseName>.<rowNumber>

The numbering should start from 1. Note that if any sub-components contained within this physical component have been named, then a similar naming convention will be used for each of these sub-components. If the *rowMajor* parameter is True, each sub-component name will have the format "<compName>_<rowNumber>_<colNumber>", where the *baseName* parameter value will be automatically substituted for any occurrence of the physical component name. Also note that the physical component which is used as a basis to create this array of instances will still be present in the layout; if this component should not be present in the generated layout, then it should be explicitly deleted.

mirrorX(Coord *yCoord*=0) – mirrors this physical component about the X coordinate axis. The optional *yCoord* parameter can be used to specify a displacement for the location of the X coordinate axis.

mirrorY(Coord *xCoord*=0) – mirrors this physical component about the Y coordinate axis. The optional *xCoord* parameter can be used to specify a displacement for the location of the Y coordinate axis.

moveBy(Coord *dx*, Coord *dy*) – moves this physical component by *dx* units in the x-direction and *dy* units in the y-direction.

moveTo(Coord *x*, Coord *y*, Location *loc*=Location.CENTER_CENTER, ShapeFilter *filter*=ShapeFilter()) – moves this physical component in the *loc* direction, such that the values specified by the *x* and *y* parameter coordinates become the handle for the bounding box for this physical component object at the given location. This bounding box is determined using the *filter* ShapeFilter parameter. If this physical component does not have any geometry on the layers specified by the *filter* ShapeFilter parameter, then an exception is raised. Any of the nine different location points for a bounding box, or a custom location point set by setCustomLocation(), are valid values for the *loc* parameter.

moveTowards(Direction *dir*, Coord *d*) – moves this physical component by *d* distance units in the direction provided by the *dir* Direction parameter. If the *dir* direction is an opposing direction (such as EAST_WEST), then an exception is raised.

overlaps(Box *box*) – returns True if this physical component overlaps the bounding box object being passed as the *box* parameter, and returns False otherwise. Note that if this physical component only touches the *box* object, True will be returned.

place(Direction *dir*, PhysicalComponent *refComp*, Coord *distance*, ShapeFilter *filter*=ShapeFilter(), bool *align*=True, ShapeFilter *refFilter*=None) – explicitly places this physical component by moving it in the given direction *dir*, so that the two components are spaced *distance* units apart. If the *filter* parameter is not specified (the default value), then the bounding boxes for both physical components will be calculated for all geometries on all layers. In this case, the resulting bounding box will be determined as the merged bounding boxes from each layer on which the physical component has geometry. Otherwise, the layer bounding boxes for each component will be used, as defined for each layer in the *filter* ShapeFilter parameter. If the *refFilter* parameter is not specified, then it will be assigned the same value as the *filter* parameter. Note that unlike

the **fgMinSpacing**() and **fgPlace**() methods, this **place**() method does not automatically exclude text layer shapes when determining the merged bounding box. Thus, it may be necessary to specifically exclude the text layer in the *filter* and/or *refFilter* parameter values. If the *align* parameter is True, then this physical component is first aligned with the *refComp* physical component in the direction orthogonal to the *dir* direction; if this *align* parameter is False, then no alignment is performed. If the *dir* direction is not one of the four primary directions (NORTH, SOUTH, EAST or WEST), then an exception is raised. If the *refComp* physical component is the same as this physical component, then an exception is raised.

rotate90(Point *origin*=None) – rotates this physical component by 90 degrees, in a counter-clockwise direction. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

rotate180(Point *origin*=None) – rotates this physical component by 180 degrees, in a counter-clockwise direction. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

rotate270(Point *origin*=None) – rotates this physical component by 270 degrees, in a counter-clockwise direction. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

setCustomLocation(Location *loc*, Point *point*) – associates *loc* custom location for this physical component with a position specified by *point* parameter.

setName(string *name*) – sets the name for this physical component

snap(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) – snaps both the X and Y coordinates of the lower-left vertex point of the bounding box for this physical component to the grid points defined by the *grid* parameter. This bounding box will be determined using the layers specified by the *filter* ShapeFilter parameter. The *snapType* parameter is used to specify the snapping type (CEIL, FLOOR, ROUND or TRUNC) which should be used; if this *snapType* parameter is not specifed, then the snapping type defined by the *grid* Grid object will be used by default. This method is very useful to ensure that physical components are aligned to manufacturing grid points. If the bounding box determined by the *filter* ShapeFilter parameter is inverted, then an exception is raised.

snapX(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) – snaps only the X coordinate of the lower-left vertex point of the bounding box for this physical component to grid points defined by the *grid* parameter. If the bounding box determined by the *filter* ShapeFilter parameter is inverted, then an exception is raised.

snapY(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) – snaps only the Y coordinate of the lower-left vertex point of the bounding box for this physical component to grid points defined by the *grid* parameter. If the bounding box determined by the *filter* ShapeFilter parameter is inverted, then an exception is raised.

Feedback

snapTowards(Grid *grid*, Direction *dir*, ShapeFilter *filter*=ShapeFilter()) – snaps the coordinates of the lower-left vertex point of the bounding box for this physical component to the grid points defined by the *grid* parameter. This bounding box will be determined using the layers specified by the *filter* ShapeFilter parameter. The *dir* Direction parameter is used to specify the one-dimensional direction in which the snapping operation should take place. If the bounding box determined by the *filter* ShapeFilter parameter is inverted, then an exception is raised. If the *dir* direction is not one of the four primary directions (NORTH, SOUTH, EAST or WEST), then an exception is raised.

Note that the above **snap**() methods will only change the location of this physical component; they will not change the length, width or size of any member shapes.

transform(Transform *trans*) – applies the transform *trans* passed as a parameter to this physical component.

Many of these PhysicalComponent methods concerned with the placement of design objects (such as the **place**(), **fgPlace**(), **abut**(), **align** and **move** methods) have a physical component as one of the parameters. Note that these parameters can either be physical components, or any class objects which are derived from the PhysicalComponent class. In particular, these parameters can be Grouping objects, in which case all of the physical components within these Grouping objects will be operated upon by these methods.

Attributes:

In addition to these methods for this PhysicalComponent object, there is also an attribute (or property) defined for this object, described as follows:

props – the properties and values which have been defined for this physical component

Note that this props attribute value will be the same value as returned by the "**getProps**()"method.

Shape Classes

Shape

Description: This is the base class for all shape objects defined for the Python API. All of the basic geometric shape objects (such as rectangles, polygons and lines) are derived from this base Shape class. Any shape object is defined on a single layer in the design. Note that this Shape class is an abstract class, so that all of the methods for this Shape class are used by the actual class objects which are derived from this base Shape class. Note that any Shape object can have an optional name, so that shapes in a design can either be named or unnamed.

Feedback

Methods:

getBBox(ShapeFilter *filter*=ShapeFilter()) – returns the bounding box for this Shape. Any layers specified in the ShapeFilter *filter* parameter are used to perform this bounding box calculation. Note that if this Shape object does note exist on any of the specified layers, then a large inverted bounding box will be returned.

getColorMask() – returns coloring information (string *anchor*, string *color*) for this Shape for double and triple patterning. String *anchor* = `"ANCHORED" | "UNANCHORED`. String *color* = `"GRAY"` | "MASKCOLOR*n*", where *n* is an integer from 1 to 127. Getting color for Dot, Text, and TextDisplay always returns (`"UNANCHORED"`, None).

getLayer() – returns the current Layer for this Shape; this is typically the Layer on which this Shape has been drawn.

getName() – returns any optional name which has been assigned to this Shape; if no name has been assigned to this Shape, then None is returned.

getNet() – returns any net to which this Shape has been assigned.

getPin() – returns any pin to which this Shape has been assigned.

setColorMask(string *anchor*, string *color*) – sets coloring information for this Shape for double and triple patterning. String *anchor* = `ANCHORED | "UNANCHORED`. String *color* = `GRAY` | "MASKCOLOR*n*", where *n* is is an integer from 1 to 127. Setting color for Dot, Text, and TextDisplay is ignored.

setLayer(Layer *layer*) – sets the Layer for this Shape.

setName(string *name*) – sets the optional name for this Shape. If this *name* is being used as the name of a Shape in the current design, then an exception is raised. If the *name* parameter is None (or an empty string), then any name will be removed from this Shape.

setNet(Net *net*) – sets the Net for this Shape, so that this Shape is assigned to a net in the current design. Note that if the *net* parameter is set to None, then this Shape is removed from any assigned net.

Inherited Methods: This Shape class inherits all methods which are defined for the PhysicalComponent class.

Attributes:

In addition to these methods for this Shape class, there are also attributes (or properties) defined for this Shape class, described as follows:

bbox – the bounding box for this Shape

layer – the Layer associated with this Shape

name – the name of this Shape

Feedback

Note that these **layer** and **name** attributes are also settable, so that they can be used in the same way as the "**getLayer**()" and "**setLayer**()" methods, and the "**getName**()" and "**setName**()" methods. The **bbox** attribute is read-only, and can be used in the same way as the "**getBBox**()" method.

In addition to all of the methods which are provided for the PhysicalComponent and Shape abstract base classes, there are specific methods which have been defined for each of these different derived shape objects. These shape classes and their associated methods are described in the following sections.

Arc

Description: This Arc class is used to represent arc shapes in physical layout designs. An Arc shape is a segment of the circumference of an ellipse, so that this Arc object would be defined by means of a bounding box for this ellipse, along with the starting and stopping angles for the arc. These three parameters (ellipse bounding box, start angle and stop angle) are used to define an Arc shape. Note that the stop angle minus the start angle must be less than or equal to pi radians (180 degrees).

Creation:

Arc(Layer *layer*, Box *box*, double *startAngle*=0.0, double *endAngle*=0.0, Box *arcBox*=None) – creates an arc, which is defined by either the ellipse bounding box and the starting and stopping angles of the arc, or by calculating the overlap or interception of the *box* and *arcBox*. The *layer* parameter is a Layer object, while the *box* bounding box parameter is a Box object. The *startAngle* and *endAngle* for the arc are defined in radians, where *endAngle* is greater than *startAngle*, and the difference between these two angles is less than or equal to pi. The *arcBox* parameter is a Box object and if it is given, the *startAngle* and *endAngle* parameters are ignored. If the ellipse bounding box is invalid, or if the start and stop angles are invalid, then an exception is raised.**Methods:**

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this arc object.

getEllipseBBox() – returns the bounding box for the ellipse which defines this arc.

getStartAngle() – returns the start angle of this arc in radians.

getStopAngle() – returns the stop angle of this arc in radians.

setEllipseBBox(Box *box*) – sets the bounding box for the ellipse which defines this arc. Note that the current start angle and stop angle for this arc are left unchanged. If the ellipse bounding box is invalid, then an exception is raised.

setStartAngle(double *startAngle*) – sets the start angle (specified in radians) for this arc. Note that the current ellipse bounding box and stop angle for this arc are left unchanged. If the resulting start and stop angles are invalid, then an exception is raised.

Feedback

setStopAngle(double *stopAngle*) – sets the stop angle (specified in radians) for this arc. Note that the current ellipse bounding box and start angle for this arc are left unchanged. If the resulting start and stop angles are invalid, then an exception is raised.

Inherited Methods: This Arc class inherits all methods defined for the Shape class, as well as the PhysicalComponent class.

Examples:

```
arc1 = Arc(Layer('metal1'), Box(0,0,10.0,10.0), 0.25, 1.25)
arc1.getLayer()        # returns Layer('metal1')
arc1.getBBox()
# perform mirroring operations
arc1.mirrorX()
arc1.getBBox()
arc1.mirrorX()
arc1.getBBox()         # back to original position
# change arc shape
arc1.setEllipseBBox(Box(0,0,11,11))
arc1.setStartAngle(0.0)
arc1.setStopAngle(1.5)
arc1.getEllipseBBox()  # returns Box(0,0,11,11)
arc1.getStartAngle()   # returns 0.0
arc1.getStopAngle()    # returns 1.5
# create cloned copy of this arc shape
arc2 = arc1.clone()
```

Donut

Description: This Donut class is used to represent donut shapes used in analog layout and physical designs. This Donut object would be defined by means of a center point, along with the inner and outer radius values.

Creation:

Donut(Layer *layer*, Point *center*, Coord *radius*, Coord *holeRadius*, double *startAngle*=0.0, double *endAngle*=360.0) – creates a donut, which is defined by the center point, the radius of the ellipse (outer radius), and the radius of the hole (inner radius). The *layer* parameter is a Layer object, while the *centerPoint* parameter is a Point object, and the *outerRadius* and *innerRadius* parameters are coordinate values. The *startAngle* and *endAngle* for the donut are defined in degrees, and the difference between these two angles is less than or equal to 360. If the two radius values are invalid, or if the start and end angles are invalid, then an exception is raised.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this donut object

getCenter() – returns the center point for this donut.

Feedback

getEndAngle() – returns the end angle of this donut in degrees.

getHoleBBox() – returns the bounding box for the hole defined by the center point and the hole radius (inner radius) for this donut.

getHoleRadius() – returns the value of the hole radius (inner radius) for this donut.

getRadius() – returns the radius (outer radius) value for this donut.

getStartAngle() – returns the start angle of this donut in degrees.

setAngles(double *startAngle*, double *endAngle*) – sets the start and end angles (specified in degrees) for this donut. Note that the current center point, radius, and hole radius for this donut are left unchanged. If the resulting start and end angles are invalid, then an exception is raised.

setCenter(Point *center*) – sets the center point for this donut.

setEndAngle(double *endAngle*) – sets the end angle (specified in degrees) for this donut. Note that the current center point, radius, hole radius, and start angle for this donut are left unchanged. If the resulting start and end angles are invalid, then an exception is raised.

setHoleRadius(Coord *holeRadius*) – sets the hole radius (inner radius) for this donut. If the resulting radius values are invalid for this donut, then an exception is raised.

setRadius(Coord *radius*) – sets the radius (outer radius) value for this donut. If the resulting radius values are invalid for this donut, then an exception is raised.

setStartAngle(double *startAngle*) – sets the start angle (in degrees) for this donut. Note that the current center point, radius, hole radius, and end angle for this donut are left unchanged. If the resulting start and end angles are invalid, then an exception is raised.

Inherited Methods: This Donut class inherits all methods defined for the Shape class, as well as the PhysicalComponent class.

Examples:

```
donut1 = Donut(Layer('metal1'), Point(50.0,50.0), 40.0, 20.0)
donut1.getLayer()          # returns Layer('metal1')
donut1.getBBox()
donut1.getHoleBBox()
# perform various rotation and mirroring operations
donut1.mirrorX()
donut1.getBBox()
donut1.mirrorX()
donut1.getBBox()           # back to original position
donut1.rotate90()
donut1.rotate180()
donut1.rotate90()
donut1.getBBox()           # back to original position
# change donut shape
donut1.setCenter(Point(0,0))
```

Feedback

```
donut1.setHoleRadius(15.0)
donut1.setRadius(45.0)
donut1.getCenter()          # returns Point(0,0)
donut1.getHoleRadius()      # returns 15.0
donut1.getRadius()          # returns 45.0
# create cloned copy of this donut shape
donut2 = donut1.clone()
```

Dot

Description: This Dot class is used to represent dot shapes used in physical layout designs. This Dot object would be defined by means of an origin point, along with possible width and height values.

Creation:

Dot(Layer *layer*, Point *origin*, Coord *width* = 0, Coord *height* = 0) – creates a dot at the given origin, with the given height and width values (which have zero as default values). The *layer* parameter is a Layer object, the *origin* parameter should be a Point object, while the *width* and *height* parameter values are coordinate values.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this dot object

getHeight() – returns the height of this dot.

getOrigin() – returns the point for the origin of this dot.

getWidth() – returns the width of this dot.

setHeight(Coord *height*) – sets the height value for this dot.

setOrigin(Point *origin*) – sets the values of the origin point to *origin* for this dot.

setWidth(Coord *width*) – sets the width value for this dot.

Inherited Methods: This Dot class inherits all methods defined for the Shape class, as well as the PhysicalComponent class.

Examples:

```
dot1 = Dot(Layer('metal1'), Point(5,10))
dot1.getLayer()         # returns Layer('metal1')
dot1.getBBox()
# perform mirroring operations
dot1.mirrorX()
dot1.getBBox()
dot1.mirrorX()
dot1.getBBox()          # back to original position
# change dot shape
```

Feedback

```
dot1.getOrigin()        # returns Point(5,10)
dot1.getHeight()        # returns 0.0
dot1.getWidth()         # returns 0.0
dot1.setOrigin(Point(5.0, 8.0))
dot1.setHeight(1.0)
dot1.setWidth(0.5)
dot1.getOrigin()        # returns Point(5,8)
dot1.getHeight()        # returns 1.0
dot1.getWidth()         # returns 0.5
# create cloned copy of this dot shape
dot2 = dot1.clone()
```

Ellipse

Description: This Ellipse class is used to represent circular shapes used in physical layout designs. This Ellipse object would be defined by means of a bounding box, which effectively specifies the values for the two radii of the ellipse shape. In the case where these two radii values are the same, then the bounding box would be a square, and a circle shape would be described.

Creation:

Ellipse(Layer *layer*, Box *box*, double *startAngle*=0.0, double *endAngle*=360.0) – creates an ellipse having the *box* parameter as its bounding box; this bounding box defines the dimensions of the ellipse. (Note that a circle object is the special case of the ellipse where the bounding box rectangle is a square). The *layer* parameter is a Layer object, while the *box* parameter is the bounding box defined by a Box object. The *startAngle* and *endAngle* for the ellipse are defined in degrees, and the difference between these two angles is less than or equal to 360. If the bounding box is invalid, or if the start and stop angles are invalid, then an exception is raised.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this ellipse object

genPolygonPoints(Box *box*, unsigned *numPoints*, Coord *gridSize*) – generates a list of points, which can be used to approximate this ellipse using straight-line segments. The returned list of polygon points can then be used to construct a polygon which will approximate this ellipse shape. The *box* parameter is the box which defines the dimensions of an ellipse, while the *numPoints* parameter is the number of points which should be generated. The *gridSize* parameter specifies the size of the Grid which should be used when the generated polygon points are `snapped` to the nearest grid point. This list of points is returned as a PointList object.

Note that in some cases, it is possible for the returned PointList to have some coincident and/or collinear points. For example, this can occur when there are a large number of points, and the specified *gridSize* is small. Thus, when using this list of points to construct

a Polygon object, it is best to ensure that any such coincident and/or collinear points are removed. This can be done by calling the PointList "**compress**()" method.

Note that this method is a static method, so that it can be called without explicitly creating an Ellipse object. This makes it more convenient to approximate an ellipse using

a set of polygon points, without first creating an ellipse, and then destroying it after this list of polygon points has been generated. Also, note that this method can be used to generate the polygon points which approximate a circle; this occurs when the *box* rectangle parameter is a square.

getStartAngle() – returns the start angle of this ellipse in degrees.

getEndAngle() – returns the end angle of this ellipse in degrees.

setAngles(double *startAngle*, double *endAngle*) – sets the start and end angles (specified in degrees) for this ellipse. Note that the current bounding box for this ellipse is left unchanged. If the resulting start and end angles are invalid, then an exception is raised.

setBBox(Box *box*) – sets the bounding box for this ellipse. If the bounding box is invalid, then an exception is raised.

setEndAngle(double *endAngle*) – sets the end angle (specified in degrees) for this ellipse. Note that the current bounding box and start angle for this ellipse are left unchanged. If the resulting start and end angles are invalid, then an exception is raised.

setStartAngle(double *startAngle*) – sets the start angle (specified in degrees) for this ellipse. Note that the current bounding box and end angle for this ellipse are left unchanged. If the resulting start and end angles are invalid, then an exception is raised.

Inherited Methods: This Ellipse class inherits all methods defined for the Shape class, as well as the PhysicalComponent class.

Examples:

```
ellipse1 = Ellipse(Layer('metal1'), Box(0,0,10.0,20.0))
ellipse1.getLayer()        # returns Layer('metal1')
ellipse1.getBBox()
# perform various mirroring and rotation operations
ellipse1.mirrorY()
ellipse1.getBBox()
ellipse1.mirrorY()
ellipse1.getBBox()         # back to original position
ellipse1.rotate90()
ellipse1.rotate180()
ellipse1.rotate90()
ellipse1.getBBox()         # back to original position
# create cloned copy of this ellipse shape
ellipse2 = ellipse1.clone()
# now approximate an ellipse using a set of polygon points
# construct list of polygon points
```

```
box1 = Box(0, 0, 1.5, 2.5)
points = Ellipse.genPolygonPoints(box1, 200, 0.005)
# remove any extra points
if points.hasExtraPoints():
  points.compress()
# construct the polygon
polygon1 = Polygon(Layer('metal1'), points)
```

Line

Description: This Line class is used to represent a linear shape used in physical layout designs. This Line object would be defined by a list of points which would be used to define this Line object.

Creation:

Line(Layer *layer*, PointList *points*) – creates a line using the *points* list of points which is passed as a parameter. Note that any coincident (duplicate) or collinear points will be removed from this list of points, before creating the Line object. The *layer* parameter is a Layer object, while the *points* parameter is a Python PointList object, which is essentially a Python list of Point objects. Note that there must be at least two distinct points, after any coincident or collinear points are removed from this list of points, or else an exception is raised.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this Line object

getNumPoints() – returns the number of points contained in this line.

getPoints() – returns the points contained in this line, as a Python PointList object.

setPoints(PointList *points*) – sets the points for this line. Note that any coincident or collinear points will first be removed from this *points* parameter list of points; there should be at least two distinct remaining points, or else an exception is raised.

Inherited Methods: This Line class inherits all methods defined for the Shape class, as well as the PhysicalComponent class.

Attributes:

In addition to these methods for the Line class, there is also an attribute (or property) defined for the Line class, described as follows:

points – the list of points which define this line

Note that this **points** attribute will return the same value as the "**getPoints**()" method, and can also be used just the same as the "**setPoints**()" method to set the list of points for the Line object.

Feedback

Examples:

```
line1 = Line(Layer('metal1'), [Point(0,0), Point(10.0,20.0)])
line1.getLayer()       # returns Layer('metal1')
line1.getBBox()
# perform mirroring operations
line1.mirrorX()
line1.getBBox()
line1.mirrorX()
line1.getBBox()        # back to original position
# change the points which define this line shape
line1.getNumPoints()   # returns 2
line1.getPoints()
pts = [Point(0,0), Point(5,6), Point(7,8), Point(8,9)]
line1.setPoints(pts)
line1.getNumPoints()   # returns 4
line1.getPoints()
# create cloned copy of this line shape
line2 = line1.clone()
```

Path

Description: This Path class is used to represent path shapes which are used in physical layout designs. This Path object would be defined by means of a list of points which define the path shape, along with a possible width for this Path shape. In addition, a path style for the beginning and ending points can be specified.

Creation:

Path(Layer *layer*, Coord *width*, PointList *points*, PathStyle *style*=PathStyleTRUNCATE) – creates a path object, using the *points* list of points to define the path. The *width* parameter is used to specify the width of this path. The *layer* parameter is a Layer object, the *width* parameter is an integer value, and the *points* parameter is a Python list of Point objects. The *style* parameter is used to specify a path style for the beginning and ending points of the Path. These path styles are specified using the PathStyle class enumerated values: TRUNCATE, EXTEND, ROUND or VARIABLE. Any coincident or collinear points should first be removed, by calling the **compress**() method for the PointList class on the *points* PointList parameter. Note that if there are any coincident or collinear points, or if there are less than two points, then an exception is raised.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this Path object

getBeginExt() – returns the beginning extension value for this path. Note that only a path with the VARIABLE path style can have a non-zero value.

Feedback

getBoundary(bool *usePathOrder*=False) – returns a list of the points which define the boundary for this path. These points can then be used to construct a Polygon shape, which represents the boundary for this path as a polygon outline. This can be used to check for overlap between this path and other design objects. Note that the list of boundary points returned by this method does not use a specific ordering. If a specific ordering is required, then *usePathOrder* should be set to True, and the boundary points will use the same ordering as the points which define the path. For example, the first boundary point will be the boundary point which is to the right of the first path point.

getEdgeLength(int *index*) – returns the length from the Path object for the specified segment number.

getEndExt() – returns the ending extension value for this path. Note that only a path with the VARIABLE path style can have a non-zero value.

getNumPoints() – returns the number of points for this path.

getPoints() – returns the list of points that define this path. This list of points is returned as a PointList object.

getStyle() – returns the end point style for this path, as a string having one of the four PathStyle enumerated values TRUNCATE, EXTEND, ROUND or VARIABLE. The TRUNCATE style has no extension beyond the beginning and ending points for the path, while the EXTEND style has a one-half width extension beyond the beginning and ending points. The ROUND style has a circular extension, with a radius of one half width beyond the beginning and ending points, while the VARIABLE style allows for differing beginning point and ending point extension values.

getWidth() – returns the width of this path

isOrthogonal() – returns True, if all the points in this path are orthogonal, and returns False, otherwise. Note that this path is not considered to be a closed shape, when the points in this path are checked.

setBeginExt(Coord *beginExt*) – sets the beginning extension value for this path. Note that only a path with the VARIABLE path style can have a non-zero value; for other path styles, an exception is raised.

setEndExt(Coord *endExt*) – sets the ending extension value for this path. Note that only a path with the VARIABLE path style can have a non-zero value; for other path styles, an exception is raised.

setPoints(PointList *points*) – sets the list of points which defines this path to the *points* list of points. Note that if this *points* list of points contains any coincident or collinear points, or if this list of points has less than two points, then an exception is raised.

setStyle(string *style*) – sets the end point style for this path. The *style* parameter should be one of the four PathStyle enumerated values TRUNCATE, EXTEND, ROUND or VARIABLE. The TRUNCATE style has no extension beyond the beginning and ending

points for the path, while the EXTEND style has a one-half width extension beyond the beginning and ending points. The ROUND style has a circular extension, with a radius of one half width beyond the beginning and ending points, while the VARIABLE style allows for differing beginning point and ending point extension values.

setWidth(Coord *width*) – sets the width value for this path

Inherited Methods: This Path class inherits all methods defined for the Shape class, as well as the PhysicalComponent class.

Attributes:

In addition to these methods for the Path class, there is also an attribute (or property) defined for the Path class, described as follows:

points – the list of points which define this path

Note that this **points** attribute will return the same value as the "**getPoints**()" method, and can also be used like the "**setPoints**()" method to set the list of points for the Path object.

Examples:

```
p1 = [Point(0,0), Point(1,0), Point(1,1), Point(2,1)]
path1 = Path(Layer('metal1'), width=0.05, points=p1)
path1.getLayer()    # returns Layer('metal1')
path1.getBBox()
# perform mirroring operations
path1.mirrorY()
path1.getBBox()
path1.mirrorY()
path1.getBBox()     # back to original position
# change path shape
path1.setWidth(0.5)
path1.getWidth()
path1.isOrthogonal()
path1.getNumPoints()
path1.getPoints()
p1 = [Point(0,0), Point(1,0), Point(1,1), Point(2,1), Point(2,2)]
path1.setPoints(p1)
path1.getNumPoints()   # returns 5
path1.getPoints()
path1.isOrthogonal()   # returns True
# use boundary points for this path to create polygon shape
p2 = path1.getBoundary()
path1.setWidth(0.2)
poygon1 = Polygon(Layer('metal2'), p2)
# change end point style for this path shape
path1.getStyle()       # returns PathStyle.TRUNCATE
path1.setStyle(PathStyle.VARIABLE)
path1.setBeginExt(0.5)
path1.setEndExt(0.6)
path1.getStyle()       # returns PathStyle.VARIABLE
```

Feedback

```
path1.getBeginExt()     # returns 0.5
path1.getEndExt()       # returns 0.6
# create cloned copy of this path shape
path2 = simpPath1.clone()
```

PathSeg

Description: This PathSeg class is used to represent path segment shapes used in physical layout designs. This PathSeg object would be defined by means of a starting point and an ending point. The starting point and ending point should define a horizontal, vertical or diagonal path segment. The horizontal or vertical segment represents part of an orthogonal Manhattan route, while the diagonal segment represents part of a diagonal route. This path segment shape is typically used for routing applications.

Creation:

PathSeg(Layer *layer*, Point *begin*, Point *end*) – creates a segment of a path using the two points which are passed as the starting point and the ending point for the path segment. The *layer* parameter is a Layer object, while *begin* and *end* parameters are Point objects. The *begin* point and the *end* point for this path segment should define either a horizontal, vertical or diagonal segment; otherwise, an exception is raised. Note that when a path segment is created, it will initially have zero width; the **setWidth**() method should be used to set the width of the path segment to any desired non-zero value.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this PathSeg object

getBoundary( ) – returns a list of the points which define the boundary for this path segment. These points can then be used to construct a Polygon shape, which represents the boundary for this path segment as a polygon outline.

getPoints() – returns the two points which define this path segment; these two points are returned as a tuple, such as `(beginPoint, endPoint)`.

getWidth() – returns the width of this path segment

isOrthogonal() – returns True, if these two points for the path segment are orthogonal, and returns False, otherwise.

setPoints(Point *beginPoint*, Point *endPoint*) – sets the two points which define this path segment. Note that these two points should define either a horizontal, vertical or diagonal segment; otherwise, an exception is raised.

setWidth(Coord *width*) – sets the width value for this path segment

Inherited Methods: This PathSeg class inherits all methods defined for the Shape class, as well as the PhysicalComponent class.

Feedback

Examples:

```
# create diagonal routing path segment
pathseg1 = PathSeg(Layer('metal1'), Point(5,5), Point(8,8))
pathseg1.getLayer()       # returns Layer('metal1')
pathseg1.getBBox()
# perform mirroring operations
pathseg1.mirrorY()
pathseg1.getBBox()
pathseg1.mirrorY()
pathseg1.getBBox()        # back to original position
# change path segment to Manhattan routing path segment
pathseg1.isOrthogonal()   # returns False
pathseg1.setPoints(Point(5,5), Point(5,8))
pathseg1.isOrthogonal()   # returns True
pathseg1.getPoints()
pathseg1.getWidth()       # returns 0.0
pathseg1.setWidth(0.5)
pathseg1.getWidth()       # returns 0.5
# create cloned copy of this path segment shape
pathseg2 = pathseg1.clone()
```

Polygon

Description: This Polygon class is used to represent polygon shapes which are used in physical layout designs. This Polygon object would be defined by means of a list of points which define the vertices for this Polygon object.

Creation:

Polygon(Layer *layer*, PointList *points*) – creates a polygon using the *points* list of points which is passed as a parameter. Note that this list of points should contain at least three points. It should also not contain any coincident (duplicate) or collinear points. If any of these conditions are violated, then an exception is raised. Any coincident or collinear points should first be removed, by calling the **compress**() method for the PointList class on the *points* PointList parameter. The *layer* parameter is a Layer object, while the *points* parameter is a Python list of Point objects.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this Polygon object

getEdgeLength(int *index*) – returns the length from the Polygon object for the specified segment number.

getNumPoints() – returns the number of points that define this polygon

getPoints() – returns the list of points that define this polygon. This list of points is returned as a PointList object.

Feedback

isOrthogonal() – returns True, if all the vertices in this polygon are orthogonal, and returns False, otherwise. Note that this polygon is not considered to be a closed shape, when the vertices of this polygon are checked.

setPoints(PointList *points*) – sets the list of points which defines this polygon to the *points* list of points. Note that if this *points* list of points contains any coincident or collinear points, or if this list of points has less than three points or if the first and last points are the same, then an exception is raised.

subdivide(unsigned *numPoints*, bool *verticalChop*=False) – subdivides this polygon into multiple sub-polygons, each of which has fewer than *numPoints* points. The *verticalChop* parameter specifies whether to subdivide the polygon object in the horizontal (False) or vertical (True) direction. Note that hese sub-polygons may be abutted at common edges, but will not be overlapped. In addition, these sub-polygon shapes are used to create a Grouping object, which is the return value for this method.

If this polygon can not be sub-divided, then this polygon will be used as the return value. This method can be useful when a polygon has a large number of vertices, as many utility programs which generate GDS2 data can not output polygons having too many vertices. Note that this polygon will not be destroyed when this method is used; if it is desired to destroy the original polygon after sub-division, then it should be explicitly destroyed. Also note that this method will only work correctly when the vertices of this polygon are orthogonal or meet at 45 degree angles.

Inherited Methods: This Polygon class inherits all methods defined for the Shape class, as well as the PhysicalComponent class.

Attributes:

In addition to these methods for the Polygon class, there is also an attribute (or property) defined for the Polygon class, described as follows:

points – the list of points which define this polygon

Note that this **points** attribute will return the same value as the "**getPoints**()" method, and can also be used just the same as the "**setPoints**()" method to set the list of points for the Polygon object.

Examples:

```
polygon1 = Polygon(Layer('metal1'), [Point(10.0,20.0),
 Point(30.0,40.0), Point(50.0,20.0)])
polygon1.getLayer()            # returns Layer('metal1')
polygon1.getNumPoints()        # returns 3
polygon1.getPoints()
ptList = [Point(10,0), Point(0,10), Point(-10,0), Point(0,-10)]
polygon1.setPoints(ptList)
polygon1.getNumPoints()        # returns 4
polygon1.getPoints()
polygon1.isOrthogonal()        # returns False
```

Feedback

```
# create cloned copy of this polygon shape
polygon2 = polygon1.clone()
```

Rect

Description: This Rect class is used to represent rectangular shapes used in physical layout designs. This Rect object is defined by means of a bounding box.

Creation:

Rect(Layer *layer*, Box *box*) – creates a rectangle having the *box* parameter as a bounding box; this bounding box defines the dimensions of the rectangle. The *layer* parameter is a Layer object, while the *box* parameter is the bounding box defined by a Box object. If the bounding box is invalid, then an exception is raised.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this Rect object.

The following three methods are used to generate a set of equally sized rectangles inside a specified Box object. These methods would typically be used to generate via fill rectangles within a specified region in a layout design. Note that the dimensions of these rectangles are specified using the *width* and *height* parameters, while the spacing between these rectangles are specified using the pair of *space* parameters. The set of generated rectangles will be added to the grouping specified by the *group* parameter. The return value for each of these methods is the number of rectangles which are generated.

expand(Coord *coord*) – expands this rectangle by the *coord* coordinate value in each direction

expandDir(Direction *dir*, Coord *coord*) – expands this rectangle by the *coord* coordinate value in the specified *dir* direction.

expandToGrid(Grid *grid*, Direction *dir*=None) – expands the edges of this rectangle in the specified direction *dir* to align with the nearest grid point. The *grid* parameter is used to specify the Grid object which should be used to determine the nearest grid point value.

If the *dir* parameter is not specified, then all four edges of this rectangle will be expanded to grid. Note that this method will always expand the edges of this rectangle towards the outside, so that the SnapType for the specified *grid* will be ignored.

Note that the following three methods are defined as static methods, so that they can be called without explicitly creating a Rect object. For example, they can be directly called using the syntax Rect.fillBBoxWithRects(*layer*, *box*).

fillBBoxWithRects(Layer *layer*, Box *box*, Coord *width*=None, Coord *height*=None, Coord *spaceX*=None, Coord *spaceY*=None, Gap *gapStyle*=GapStyle.MINIMUM, Grouping *group*=None) – fills the specified *box* with equally sized rectangles constructed on the

specified *layer*. If the *width* or *height* parameter is not specified, then the minimum width design rule value for the specified *layer* will be used. Similarly, if the *spaceX* or *spaceY* parameter is not specified, then the minimum spacing design rule value for the specified *layer* will be used. The *gapStyle* parameter can be used to control how the spacing between rectangles should be distributed (as one of MINIMUM, DISTRIBUTE or MIN_CENTER). The *group* parameter can be used to store the fill patterns which are generated by this method; if this grouping parameter is not specified, then no grouping will be used.

fillDiagBoxWithRects(Layer *layer*, Box *diagBox*, Coord *width*=None, Coord *height*=None, Coord *space45*=None, Coord *space135*=None, GapStyle *gapStyle*=GapStyle.MINIMUM, Grouping *group*=None) – fills the specified diagonal *box* with equally sized rectangles constructed on the specified *layer*. Note that the dimensions and spacing values for these rectangles are specified using diagonal coordinate values; orthogonal coordinate values can be converted to diagonal coordinate values using the Point class **toDiagAxes**() method. If the *width* or *height* parameter is not specified, then the minimum width design rule value for the specified *layer* will be used. Similarly, if the *space45* or *space135* parameter is not specified, then the minimum spacing design rule value for the specified *layer* will be used. The *gapStyle* parameter can be used to control how the spacing between rectangles should be distributed (as one of MINIMUM, DISTRIBUTE or MIN_CENTER). The *group* parameter can be used to store the fill patterns which are generated by this method; if this grouping parameter is not specified, then no grouping will be used.

fillDiagBoxWithDiagRects(Layer *layer*, Box *diagBox*, Coord *width45*, Coord *height135*, Coord *space45*, Coord *space135*, GapStyle *gapStyle*=GapStyle.MINIMUM, Grouping *group*=None) – fills the specified diagonal *box* with equally sized diagonal rectangles constructed on the specified *layer*. Note that the spacing values for these diagonal rectangles are specified using diagonal coordinate values; orthogonal coordinate values can be converted to diagonal coordinate values using the Point class **toDiagAxes**() method. The *gapStyle* parameter can be used to control how the spacing between these rectangles should be distributed (as one of MINIMUM, DISTRIBUTE or MIN_CENTER). The *group* parameter can be used to store the fill patterns which are generated by this method; if this grouping parameter is not specified, then no grouping will be used.

getBottom() – returns the coordinate value for the bottom of this rectangle

getCoord(Direction *dir*) – returns the coordinate of this rectangle in the specified *dir* direction. For example, if the *dir* direction value is NORTH, then the top coordinate value for this rectangle will be returned. In the case of a full-line direction (NORTH_SOUTH or EAST_WEST), the center coordinate value will be returned. This *dir* direction should be one dimensional; otherwise, an exception is raised.

getHeight() – returns the height of this rectangle

getLeft() – returns the coordinate value for the left side of this rectangle

Feedback

getRight() – returns the coordinate value for the right side of this rectangle

getTop() – returns the coordinate value for the top of this rectangle

getWidth() – returns the width of this rectangle

setBBox(Box *box*) – sets the bounding box to the *box* parameter for this rectangle.

setBottom(Coord *v*) – sets the coordinate value for the bottom of this rectangle

setCoord(Direction *dir*, Coord *coord*) – sets this rectangle to have the coordinate value in the specified direction. For example, if the direction is NORTH, then the top coordinate value will be set to the specified value. If the *dir* direction is not one of the four primary directions (NORTH, SOUTH, EAST, WEST), then an exception is raised.

setLeft(Coord *v*) – sets the coordinate value for the left side of this rectangle

setRight(Coord *v*) – sets the coordinate value for the right side of this rectangle

setTop(Coord *v*) – sets the coordinate value for the top of this rectangle

Note that these **setBottom**(), **setCoord**(), **setLeft**(), **setRight**() and **setTop**() methods will check to see if the resulting rectangle will have zero area; if so, then the rectangle will be extended by one unit in the opposite direction to ensure the resulting rectangle will have positive width and height. This approach was taken, so that the developer could independently modify individual rectangle edges, one at a time, without exceptions being generated during this process.

Inherited Methods: This Rect class inherits all methods defined for the Shape class, as well as the PhysicalComponent class.

Attributes:

In addition to these methods for the Rect class, there are also five attributes (or properties) defined for the Rect object, described as follows:

bottom – the coordinate value for the bottom of this rectangle

left – the coordinate value for the left side of this rectangle

right – the coordinate value for the right side of this rectangle

top – the coordinate value for the top of this rectangle

bbox – the bounding box which defines this rectangle

Note that these **bottom**, **left**, **right** and **top** attribute values will return the same values as returned by the “**getBottom**()”, “**getLeft**()”, “**getRight**()” and “**getTop**()” methods. In addition, these attributes are settable, so that they can be used to set these values in the same manner as the “**setBottom**()”, “**setLeft**()”, “**setRight**()” and “**setTop**()” methods. The

Feedback

bbox attribute will return the same value as the "**getBBox**()" method, and can be used to set the bounding box for this Rect, the same as the "**setBBox**()" method.

Examples:

```
rect1 = Rect(Layer('metal1'), Box(0,0,10.0,20.0))
rect1.getLayer()        # returns Layer('metal1')
rect1.getBBox()
# get dimensions of this rectangle
rect1.getTop()          # returns 20.0
rect1.getBottom()       # returns 0.0
rect1.getLeft()         # returns 0.0
rect1.getRight()        # returns 10.0
rect1.top               # returns 20.0
rect1.bottom            # returns 0.0
rect1.left              # returns 0.0
rect1.right             # returns 10.0
# get height and width
rect1.getHeight()       # returns 20.0
rect1.getWidth()        # returns 10.0
# set new dimensions for this rectangle
rect1.setTop(10.0)
rect1.right = 15.0
# check that height and width have changed
rect1.getHeight()       # returns 10.0
rect1.getWidth()        # returns 15.0
# perform various mirroring and rotation operations
rect1.mirrorX()
rect1.getBBox()
rect1.mirrorX()
rect1.getBBox()         # back to original position
rect1.rotate90()
rect1.rotate180()
rect1.rotate90()
rect1.getBBox()         # back to original position
# fill this rectangle with field of sub-rectangles
# First fill using DRC minimum sizes and spacing;
# use a Grouping to store these sub-rectangles
group1 = Grouping()
Rect.fillBBoxWithRects(Layer('metal2'), rect1.getBBox(),
                       group=group1)
# now fill using pre-specified sizes and spacing
group1.destroy()
Rect.fillBBoxWithRects(Layer('metal2'), rect1.getBBox(),
                       width = 1.0, height = 1.0,
                       spaceX = 0.5, spaceY = 0.5)
# create cloned copy of this rectangle shape
rect2 = rect1.clone()
```

Feedback

Text

Description: This Text class is used to represent text labels which can be incorporated within a physical layout design. These text labels can be used to annotate design objects within the layout, but are not attached to these design objects. This Text object is defined by means of a text string, an origin point for locating this text string in the layout design, and a height value which specifies the size of the text string. In addition, the designer optionally specify the specific location, orientation, font, visibility and drafting mode for this text label. Note that these optional capabilities for the text label can be specified when the text label is created, or can be later modified after text label creation, through the use of methods for this Text class.

Creation:

Text(Layer *layer*, string *text*, Point *origin*, Coord *height*,

Location *location*=Location.UPPER_LEFT, Orientation *orient*=Orientation.R0,

Font *font*=Font.STICK, bool *overbar*=False, bool *visible*=True, bool *drafting*=False) – creates a Text object, using the *text* string parameter as the text for this Text object, which is placed at the location given by the *origin* parameter, with the height specified by the *height* parameter. The required *layer* parameter is a Layer object, the *text* parameter is a string, the *origin* parameter is a Point object, and the *height* parameter is a floating-point coordinate value. The optional *location* parameter specifies the horizontal and vertical alignment for this Text object, while the optional *orient* parameter specifies the orientation for this Text object. The optional *font* parameter specifies the font which should be used to display the Text object, while the optional *overbar* parameter specifies whether an overbar should be used to display this Text object. The optional *visible* parameter specifies whether this Text object should be visible or not; this flag allows Text objects to be created, but not necessarily visible. The optional *drafting* parameter specifies whether drafting mode should be enabled for this Text object; if drafting mode is enabled, then text will always be drawn left-to-right or top-to-bottom.

Note that by default, drafting mode is set to False when a text object is created. Although this default is different from OpenAccess (which uses a default value of True for drafting mode), it does agree with the default which is used by the Cadence SKILL language.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this Text object

getAlignment() – returns the horizontal and vertical alignment for this Text object, as a Location object.

getFont() – returns the font for this Text object

getHeight() – returns the height for this Text object

Feedback

getOrientation() – returns the orientation for this Text object

getOrigin() – returns the origin point for this Text object

getText() – returns the text string for this Text object

hasOverbar() – returns True if an overbar is used to display this Text object, and returns False, otherwise.

isDrafting() – returns True if drafting mode is enabled for this Text object, and returns False, otherwise. If drafting mode is enabled, then this Text object will always be drawn left-to-right or top-to-bottom.

isVisible() – returns True if this Text object is visible (currently being displayed), and returns False, otherwise.

setAlignment(Location *location*) – sets the horizontal and vertical alignment for this Text object, using a Location object.

setDrafting(bool *drafting*) – sets a flag, to control drafting mode for this Text object. If *drafting* is True, then drafting mode will be used to display this Text object; if *drafting* is False, then drafting mode will not be used.

setFont(Font *font*) – sets the font for this Text object; the specified font value should be one of the pre-defined fonts which are provided by the Font class.

setHeight(Coord *height*) – sets the height for this Text object

setOrientation(Orientation *orient*) – sets the orientation for this Text object

setOrigin(Point *origin*) – sets the origin point for this Text object

setOverbar(bool *overbar*) – sets a flag, to control the display of an overbar for this Text object. If *overbar* is True, then an overbar will be displayed for the Text object; if *overbar* is False, then no overbar will be displayed.

setText(string *text*) – sets the text string for this Text object

setVisible(bool *visible*) – sets a flag, so that this Text object is visible, and will be currently displayed. This method is provided, so that Text objects can not be displayed, without requiring that they be destroyed.

Inherited Methods: This Text class inherits all methods defined for the Shape class, as well as the PhysicalComponent class.

Examples:

```
text1 = Text(Layer('metal1'),'My PyCell', Point(1,1), 2.5)
text1.getLayer()         # returns Layer('metal1')
text1.getBBox()
# perform mirroring operations
text1.mirrorY()
```

Feedback

```
text1.getBBox()
text1.mirrorY()
text1.getBBox()          # back to original position
# change this text shape
text1.getText()          # returns 'My PyCell'
text1.getOrigin()        # returns Point(1,1)
text1.getHeight()        # returns 2.5
text1.getAlignment()     # returns Location.UPPER_LEFT
text1.getOrientation()   # returns Orientation.R0
text1.getFont()          # returns Font.STICK
text1.isVisible()        # returns True
text.setText('My New PyCell')
text1.setOrigin(Point(2,2))
text1.setHeight(1.5)
text1.setAlignment(Location.LOWER_RIGHT)
text1.setOrientation(R180)
text1.setFont(Font.EURO_STYLE)
text1.setVisible(True)
text1.getText()          # returns 'My New PyCell'
text1.getOrigin()        # returns Point(2,2)
text1.getHeight()        # returns 1.5
text1.getAlignment()     # returns Location.LOWER_RIGHT
text1.getOrientation()   # returns Orientation.R180
text1.getFont()          # returns Font.EURO_STYLE
text1.isVisible()        # returns True
# create cloned copy of this text shape
text2 = text1.clone()
```

TextDisplay

Description: This is the base class for all attribute display objects which are defined for the Python API. The attribute display objects which are defined by the AttrDisplay class are derived from this base TextDisplay class. Note that this TextDisplay class is an abstract base class, so that all of the methods for this TextDisplay class are used by the AttrDisplay class objects which are derived from this base class.

Also note that this TextDisplay class provides the same methods as the Text class. In addition, there are methods to obtain and set the text display format. This text display format specifies whether the name or value for the associated object should be displayed.

Methods:

getAlignment() – returns the horizontal and vertical alignment for this TextDisplay object, as a Location object.

getFont() – returns the font for this TextDisplay object

getFormat() – returns the text display format for this TextDisplay object; this will be one of the enumerated values TextDisplay.Format.NAME, TextDisplay.Format.VALUE or TextDisplay.Format.NAME_VALUE.

Feedback

getHeight() – returns the height for this TextDisplay object

getOrientation() – returns the orientation for this TextDisplay object

getOrigin() – returns the origin point for this TextDisplay object

getText() – returns the text string for this TextDisplay object

hasOverbar() – returns True if an overbar is used to display this TextDisplay object, and returns False, otherwise.

isDrafting() – returns True if drafting mode is enabled for this TextDisplay object, and returns False, otherwise. If drafting mode is enabled, then this TextDisplay object will always be drawn left-to-right or top-to-bottom.

isVisible() – returns True if this TextDisplay object is visible (currently being displayed), and returns False, otherwise.

setAlignment(Location *location*) – sets the horizontal and vertical alignment for this TextDisplay object, using a Location object.

setDrafting(bool *drafting*) – sets a flag, to control drafting mode for this TextDisplay object. If *drafting* is True, then drafting mode will be used to display this TextDisplay object; if *drafting* is False, then drafting mode will not be used.

setFont(Font *font*) – sets the font for this TextDisplay object; the specified font value should be one of the pre-defined fonts which are provided by the Font class.

setFormat(TextDisplay.Format *format*) – sets the *format* text display format for this TextDisplay object using one of the enumerated values TextDisplay.Format.NAME, TextDisplay.Format.VALUE or TextDisplay.Format.NAME_VALUE.

setHeight(Coord *height*) – sets the height for this TextDisplay object

setOrientation(Orientation *orient*) – sets the orientation for this TextDisplay object

setOrigin(Point *origin*) – sets the origin point for this TextDisplay object

setOverbar(bool *overbar*) – sets a flag, to control the display of an overbar for this TextDisplay object. If *overbar* is True, then an overbar will be displayed for the TextDisplay object; if *overbar* is False, then no overbar will be displayed.

setVisible(bool *visible*) – sets a flag, so that this TextDisplay object is visible, and will be currently displayed. This method is provided, so that TextDisplay objects can not be displayed, without requiring that they be destroyed.

Inherited Methods: This TextDisplay class inherits all methods defined for the Shape class, as well as the PhysicalComponent class.

Feedback

AttrDisplay

Description: The AttrDisplay class is used to represent attributes of design layout objects which can be displayed when the objects are viewed. For example, names of terminals or nets can be displayed for specific terminals or nets in a layout design. This AttrDisplay shape is very similar to the Text shape, except that the text string represents the value of a specific design attribute. These attribute displays can be associated with a particular object in the layout design. Note that either the name of the attribute, the value of the attribute or both the name and value can be displayed in the design layout. In addition to the properties of text strings which are provided by the Text class, these attribute displays also provide specific information about the associated design objects. In particular, the attribute display represents a text string which is the value of a specific attribute of the associated design object. Whenever the value of the attribute changes, then the text string for the attribute display will automatically be updated with the new attribute value.

Note that there are a large number of pre-defined attributes which can be assigned to an attribute display. There are attributes for design objects (DloGen.AttrType), instances (Instance.AttrType), instance terminals (InstTerm.AttrType), nets (Net.AttrType) and terminals (TermAttrType). As an example, the OpenAccess library name for a design instance can be specified as an attribute value to be displayed, by using the enumerated type value DloGen.AttrType.LIB_NAME when the display attribute is created.

There are a number of enumerated class types which are used to define symbolic constant values which should be used with the AttrDisplay attribute display class. These classes and their attributes (or symbolic constant values) can be summarized as follows:

Class Name	Attributes
TextDisplay.Format	NAME, NAME_VALUE, VALUE
DloGen.AttrType	CELL_NAME, CELL_TYPE, LAST_SAVED_TIME, LIB_NAME, VIEW_NAME
Instance.AttrType	CELL_NAME, IS_BOUND, LIB_NAME, NAME, NUM_BITS, VIEW_NAME
InstTerm.AttrType	NAME
Net.AttrType	IS_EMPTY, IS_GLOBAL, IS_IMPLICIT, NAME, NUM_BITS, SIG_TYPE
Term.AttrType	HAS_PINS, NAME, NUM_BITS

In addition to these pre-defined attributes, each of these enumerated classes also provides the **getMembers**() class method, which can be used to obtain a list of all of the attributes which are defined for the class. Also note that there is no need to

`construct` any of these class objects; simply directly use the class attributes such as TextDisplay.Format.NAME , Instance.AttrType.CELL_NAME or Net.AttrType.IS_GLOBAL.

Creation:

AttrDisplay(*obj*, *attribute*, Layer *layer*, Point *origin*, Coord *height*,

Location *location*=Location.UPPER_LEFT, Orientation *orient*=Orientation.R0,

Font *font*=Font.STICK,

TextDisplay.Format *format*=TextDisplay.Format.NAME_VALUE,

bool *overbar*=False, bool *visible*=True, bool *drafting*=False) – creates an AttrDisplay object for the specified *obj* object , with the specified *attribute* attribute type. The *obj* object should be a design object corresponding to one of the classes DloGen, Instance, InstTerm, Net or Term. The specified *attribute* should use one of the pre-defined enumerated classes DloGen.AttrType, Instance.AttrType, InstTerm.AttrType, Net.AttrType or Term.AttrType. Note that the attribute type should correspond with the type of the design object; for example, a DloGen.AttrType should only be used with a DloGen class object. Otherwise, an exception is raised. This attribute display is placed at the location given by the *origin* parameter, with the height specified by the *height* parameter. The required *layer* parameter is a Layer object, the *origin* parameter is a Point object, and the *height* parameter is a floating-point coordinate value. The optional *location* parameter specifies the horizontal and vertical alignment for this AttrDisplay object, while the optional *orient* parameter specifies the orientation for this AttrDisplay object. The optional *font* parameter specifies the font to be used to display the AttrDisplay object. The optional *format* text display format specifies whether the name or value (or both) should be displayed; *format* should be either TextDisplay.Format.NAME, TextDisplay.Format.VALUE or TextDisplay.Format.NAME_VALUE. The optional *overbar* parameter specifies whether an overbar should be used for this AttrDisplay object. The optional *visible* parameter specifies whether this AttrDisplay object should be visible or not; this flag allows an AttrDisplay object to not be shown, without destroying the AttrDisplay object. The optional *drafting* parameter specifies whether drafting mode should be enabled; if drafting mode is enabled, then text will always be drawn left-to-right or top-to-bottom.

Methods:

getAttribute() – returns the display attribute type for this AttrDisplay object; this value will make use of one of the pre-defined sub-class types: DloGen.AttrType, Instance.AttrType, InstTerm.AttrType, Net.AttrType or Term.AttrType.

getObject() – returns the object for which this AttrDisplay attribute display is being defined; this will be either a DloGen, Instance, InstTerm, Net or Term class object.

Inherited Methods: This AttrDisplay class inherits all methods defined for the base TextDisplay class, as well as the Shape class.

Feedback

Examples:

```
# illustrate use of display attributes using Transistor
# create instance of Transistor
p = ParamArray()
inst = Instance('cnPyBasicDlo/PyTransistor', p)
# display text for Transistor cell name
instAttr = AttrDisplay(inst, inst.AttrType.CELL_NAME,
                       Layer('poly1'), Point(-1, -1), 0.10)
# first check that object and attribute are properly assigned
instAttr.getObject()       # returns instance
instAttr.getAttribute()    # returns Instance.AttrType.CELL_NAME
# change text display properties for this attribute display
instAttr.getOrigin()       # returns Point(-1, -1)
instAttr.getHeight()       # returns 0.10
instAttr.getAlignment()    # returns Location.UPPER_LEFT
instAttr.getOrientation()  # returns Orientation.R0
instAttr.setOrigin(Point(-1.1, -1.1))
instAttr.setHeight(0.15)
instAttr.setAlignment(Location.LOWER_LEFT)
instAttr.getOrigin()       # returns Point(-1.1, -1.1)
instAttr.getHeight()       # returns 0.15
instAttr.getAlignment()    # returns Location.LOWER_LEFT
# Now display text for Transistor Source and Drain;
# this is done using instance terminal attributes.
instTermS = inst.findInstTerm('S')
instTermD = inst.findInstTerm('D')
originS = instTermS.getInstPins()[0].getBBox().lowerLeft()
originD = instTermD.getInstPins()[0].getBBox().lowerRight()
termAttrS = AttrDisplay(instTermS, instTermS.AttrType.NAME,
                        Layer('metal1'), originS, 0.10)
termAttrD = AttrDisplay(instTermD, instTermD.AttrType.NAME,
                        Layer('metal1'), originD, 0.10)
```

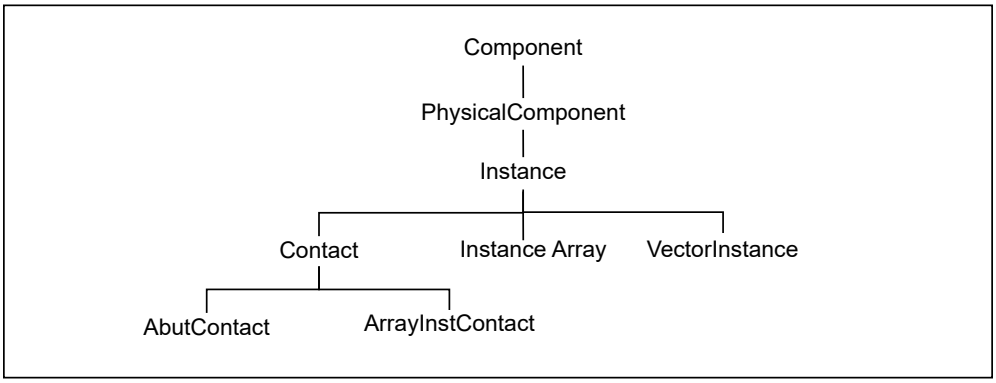
PHYSICAL COMPONENT RELATED CLASSES

Instance Classes

The Instance class is the base class for all types of instances of DLO objects. This Instance class is derived from the PhysicalComponent class, and is the base class for all instances of DLO class objects. For example, there are currently three sets of classes provided in the Python API which are derived from the Instance class; these are the InstanceArray class, the VectorInstance class, and the Contact classes. These Contact classes include the Contact class, the AbutContact class and the ArrayInstContact class.

Feedback

These different sets of Instance classes can be represented using the following class inheritance diagram:



Instance

Description: An instance object represents an instantiation of a DLO master. It is used to instantiate any DLO master an as object in a layout design. For example, the Contact DLO can be instantiated as an instance one or more times in the same design. Each such instance will be a unique instantiation of the Contact DLO master.

Creation:

Instance(string *dloName*, *params* = None, NodeSpec *nodeSpec* = None, string *name* = , Transform *trans*=None, bool *checkParams*=False) – creates an Instance object, where the *dloName* parameter specifies the OpenAccess name to be used for this Instance object. The *dloName* parameter is a string of the form `libName/cellName/viewName`, where the libName and the viewName are optional. If the libName is not specified, then the library name associated with the current DloGen is used; if the viewName is not specified, then the default viewName is used (currently `layout`). The *nodeSpec* parameter is a specification of the connectivity to be used for this instance. This *nodeSpec* parameter should be a Python dictionary whose keys are instance terminal names and whose values are net names (or nets). This *nodeSpec* parameter can also specify a list of net names (or nets), a single net name or a single net. If the single net name is an empty string, or this *nodeSpec* parameter is set to None, then this will be interpreted to mean that no connectivity should be defined. The *name* parameter is an optional string to be used to name this Instance object. The *trans* parameter is an optional Transform which should be applied when this instance is created. This optional *trans* parameter conveniently allows for the instance to be placed at the time of creation, rather than be moved after creation. The optional *checkParams* parameter allows you to check whether all parameters provided in the *params* are actually the PCell parameters. By default, non-PCell parameters are converted to properties and stored on the created Instance object. If *checkParams*=True, an exception is thrown when a non-PCell parameter is provided. When this Instance object is created, it will find all of the terminals on the Master object

for this Instance object, create the necessary nets (using the net names in the *nodeSpec* parameter, if it has been specified), bind these instance terminals to these nets, and finally create the user-accessible instance pin objects corresponding to these nets.

If the specified Instance can not be created for any reason, then an exception is raised. In addition, if the Pyros™ layout tool is being used, then all geometries generated up to the point at which the exception is raised will also be displayed. In addition, the instance bounding box rectangle and a text string describing the error condition will also be displayed by the Pyros layout viewing tool. Note that this Instance creation method may leave behind the OpenAccess instance which was created, even though an exception was raised during the Instance creation process. This remaining OpenAccess instance can be removed by calling the static Instance class **_destroyOAExceptionInstance**() method.

Note that it is possible for this Instance creation method to instantiate the same parameterized cell superMaster as its containing design. Although OpenAccess will allow circular instantiations to occur, the system will check for any such circular instantiations, and if any are found, then an exception is raised.

Note that if the DLO master is not a parameterized cell, then the *params* parameter should not be specified. If this *params* parameter is specified for a fixed (non-parameterized) cell, then an exception is raised.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this Instance object. Note that this method will not copy any attributes which may have been defined for this instance; if it is desired that any instance attributes be copied, then this should be handled by the derived class **clone**() method.

destroy( ) – destroys this Instance object.

find(string *name*=) – returns the Instance object for an Instance having this *name* name in the current DloGen design. This is done by searching through all Physical Component instances in the current DloGen design. If there is no Instance having this name in the current DloGen design, then None will be returned. If the *name* parameter is the empty string, then the first Instance in the design will be returned.

Note that this method is a static method, so that it can be called without explicitly creating an Instance object. For example, it can be directly called using the syntax `Instance.find('tran1')`. This makes it more convenient to find a named instance within the current design.

findInstPin(string *name*=) – returns the InstPin instance pin object which has the same name as the *name* parameter. If the *name* parameter is the empty string, then the first instance pin for this Instance will be returned. If there is no instance pin for this Instance having this name, then an exception is raised.

findInstTerm(string *name*=) – returns the InstTerm instance terminal object which has the same name as the *name* parameter. If the *name* parameter is the empty string, then the first instance terminal for this Instance will be returned. If there is no instance terminal for this Instance having this name, then an exception is raised.

flatten(bool *promoteNames*=False, bool *promotePins*=False, bool *promoteBlockages*=False) – flattens this Instance into the lowest-level primitive layout shapes which are contained in this Instance. Note that this Instance can contain a hierarchical design with an arbitrary number of levels of design hierarchy. These primitive layout shapes are then used to create a Grouping object having the same name as this Instance object; this Grouping object is returned by this method. Note that this Instance is destroyed, and then replaced by the returned Grouping object. Also note that, by default, any net, pin, or name information for the primitive shapes contained inside this Instance is not preserved. However, if *promoteNames* is True, then the existing names of flattened shapes are promoted. The promoted shape name is the hierarchical shape name; for example, if there is a shape with name "GATE" in the master of instance with name "I1", then the resulting shape has the name "I1/GATE". If *promotePins* is True, pin shapes are promoted, that is, they are added to the existing top level pin with the same name. If the top level pin with such a name does not exist, it is created. If *promoteBlockages* is True, then blockage objects are promoted.

getBBox(ShapeFilter *filter*=ShapeFilter()) – returns the bounding box for this Instance. Any layers specified in the ShapeFilter *filter* parameter are used to perform this bounding box calculation. Note that if this Instance does not exist on any of the specified layers, then a large inverted bounding box will be returned.

getBit(unsigned int *index*) – for a single-bit instance, always returns this Instance object.

getCompRefs() – returns a uniform list of all of the PhysicalCompRef objects which are references to physical components contained inside this Instance.

getDefaultParams(ParamArray *params*=None) – returns the ParamArray which provides the default parameters and values which were used when this Instance was created. If the *params* parameter is specified, then this ParamArray will be set to the same values as are returned by this method. If this Instance is not bound to a master design object, then an exception is raised.

getDloName() – returns the name for the master design object for this Instance. This master design name will be a string of the form `<libName>/<cellName>/<viewName>`.

getInstPins() – returns a uniform list of all of the instance pins for this Instance. If this Instance has no instance pins, then the returned list will be empty.

getInstTerms() – returns a uniform list of all of the instance terminals for this Instance. If this Instance has no instance terminals, then the returned list will be empty.

getMaster(bool *buildPyObj*=False) – returns the master design object for this Instance. This master design object is a Dlo object. If the *buildPyObj* parameter is True, then

PhysicalComponent, Instance, Term, Pin, and Grouping objects are created on Dlo from the master design. If this Instance is not bound to a master design object, then an exception is raised.

getName() – returns the name string for this Instance.

getNumBits() – returns the number of bits of this Instance. This function always returns 1 for scalar and array instances, but it can return 1 or more for vector instances.

getOrientation() – returns the orientation for this Instance.

getOrigin() – returns the current Point object which is the origin for this Instance.

getParams(ParamArray *params*=None, bool *all*=True) – returns the ParamArray which provides the explicit parameters and values which were used when this Instance object was created. If the Boolean *all* parameter is True, then all parameters (both explicit and default) will be returned. If the *params* parameter is specified, then this ParamArray will be set to the same values as are returned by this method. If this Instance is not bound to a master design object, then an exception is raised.

getParamSpecs() – returns the ParamSpecArray which defines the parameters and their specifications for this Instance. If this Instance is not bound to a master design object, then an exception is raised.

getTransform() – returns the Transform object for this Instance.

setConnectivity(self, NodeSpec *nodeSpec*, bool strict=True) – defines the connectivity to be used for this Instance object. This *nodeSpec* parameter can take one of a number of different formats. It can be a Python dictionary whose keys are instance terminals and whose values are net names (or nets). This *nodeSpec* parameter can also specify a list of net names (or nets), a single net name, or a single net. If the single net name is an empty string, or if this *nodeSpec* parameter is set to None, then this will be interpreted to mean that no connectivity should be defined. If this *nodeSpec* parameter dictionary does not contain entries for all of the instance terminals in the current design, then an exception is raised. Similarly, if the number of net names (or nets) in the *nodeSpec* parameter list of net names (or nets) does not match the number of instance terminals, then an exception is raised. When the optional parameter `strict` is set to False, then NodeSpec does not

have to specify nets for all terms.

setMaster(string *libName*=, string *viewName*=, string *cellName*=) – sets the master for this instance to be the design specified by the *libName*, *cellName* and *viewName* parameter values. The *libName* parameter specifies the library name for the master design, the *cellName* parameter specifies the cell name for the master design and the *viewName* parameter specifies the view name for the master design. If any of these three parameters are not specified, then the existing library name, cell name or view name for this Instance will be used by default. This method will also automatically update the connectivity for this Instance. All existing parameters for this Instance will be removed when this method is

used to change the master. If the same set of parameters should be used after changing masters, then the **getParams**() and **setParams**() methods should be used to save and restore the set of parameters.

setName(string *name*) – sets the name string for this Instance. If this *name* is already being used by an Instance in the current design, then an exception is raised.

setOrientation(Orientation *orient*) – sets the orientation for this Instance.

setOrigin(Point *point*) – sets the Point *point* parameter to be the origin for this Instance.

setParams(ParamArray *params*, bool *checkParams*=False) – uses the passed ParamArray *params* to set the parameter values for this Instance. The optional *checkParams* parameter allows you to check whether all parameters provided in the *params* are actually the PCell parameters. By default, non-PCell parameters are converted to properties and stored on this Instance object. Note that already existing properties with the same name will be overridden. If *checkParams*=True, an exception is thrown when a non-PCell parameter is provided. This method will also automatically update the connectivity for this Instance, as changing parameter values may cause connectivity changes. Instance pins and instance terminals will be created or destroyed, corresponding to the pins and terminals on the submaster which is created for the new parameter values.

If any of these parameter values is not defined as a parameter for the master of this Instance, then an exception is raised. If the value type for any of these parameter values does not agree with the expected type (as defined by the ParamSpecArray object), then an exception is raised. If this Instance is not bound to a master design object, then an exception is raised.

setTransform(Transform *trans*) – sets the Transform object for this Instance.

_destroyOAExceptionInstance(str *name*) – destroys any OpenAccess instance which may be left behind when the Instance class creation method raises an exception. This method is provided, so that the PyCell author can safely create another instance, without the possibility of any “left over” geometries remaining from an instance which failed to be successfully created. The *name* parameter is the same name as was used for the *name* parameter in the Instance creation method. If an OpenAccess instance with this name does not exist in the current design, then an exception is raised. If the OpenAccess instance with this name has a valid Dlo Instance object, then an exception is raised.

Inherited Methods: This Instance class inherits all methods defined for the PhysicalComponent class.

Attributes:

In addition to these methods for this Instance class, there are also two attributes (or properties) defined for this Instance class, described as follows:

name – the name for this Instance object

Feedback

AttrType – the display attribute type for this Instance object

The **AttrType** attribute is used with the AttrDisplay class, and has the attributes CELL_NAME, IS_BOUND, LIB_NAME, NAME, NUM_BITS and VIEW_NAME as well as the **getMembers**() class method defined for the Instance.AttrType class.

Note that the **name** attribute is also settable, so that it can be used in the same way as the "**getName**()" and "**setName**()" methods.

Examples:

```
# Create instance of basic Python transistor example,
# which uses default values for all of the parameters
p = ParamArray()
inst1 = Instance('cnPyBasicDlo/PyTransistor', p, None, 'tran1')
inst1.getName()            # returns 'tran1'
inst1.getDloName()   # returns 'cnPyBasicDlo/PyTransistor/layout'
inst1.getOrigin()          # returns Point(0,0)
inst1.getMaster()          # returns master design object
Instance.find('tran1')     # returns 'tran1' Instance object
# check that no nets were created for connectivity
inst1.getInstPins()        # returns list of InstPin objects
inst1.getInstTerms()       # returns list of InstTerm objects
for pin in inst1.getInstPins():
  pin.getNet()
for term in inst1.getInstTerms():
  term.getNet()
# now define connectivity for this instance
connDict = {'D':'d', 'G':'g', 'S':'s', 'B':'b'}
inst1.setConnectivity(connDict)
# now check that connectivity has been updated
inst1.getInstPins()        # returns list of InstPin objects
inst1.getInstTerms()       # returns list of InstTerm objects
for pin in inst1.getInstPins():
  pin.getNet()
for term in inst1.getInstTerms():
  term.getNet()
# Illustrate use of flatten method for an instance;
# used to obtain all lowest-level primitive shapes.
self.getComps()            # returns instance
group1 = inst1.flatten()
self.getComps()            # returns grouping
group1.ungroup()
self.getComps()            # returns all primitive shapes
```

InstanceArray

Description: The InstanceArray object represents an array of instances of one single DLO master design. It is used to instantiate a single DLO master as a rectangular array of instances. This class is typically used for memory efficiency reasons. For example, this

InstanceArray class can be used to represent an array of individual memory cells in a physical layout design.

Note that this InstanceArray class does not provide any mechanism for specifying the connectivity between individual elements of the array of instances. There is just a single copy of the terminals for the DLO master design. The width of any nets connected to these terminals is not affected by the dimensions of the array. Thus, as regards to connectivity, this instance array behaves like a single scalar instance.

Creation:

InstanceArray(string *dloName*, Coord *dX*, Coord *dY*, unsigned int *numRows*, unsigned int *numCols*, ParamArray *params*=None, NodeSpec *nodeSpec* = None, string *name*=, Transform *trans*=None, bool *checkParams*=False) – creates an InstanceArray object, where the *dloName* parameter specifies the OpenAccess name to be used for this InstanceArray object. This *dloName* parameter is a string of the form `libName/cellName/viewName`, where the libName and the viewName are optional. If the libName is not specified, then the library name associated with the current DloGen is used; if the viewName is not specified, then the default viewName is used (currently `layout`). The *dX* parameter specifies the offset spacing between array elements in the X direction, while the *dY* parameter specifies the offset spacing between array elements in the Y direction. The *numRows* parameter specifies the number of rows in the Y direction, while the *numCols* parameter specifies the number of columns in the X direction. The *params* parameter specifies the array of parameter values which should be used when each of the parameterized cell DLO masters in the array is instantiated. The *nodeSpec* parameter is an optional specification for desired connectivity, while the *name* parameter is an optional string to be used to name this InstanceArray object. The *trans* parameter is an optional Transform which should be applied when this instance array is created. This optional *trans* parameter conveniently allows for each element of this instance array to be placed at the time of creation, rather than to be moved after creation. The optional *checkParams* parameter allows you to check whether all parameters provided in the *params* are actually the PCell parameters. By default, non-PCell parameters are converted to properties and stored on the created InstanceArray object. If *checkParams*=True, an exception is thrown when a non-PCell parameter is provided. When this InstanceArray object is created, it will find all of the terminals on the Master object for this Instance Array object, create the necessary nets (using the connectivity specified in the *nodeSpec* parameter, if it has been specified), bind these instance terminals to these nets, and finally create the user-accessible instance pins corresponding to these nets.

Note that if the DLO master is not a parameterized cell, then the *params* parameter should not be specified. If this *params* parameter is specified for a fixed (non-parameterized) cell, then an exception is raised.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this InstanceArray object.

find(string *name*=) – returns the InstanceArray object for an InstanceArray having this *name* in the current DloGen design. This is done by searching through all InstanceArray objects in the current DloGen design. If the *name* parameter is the empty string, then the first InstanceArray in the design will be returned. If there is no InstanceArray having this *name* in the current design, then None will be returned.

Note that this method is a static method, so that it can be called without explicitly creating an InstanceArray object. For example, it can be directly called using the syntax `InstanceArray.find('mem1')`.

flatten() – flattens this InstanceArray object into the lowest-level primitive layout shapes which are contained in this InstanceArray. These primitive layout shapes are then used to create a Grouping object having the same name as this InstanceArray object, which is returned by this method. Note that this InstanceArray will be destroyed, and replaced by the returned Grouping object. Also note that any net, pin or name information for the primitive shapes contained inside this InstanceArray will not be preserved.

getDX() – returns the offset spacing between columns in the X direction for this InstanceArray object.

getDY() – returns the offset spacing between rows in the Y direction for this InstanceArray object.

getMemberBBox(unsigned int *row,* unsigned int *col,* ShapeFilter *filter*=ShapeFilter()) – returns the bounding box for the instance located at the specified row and column location in this InstanceArray object. Any layers specified in the ShapeFilter *filter* parameter are used to perform this bounding box calculation. Note that if the array instance does not exist on any of the specified layers, then a large inverted bounding box will be returned.

getMemberRefs() – returns a uniform list of all of the members of this InstanceArray object, which are being used as physical component reference objects.

getMemberTransform(unsigned int *row,* unsigned int *col*) – returns the transform for the instance located at the specified row and column location in this InstanceArray object. Note that the Transform for this entire InstanceArray object is given by the inherited **getTransform**() method, which should be the same as the transform for the member instance located at (0,0).

getNumCols() – returns the number of columns in this InstanceArray object.

getNumRows() – returns the number of rows in this InstanceArray object.

setDX(Coord *dX*) – sets offset spacing between columns for this InstanceArray object.

setDY(Coord *dY*) – sets offset spacing between rows for this InstanceArray object.

setName(string *name*) – sets the name string for this InstanceArray object. If this *name* is being used by an InstanceArray in the current design, then an exception is raised.

Feedback

setNumCols(unsigned int *numCols*) – sets number of columns for this InstanceArray object.

setNumRows(unsigned int *numRows*) – sets number of rows for this InstanceArray object.

Inherited Methods: This InstanceArray class inherits all methods defined for the Instance class, as well as all methods defined for the PhysicalComponent class.

Attributes:

In addition to these methods for this InstanceArray class, there is also an attribute (or property) defined for this InstanceArray class, described as follows:

name – the name for this InstanceArray object

Note that this **name** attribute is also settable, so that it can be used in the same way as the "**getName**()" and "**setName**()" methods.

Examples:

```
# create 2 by 3 array of basic Python transistor instances,
# which uses default values for all of the parameters
p = ParamArray()
inst2 = InstanceArray('cnPyBasicDlo/PyTransistor', 5, 4, 2, 3, p, None,
 'tran_array')
inst2.setDX(6)              # increase spacing in X-direction
inst2.getNumRows()          # returns 2
inst2.getNumCols()          # returns 3
inst2.setNumRows(10)        # increase number of rows
inst2.setNumCols(10)        # increase number of columns
inst2.getNumRows()          # returns 10
inst2.getNumCols()          # returns 12
inst2.getMemberBBox(1,2)    # returns bounding box at (1,2)
```

VectorInstance

Description: A vector instance represents several copies of the instance master with a range of index numbers to differentiate them. VectorInstance class is derived from Instance class. Vector instances have vector names, for example, I[1:3].

Creation:

VectorInstance(string *dloName*, unsigned int *startIndex*, unsigned int *stopIndex*, ParamArray *params* = None, NodeSpec *nodeSpec* = None, string *baseName* = , Transform *trans* = None, bool *checkParams* = False) – creates a vector instance object, where the *dloName* parameter specifies the OpenAccess name to be used for this VectorInstance object. The *dloName* parameter is a string of the form `libName/cellName/viewName`, where the libName and the viewName are optional. If the libName is not specified, then the library name associated with the current DloGen is used; if

the viewName is not specified, then the default viewName is used (currently `layout`). The *startIndex* parameter is the Starting bit index, while the *stopIndex* parameter is the Stopping bit index. The *nodeSpec* parameter is a specification of the connectivity to be used for this vector instance. This *nodeSpec* parameter should be a Python dictionary whose keys are vector instance terminal names and whose values are net names (or nets). This *nodeSpec* parameter can also specify a list of net names (or nets), a single net name or a single net. If the single net name is an empty string, or this *nodeSpec* parameter is set to None, then this will be interpreted to mean that no connectivity should be defined. The *baseName* parameter is the base part of the vector instance name (without bit indexing). The *trans* parameter is an optional Transform which should be applied when this vector instance is created. This optional *trans* parameter conveniently allows for the vector instance to be placed at the time of creation, rather than be moved after creation. The optional *checkParams* parameter allows you to check whether all parameters provided in the *params* are actually the PCell parameters. By default, non-PCell parameters are converted to properties and stored on the created VectorInstance object. If *checkParams*=True, an exception is thrown when a non-PCell parameter is provided. When this VectorInstance object is created, it will find all of the terminals on the Master object for this VectorInstance object, create the necessary nets (using the net names in the *nodeSpec* parameter, if it has been specified), bind these vector instance terminals to these nets, and finally create the user-accessible vector instance pin objects corresponding to these nets.

Note that if the DLO master is not a parameterized cell, then the *params* parameter should not be specified. If this *params* parameter is specified for a fixed (non-parameterized) cell, then an exception is raised.

Methods:

clone(NameMapper *nameMap* = NameMapper(), NameMapper *netMap* = NameMapper()) – returns a cloned copy of this vector instance object.

find(string *name* =) – returns the vector instance object having this name in the current DloGen design. This is done by searching through all vector instance objects in the current DloGen design. If the *name* parameter is the empty string, then the first vector instance in the design will be returned. If there is no vector instance having this name in the current design, then None will be returned.

Note that this method is a static method, so that it can be called without explicitly creating a VectorInstance object. For example, it can be directly called using the syntax `VectorInstance.find('I[1:3]')`.

flatten(bool *promoteNames* = False, bool *promotePins* = False) – flattens this vector instance into the lowest-level primitive layout shapes which are contained in this vector instance. Note that the number of shapes in the resulting grouping depends on the number of bits in this vector instance. For example, if *promoteNames* is True, and if there is a shape with name "GATE" in the master of vector instance with name "I[1:2]", then the

resulting grouping will contain two corresponding shapes with names "I[1]/GATE" and "I[2]/GATE". See also **Instance.flatten**().

getBaseName() – returns the base part of the name of this vector instance (without bit indexing).

getBit(unsigned int *index*) – returns a vector instance bit by its index. For example, for vector instance with name "I[3:0]", and index = 0, returns the vector instance bit with name "I[3]".

getCompRefs() – returns a uniform list of all of the PhysicalCompRef objects which are references to physical components contained inside this vector instance. Note that the number of references to physical components in the resulting list depends on the number of bits in this vector instance. For example, if there is a shape with name "GATE" in the master of vector instance with name "I[1:2]", then the resulting list will contain two corresponding references with names "I[1]/GATE" and "I[2]/GATE".

getNumBits() – returns the number of bits of this vector instance.

getStartIndex() – returns the starting index of this vector instance bits.

getStopIndex() – returns the stopping index of this vector instance bits.

setBaseName(string *baseName*) – sets the base part of this vector instance name.

setIndexRange(unsigned int *startIndex*, unsigned int *stopIndex*) – sets the starting and stopping indexes of this vector instance bits.

Inherited Methods: This VectorInstance class inherits all methods defined for the Instance class, as well as all methods defined for the PhysicalComponent class.

Contact

One of the class objects which is derived from the Instance class object is the Contact class. This Contact class is used to create a connection between arbitrary interconnect layers, which also meet all DRC and electrical rules. That is, the contact objects which are created by means of this Contact class will be DRC correct-by-construction, and will meet the electrical rules as specified in the technology files for the given process.

Description: The Contact class provides a connection between specified interconnect layers in a layout design. Note that these interconnect layers are not required to be vertically adjacent layers, but can be any specified interconnect layers. An interconnect layer is either a diffusion, polysilicon or metal layer in a design. An intermediate interconnect layer for a contact is any interconnect layer for the contact which lies between the two layers which are being connected by this contact. For example, if the contact is being used to connect the Layer('metal1') and Layer('metal3') layer objects, then Layer('metal2') would be an intermediate layer.

The Contact object can have one or more cuts, which are shapes on the cut layer (either a contact layer or via layer), which are used to connect the interconnect layer immediately below the cut layer to the interconnect layer immediately above the cut layer. This number of cuts can be increased in order to improve manufacturing yield, reduce parasitic resistance, or to increase current capacity for the device being designed. Note that this Contact object will automatically construct the necessary cuts, and that the user can specify the minimum number of cuts for this contact. These square fixed-size cut shapes will be constructed to meet the required DRC rules for the given technology. The number of cuts is originally determined based upon the length and width of this contact, as well

as the pitch for these cuts; the number of cuts is then adjusted to be at least as large as the minimum number of cuts specified by the user. Note that the size of this Contact object will be automatically increased, if necessary, to accommodate a larger number of cuts. In addition, if this Contact object is stretched to a larger size, then the number of cuts will be automatically increased.

There are a large number of DRC rules which are taken into account when this Contact object is created or modified. This includes such rules as minimum width, minimum area, minimum via spacing and minimum via enclosure rules. As a consequence, in some cases (depending upon the technology file used to construct the Contact) the created Contact may be larger than the size requested when the Contact was created. For example, via spacing and enclosure rules (as well as minimum area rules) may necessitate that a larger Contact object be created to meet these minimum via spacing and enclosure design rules.

Note that it may be necessary to specify additional layers besides the interconnect layer and the cut layer which should be used in the construction of this Contact object. For example, an implant layer may be required to complete the contact. It is also sometimes useful to use additional higher metal layers to increase metal density on the higher metal layers. These additional layers can be specified using the optional *addLayers* parameter when the contact is created.

Unlike other classes defined in the Python API, this Contact object is actually implemented as a Contact DLO, so that it is also a parameterized cell. The set of parameters for this Contact DLO can be modified by using the **setParams**() method inherited from the Instance class. The complete set of parameters for this Contact DLO can be briefly described as follows:

layer1 – first layer to be connected by this contact

layer2 – second layer to be connected by this contact

routeDir1 – direction from which layer1 will connect to this contact

routeDir2 – direction from which layer2 will connect to this contact

width – width of the reference box for this contact

length – length of the reference box for this contact

Feedback

addLayers – any additional layers which should be used for this contact

Note that the values for many of these Contact DLO parameters are initially set when this Contact object is first created. In addition, any of these parameter values for the Contact DLO can also be modified through the use of the **setParams**() method, which is inherited from the Instance class.

Creation:

Contact(Layer *layer1*, Layer *layer2*, NodeSpec *nodeSpec*=None, Direction *routeDir1* = Direction.NONE, Direction *routeDir2* = Direction.NONE, Point *point1* = Point.INVALID, Point *point2* = Point.INVALID, Direction *anchor* = Direction.CENTER, LayerList *addLayers* = None, string *name* =) – creates a Contact object which connects *layer1* to *layer2*. The reference box for this contact is defined by *point1* and *point2*. The *layer1* and *layer2* Layer objects should be either diffusion, polysilicon or metal layers. The *routeDir1* parameter specifies the routing direction for *layer1*, while *routeDir2* specifies the routing direction for *layer2*. Note that if these route direction parameters are specified, then minimum area design rules will not be enforced; it is assumed that minimum area design rules will be satisfied when routing to the contact has been completed. In this case, minimum area design rules will still be enforced for any intermediate layers; minimum area design rules are not enforced only for the *layer1* and *layer2* top and bottom layers. The *anchor* Direction specifies the reference point for the reference box. Note that this *anchor* reference point is kept fixed, whenever this Contact object is automatically re-sized to meet DRC requirements, or when the parameter values for the Contact DLO are changed. Thus, this *anchor* parameter can be used to control the location of this Contact object whenever it is created or re-sized. The optional *addLayers* parameter is used to specify any additional layers besides interconnect and cut layers which should be used in the construction of this Contact object, such as pwell or nwell well layers. The *name* parameter is an optional string parameter for the name of this contact instance. The *nodeSpec* parameter is used to specify the connectivity for this Contact object; if this *nodeSpec* parameter is set to None, then no connectivity will be established for this Contact object.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this Contact object.

getLayer1() – returns the Layer object for the first connection layer for this contact.

getLayer2() – returns the Layer object for the second connection layer for this contact.

getLayers() – returns the lists of routing layers, interconnect layers, and any additional layers for this contact. These Layer objects are returned as a 3-tuple of lists of Layer objects: (routeLayers, cutLayers, addLayers).

getNumCuts() – returns the current number of cuts for this contact.

Feedback

getNumHVCuts() – returns the current number of cuts in the horizontal direction, and the current number of cuts in the vertical direction. These numbers are returned as a tuple of integers: (numHorCuts, numVertCuts).

getRefBox() – returns the reference box for this contact.

setMinCuts(unsigned int *minCuts*) – the *minCuts* parameter is used to specify the minimum number of cuts for this contact. Note that the size of this contact will be automatically adjusted as necessary, to accommodate this minimum number of cuts.

stretch(Box *refBox*) – grow or shrink this contact until its length and width have exactly the same dimensions as the *refBox* reference box passed as a parameter. In addition, this contact is re-positioned, so that it is properly positioned with respect to the anchor.

stretchTo(Direction *dir*, Point *point*) – grow or shrink this contact, using the *point* parameter to define the a new reference box in the given direction. If the *dir* Direction parameter specifies a single direction (such as NORTH or SOUTH), then grow or shrink this contact until the given point is collinear with the specified side. If this *dir* Direction parameter specifies two directions (such as NORTH_EAST or SOUTH_WEST), then grow or shrink this contact until the specified corner is the given point. An exception will be raised, if the resulting reference box is inverted.

stretchToCoord(Direction *dir*, Coord *value*) – grow or shrink this contact, using the passed coordinate *value* to define the new reference box in the *dir* parameter direction. If this *dir* Direction parameter specifies a single Direction (such as NORTH or SOUTH), then grow or shrink this contact until the specified side has the given Coord *value*. If this Direction specifies two directions (such as NORTH_EAST or SOUTH_WEST), then grow or shrink this contact until the two relevant sides have this Coord *value*. An exception will be raised, if the resulting reference box is inverted.

Inherited Methods: This Contact class inherits all methods defined for the Instance class, as well as the PhysicalComponent class.

Examples:

```
contact1 = Contact(Layer('metal1'), Layer('metal3'), 'A')
contact1.getLayer1()  # returns Layer('metal1')
contact1.getLayer2()  # returns Layer('metal3')
# returns lists of route, interconnect and additional layers
(routeLayers, cutLayers, addLayers) = contact1.getLayers()
# check that Contact gets expanded to accommodate extra cuts,
# by comparing size of reference box before and after cuts
contact1.getRefBox()

contact1.setMinCuts(7)
contact1.getRefBox()
# get the actual number of cuts which were created
contact1.getNumCuts()
# also get the number of horizontal and vertical cuts
(numHorCuts, numVertCuts) = contact1.getNumHVCuts()
```

Feedback

```
# now enlarge Contact by stretching to fit larger reference box
box1 = Box(-1, -1, 2, 2)
contact1.stretch(box1)
contact1.getRefBox()   # returns Box(-1,-1,2,2)
# note that the number of cuts will be automatically increased
contact1.getNumCuts()
```

AbutContact

Description: The AbutContact class provides a specialized contact, which can be used to abut to other contacts. The AbutContact class is directly derived from the basic Contact class, and explicitly makes use of via spacing rules to create contacts which can be easily abutted to other contacts. This AbutContact class inherits all of the methods of the Contact class, and provides an additional method to control the via spacing between the two abutting contacts. In addition, the designer can specify the specific direction which should be used to abut the two contacts.

Note that minimum area design rules are not enforced for this abut contact. Instead, it is assumed that these minimum area design rules will be satisfied when these contacts are abutted to other contacts.

As is the case for the Contact object, this AbutContact object is actually implemented as a a DLO, so that it is also a parameterized cell. The set of parameters for this AbutContact DLO can be modified by using the **setParam**() method described below. This AbutContact DLO has all of the parameters of the Contact DLO, as well as three additional parameters:

abutDir – direction in which this contact will abut to another contact

abutViaSpaceFactor – specifies via spacing factor between abutting contacts

symAddLayer – symmetric generation of enclosure rectangles for additional layers

Note that the values for these AbutContact DLO parameters are initially set when this AbutContact object is first created. In addition, the **setAbutViaSpaceFactor**() method sets the value for the `abutViaSpaceFactor` parameter. These parameter values for the AbutContact DLO can also be modified through the use of the **setParams**() method, which is inherited from the Instance class.

Creation:

AbutContact(Layer *layer1*, Layer *layer2*, NodeSpec *nodeSpec*=None, Direction *routeDir1* = Direction.NONE, Direction *routeDir2* = Direction.NONE, Point *point1* = Point.INVALID, Point *point2* = Point.INVALID, Direction *anchor* = Direction.CENTER, LayerList *addLayers* = None, string *name* = , Direction

abutDir=Direction.NONE, int *abutViaSpaceFactor*=1, bool *symAddLayer*=False) – creates an AbutContact object. Note that this AbutContact creation method has the same parameters as the Contact class, along with three additional parameters. The first of these

Feedback

additional parameters is the *abutDir* parameter, which specifies the direction in which the via spacing design rule will be enforced. Extra enclosure spacing will be used, so that the abutting contacts will meet applicable via spacing rule requirements. Note that an *abutDir* value of NONE means that no via spacing rules will be applied in any direction, while a value of ANY means that via spacing rules will be applied in all four directions. Thus, an *abutDir* value of ANY should not be used, unless it is necessary to abut this contact in both horizontal and vertical directions, since the resulting contact will be larger than one with a single horizontal or vertical direction. The second additional parameter is *abutViaSpaceFactor*, which provides explicit control of the spacing between the contact reference box and the inside via cut in the direction of the abutting contact. This parameter is a multiple of one-half the via spacing design rule value. By default, this parameter is set to 1, which allows the two abutting contacts to evenly divide the via spacing requirement. The third additional parameter is the *symAddLayer* parameter, which is a Boolean flag used to specify that the enclosure rectangles created for any additional layers should be symmetrically sized for all additional layers. Note that these additional layers are specified by means of the optional *addLayers* parameter.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this AbutContact object.

setAbutViaSpaceFactor(int *abutViaSpaceFactor*) – the *abutViaSpaceFactor* parameter is used to specify the spacing between the reference box for this contact and the via cut inside for the side of the contact which matches the *abutDir* direction. This value is a non-negative integer, and is a multiple of one-half the via spacing design rule value.

Inherited Methods: This AbutContact class inherits all methods defined for the Contact class, as well as all methods defined for the Instance and PhysicalComponent classes.

Examples:

```
# create two abutting contacts, and then modify via
# spacing with the "setAbutViaSpaceFactor()" method
contact1 = AbutContact(Layer('metal1'), Layer('metal2'), 'A',
 abutDir=EAST_WEST)
contact2 = AbutContact(Layer('metal1'), Layer('metal2'), 'B',
 abutDir=EAST_WEST)
# stretch these two abutting contacts
contact1.stretch(Box(0, 0, 2, 0.5))
contact2.stretch(Box(0, 0, 0.5, 2))
# now abut these two contacts
contact1.abut(EAST, contact2)
# change the via spacing factor for the first contact
contact1.setAbutViaSpaceFactor(2)   # number of cuts is reduced
```

Feedback

ArrayInstContact

Description: The ArrayInstContact class provides a specialized contact, which only differs from the Contact class in terms of the implementation of any necessary cuts. In the case of the Contact class, any cuts are constructed using separate rectangles. However, for larger contacts which require the use of a larger number of cuts, it would be more efficient to use the InstanceArray class to implement these cuts as an array of rectangle instances. The reason for this is that the array of rectangle instances can be implemented using a single OpenAccess database object (oaArrayInst), rather than as a larger number of individual rectangle database objects (oaRect). This provides improved performance, when the number of contact cuts is large. As a general guideline or `rule of thumb`, it is suggested that this ArrayInstContact class be used in place of the usual Contact class, when the number of contact cuts will be larger than ten to twenty cuts.

As is the case for the Contact object, this ArrayInstContact object is implemented as a DLO, so that it is a parameterized cell. The set of parameters for this ArrayInstContact DLO can be modified by using the **setParams**() method inherited from the Instance class. This ArrayInstContact DLO has all of the parameters of the Contact DLO.

Creation:

ArrayInstContact(Layer *layer1*, Layer *layer2*, NodeSpec *nodeSpec*=None, Direction *routeDir1* = Direction.NONE, Direction *routeDir2* = Direction.NONE, Point *point1* = Point.INVALID, Point *point2* = Point.INVALID, Direction *anchor* = Direction.CENTER, LayerList *addLayers* = None, string *name* =) – creates a ArrayInstContact object. Note that this ArrayInstContact creation method has the same parameters as the Contact class. The only difference is that the necessary cuts for this ArrayInstContact object are constructed using the InstanceArray class, rather than using separate rectangles.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this ArrayInstContact object.

Inherited Methods: This ArrayInstContact class inherits all methods defined for the Contact class, as well as all methods defined for the Instance and PhysicalComponent classes.

Examples:

```
# create two contacts, standard contact with cut rectangles,
# and one using an InstanceArray class object for the cuts
ct1 = Contact(Layer('metal1'), Layer('metal3'), 'A')
ct2 = ArrayInstContact(Layer('metal1'), Layer('metal3'), 'A')
# check that Contact gets expanded to accommodate extra cuts,
# by comparing size of reference box before and after cuts
ct1.getRefBox()
ct1.setMinCuts(7)
ct1.getRefBox()
```

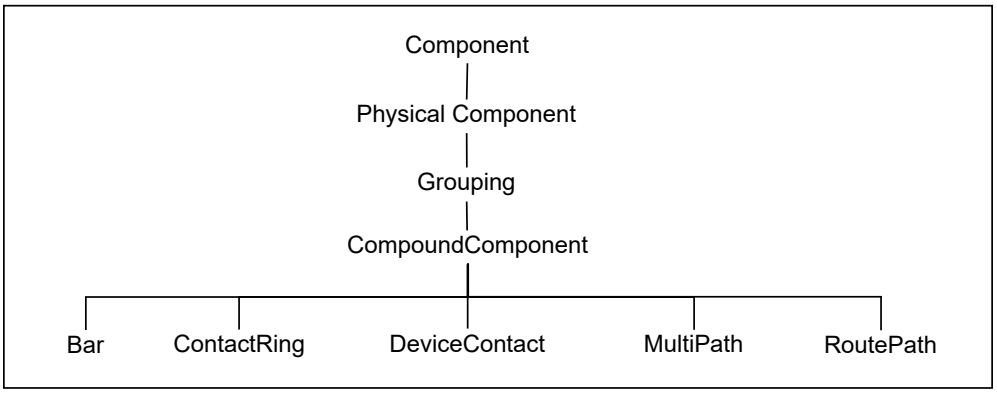
Feedback

```
ct2.getRefBox()
ct2.setMinCuts(7)
ct2.getRefBox()
# get the actual number of cuts which were created
ct1.getNumCuts()
ct2.getNumCuts()
# now enlarge contacts by stretching to fit larger reference box
box1 = Box(-1, -1, 2, 2)
ct1.stretch(box1)
ct1.getRefBox()    # returns Box(-1,-1,2,2)
box2 = Box(2, 2, 6, 6)
ct2.stretch(box2)
ct2.getRefBox()    # returns Box(2,2,6,6)
# note that the number of cuts will be automatically increased
ct1.getNumCuts()
ct2.getNumCuts()
```

Grouping Classes

The Grouping class is the base class for all CompoundComponent objects. This Grouping class is derived from the PhysicalComponent class, and in turn, is the base class for all CompoundComponent class objects. For example, there are currently four compound component classes provided in the Python API which are derived from this CompoundComponent class; these are the Bar class, the ContactRing class, the MultiPath class and the RoutePath class.

These different sets of Grouping classes can be represented using the following class inheritance diagram:



Grouping

Description: The Grouping class provides the ability to group together one or more physical components into a logical grouping, which can then be treated as a separate design unit. For example, this allows a grouping of physical components to be moved as a single unit, instead of moving each physical component in the grouping separately.

The benefit of using this Grouping class is that it makes it easy to apply different layout manipulation operations to different parts of a physical layout design. If adjustments need to be made to certain objects in the layout, then these operations can be applied in a single global operation all at once, versus operating on each object one at a time. For example, all of the physical components in a Grouping object could either be rotated or mirrored in a single operation, and then moved in a given direction. This Grouping class is a convenient construct with specialized methods for operating on sets of physical components.

This Grouping object is a logical grouping of physical component objects which is used for convenience during the PyCell design process. Note that these logical groupings of physical components can be saved as oaFigGroup objects in the OpenAccess database that is generated for the PyCell design, so that they will be persistent database objects.

Groupings can be nested, so that hierarchical Grouping objects can be created. The base Grouping would be created using a collection of physical components. Then this base Grouping object could be combined with other Grouping objects, or with other collections of physical components to create a hierarchical grouping. Note that hierarchical groupings can be created with any desired level of hierarchy.

Creation:

Grouping(string *name*=, components = None) – creates a Grouping object, where the *name* parameter specifies the name to be used for this Grouping object. The optional *components* parameter specifies either a single physical component, or a list of physical components. Each of these physical components will be added to this Grouping object, after it is created. If the *name* parameter is not specified, then a default name will be automatically generated. If this *components* parameter is not specified, then an empty Grouping object will be created; components can then be added to this empty Grouping object, using the "**add**()" method. Note that the name for this Grouping object must be unique; this *name* parameter should not conflict with any name which has been used as the name for a Grouping object, Instance object or CompoundComponent object.

Note that there are two different ways to create a Grouping object which contains all of the members contained within the current DloGen design object. The first approach is to directly call this Grouping creation method, using the "self.**getComps**()" method for the optional *components* parameter. The second approach is to simply use the DloGen "**makeGrouping**()" method. The benefit of this second approach is that it is more efficient in terms of run-time performance than the first method. Otherwise, these two different approaches produce the same result.

Methods:

add(*components*) – the *components* parameter should be either a single physical component, or a list of physical component objects. Each of these physical component will be added to this grouping.

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this Grouping object. Note that the physical components contained within this Grouping object are also cloned.

destroy() – this is the method on the base class which destroys the Container object for this Grouping object. Note that any class (such as the Bar or ContactRing class) which is derived from this Grouping object also provides their own destroy() method, which will destroy each of the individual components contained within this Grouping object.

find(string *name*=) – returns the Grouping object for a Grouping having this *name* name in the current DloGen design. This is done by searching through all Grouping objects in the current DloGen design. If the *name* parameter is the empty string, then the first Grouping in the design will be returned. If there is no Grouping having this name in the current DloGen design, then None will be returned.

Note that this method is a static method, so that it can be called without explicitly creating a Grouping object. For example, it can be directly called using the syntax `Grouping.find('group1')`. This makes it more convenient to find a named grouping within the current design.

flatten() – flattens this Grouping into the physical components which are contained in this Grouping. Note that this operation is performed recursively, so that any hierarchical groupings contained inside this Grouping object will also be flattened into the lowest-level physical components. These lowest-level physical components are then used for the new contents of this Grouping object, which is then returned by this method. Note that any hierarchical groupings within this Grouping object will automatically be flattened.

getBBox(ShapeFilter *filter*=ShapeFilter()) – returns the bounding box for this grouping. Any layers specified in the ShapeFilter *filter* parameter are used to perform this bounding box calculation. This bounding box is calculated as the merged bounding box for each of the bounding boxes for the physical components defined on the specified layers.

getComp(int *index*) – returns the component contained in this Grouping which has the specified *index* value in the list of components maintained for this Grouping. If this *index* value is out of range, then an exception will be generated. Note that these index values are assigned (or updated) as components are added to or removed from this Grouping. This index order is exactly the same as the ordering in the list of physical components returned by the **getComps**() method.

getComps() – this method returns a uniform list of all members of this grouping. Note that this list will be a `snapshot` of the members present when this method is called; this list will not be updated as any members are added to or deleted from this grouping. Instead, this method should be called again to obtain a new `snapshot` of the members of this grouping.

getLeafComps() – this method returns a uniform list of all leaf-level components which are contained in this grouping. As is the case for the **getComps**() method, this list will be a `snapshot` of the leaf-level physical components present when this method is called.

This method should be called again to obtain a new `snapshot` of the leaf-level physical components contained in this grouping.

getName() – returns the name of this grouping. Note that names need to be unique for different Grouping objects in the same PyCell design.

isPersistent() – returns True if this Grouping object is persistent, and False otherwise.

makePersistent(bool *persistent*) – makes this Grouping object persistent in the OpenAccess database, by saving this object as an oaFigGroup object in the database. Note that there is no default value for the *persistent* parameter. However, when this Grouping object is created it will not be persistent, until this method is used. Also note that when a design containing a persistent Grouping object is re-opened, then this persistent Grouping object will be restored and made available.

mirrorX(Coord *yCoord* = 0) – mirrors all physical components in this grouping about the X coordinate axis. The optional *yCoord* parameter can be used to specify a displacement for the location of the X coordinate axis. Note that this displacement value is specified in user units.

mirrorY(Coord *xCoord* = 0) – mirrors all physical components in this grouping about the Y coordinate axis. The optional *xCoord* parameter can be used to specify a displacement for the location of the Y coordinate axis. Note that this displacement value is specified in user units.

moveBy(Coord *dx*, Coord *dy*) – moves all physical components in this grouping by *dx* units in the x-coordinate and *dy* units in the y-coordinate. Note that these coordinate values are specified in user units.

moveTo(Point *destination*, Location *handle*=Location.CENTER_CENTER, ShapeFilter *filter*=ShapeFilter()) – moves all physical components in this grouping, so that the *destination* Point becomes the handle point for the bounding box for this grouping at the *handle* location. This bounding box is determined using the *filter* ShapeFilter parameter. If these physical components do not have any geometry on the layers specified by the *filter* ShapeFilter parameter, then an exception is raised.

moveTowards(Direction *dir*, Coord *distance*) – moves all physical components in this grouping by *distance* coordinate units in the *dir* parameter direction. Note that these coordinate units are specified using user units.

remove(*components*) – the *components* parameter should be either a single physical component, or a list of physical component objects. Each of these physical components will be removed from this grouping.

rotate90(Point *origin*=None) – rotates all physical components in this grouping by 90 degrees in a counter-clockwise direction. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

Feedback

rotate180(Point *origin*=None) – rotates all physical components in this grouping by 180 degrees in a counter-clockwise direction. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

rotate270(Point *origin*=None) – rotates all physical components in this grouping by 270 degrees in a counter-clockwise direction. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

setName(string *name*) – sets the name string for this Grouping. Note that this *name* should be unique for the different PhysicalComponent objects in the same PyCell design; if this *name* is already being used as the name of a physical component, then an exception is raised. In addition, this *name* should represent a scalar quantity; otherwise, an exception is raised.

transform(Transform *trans*) – applies the transform *trans* passed as a parameter to all of the physical components in this grouping.

ungroup(Grouping *owner*=None, bool *all*=False) – this method transfers all members of this Grouping object to the *owner* grouping, and then destroys this Grouping object. If this *owner* grouping is None, then the members are transferred to the DloGen design object. If *all* is True, then all intermediate-level Groupings contained within this Grouping object will also be ungrouped. This Grouping object is destroyed by destroying the Container object for this Grouping object. Note that none of the individual members of this Grouping object are destroyed.

Inherited Methods: This Grouping class inherits all methods defined for the PhysicalComponent class.

Attributes:

In addition to these methods for this Grouping class, there is also an attribute (or property) defined for this Grouping class, described as follows:

name – the name for this Grouping object

Note that this **name** attribute is also settable, so that it can be used in the same way as the "**getName**()" and "**setName**()" methods.

Examples:

```
r1 = Rect(Layer('metal1'), Box(0, 0, 5.0, 8.0))
p1 = Polygon(Layer('metal2'), [Point(0,0), Point(10,10), Point(20,0)])
a1 = Arc(Layer('poly1'), Box(0, 0, 5.0, 8.0), 1.6, 2.3)
# create a Grouping object, and add these objects to it
g1 = Grouping('myGroup')
g1.add(r1)
g1.add(p1)
g1.add(a1)
g1.getName()      # returns 'myGroup'
g1.getBBox()
```

Feedback

```
g1.getBBox(Layer('metal1'))
g1.getBBox(Layer('metal2'))
g1.getBBox(Layer('poly1'))
g1.getBBox([Layer('metal1'), Layer('metal2')])
g1.getBBox([Layer('metal1'), Layer('metal2')], False)
g1.moveBy(-2.0, 3.0)
g1.moveTowards(NORTH, 2.0)
g1.moveTo(Point(3,6), NORTH_WEST)
g1.rotate90()
g1.rotate180()
g1.rotate270()
g1.mirrorX()
g1.mirrorY()
# get all physical components in this Grouping object
compList = g1.getComps()
print 'component list = ', compList
# select physical components by using index value
g1.getComp(0)     # returns rectangle "r1"
g1.getComp(1)     # returns polygon "p1"
g1.getComp(2)     # returns arc "a1"
# ungroup all components in this Grouping object;
# the components are returned to the DloGen design
g1.ungroup()
# ungroup everything in the DloGen design object
self.makeGrouping().ungroup(all=True)
```

CompoundComponent

Description: The CompoundComponent class is the base class which is used to create objects composed of multiple physical components. This allows for the construction of more complicated compound objects from a collection of more primitive components. This CompoundComponent class is derived from the base Grouping class, and provides a locking mechanism on membership and members. When a compound component has been locked, then membership in the CompoundComponent object can not be modified; no new members can be added, and no existing members can be deleted. In addition, none of the individual members can be changed by any modification methods; the compound component can only be modified as a complete object. For example, when a compound component is locked, then this compound component can be mirrored or rotated as a unit, but no member components can be individually mirrored or rotated.

Several higher-level layout objects provided in the Python API, such as the Bar, ContactRing, RoutePath and MultiPath classes, are derived from this base CompoundComponent class. Note that these compound component objects are always in a permanently locked state, so that these pre-defined compound components can not be unlocked by the user. If the **ungroup**() method is used on any of these built-in compound component objects, then the compound component object will be destroyed, and the individual components will remain as separate physical components in the design.

Feedback

Creation:

CompoundComponent(string *name*, *components*=None) – creates a CompoundComponent object, where the *name* parameter specifies the name to be used for this compound component. The optional *components* parameter specifies either a single physical component, or a list of physical components. Each of these physical components will be added to this compound component, after it is created. The PyCell author can first create an empty CompoundComponent object, and then add components (using the "**add**()" method), after the primitive components have been created. Alternatively, these primitive components can first be created, and then the compound component can be created, which will use these components. Note that the compound component is constructed in an unlocked state; it is expected that any derived class would lock the compound component at the end of the derived class creation method.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this CompoundComponent object. Note that the physical components contained within this CompoundComponent object are also cloned. In addition, the lock state for this CompoundComponent will also be cloned.

destroy() – unlocks this compound component, and invokes the **destroy**() method for the base Grouping class, which will destroy all of the components contained in this compound component, as well as the Container object for this base Grouping class.

flatten() – flattens this CompoundComponent into the lowest-level physical components which are contained inside this CompoundComponent. These physical components are then used to create a Grouping object, which is then returned by this method. Note that this CompoundComponent will be destroyed by the flattening process, and replaced with the Grouping object which is returned by this method.

isLocked() – returns True if this CompoundComponent is locked, and False otherwise.

lock() – locks this CompoundComponent object, so that members of this CompoundComponent object can not be modified. In particular, members can not be added to or removed from this CompoundComponent object.

mirrorX(Coord *yCoord*=0) – temporarily unlocks this compound component, invokes the **mirrorX**() method for the base Grouping class, and then restores the lock state.

mirrorY(Coord *xCoord*=0) – temporarily unlocks this compound component, invokes the **mirrorY**() method for the base Grouping class, and then restores the lock state.

moveBy(Coord *dx*, Coord *dy*) – temporarily unlocks this compound component, invokes the **moveBy**() method for the base Grouping class, and then restores the lock state.

moveTo(Point *destination*, Location *handle*=Location.CENTER_CENTER, ShapeFilter *filter*=ShapeFilter()) – temporarily unlocks this compound component, invokes the **moveTo**() method for the base Grouping class, and then restores the lock state.

moveTowards(Direction *dir*, Coord *d*) – temporarily unlocks this compound component, invokes the **moveTowards**() method for the base Grouping class, and then restores the lock state.

rotate90(Point *origin*=None) – temporarily unlocks this compound component, invokes the **rotate90**() method for the base Grouping class, and then restores the lock state.

rotate180(Point *origin*=None) – temporarily unlocks this compound component, invokes the **rotate180**() method for the base Grouping class, and then restores the lock state.

rotate270(Point *origin*=None) – temporarily unlocks this compound component, invokes the **rotate270**() method for the base Grouping class, and then restores the lock state.

transform(Transform *trans*) – temporarily unlocks this compound component, invokes the **transform**() method for the base Grouping class, and then restores the lock state.

snap(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) – temporarily unlocks this compound component, invokes the **snap**() method for the base PhysicalComponent class, and then restores the lock state. Note that this method does not apply the snapping operation to each component in this compound component; instead, it performs the snapping operation on the entire compound component as a whole.

snapX(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) – temporarily unlocks this compound component, invokes the **snapX**() method for the base PhysicalComponent class, and then restores the lock state. Note that this method does not apply the snapping operation to each component in this compound component; instead, it performs the snapping operation on the entire compound component as a whole.

snapY(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) – temporarily unlocks this compound component, invokes the **snapY**() method for the base PhysicalComponent class, and then restores the lock state. Note that this method does not apply the snapping operation to each component in this compound component; instead, it performs the snapping operation on the entire compound component as a whole.

snapTowards(Grid *grid*, Direction *dir*, ShapeFilter *filter*=ShapeFilter()) – temporarily unlocks this compound component, invokes the **snapTowards**() method for the base PhysicalComponent class, and then restores the lock state. Note that this method does not apply the snapping operation to each component in this compound component; instead, it performs the snapping operation on the entire compound component as a whole.

Feedback

ungroup() – unlocks this compound component, and invokes the **ungroup**() method for the base Grouping class. Note that this will destroy the compound component, but will not destroy any of the individual components contained in this compound component.

unlock() – unlocks this CompoundComponent object, so that members of this CompoundComponent object can be modified. In particular, members can be added to or removed from this CompoundComponent object.

Inherited Methods: This CompoundComponent class inherits all methods defined for the Grouping class, as well as all methods defined for the PhysicalComponent class.

Examples:

```
# create sample compound component using 4 rectangles
rect1 = Rect(Layer('metal1'), Box(0,0,1,1))
rect2 = Rect(Layer('metal1'), Box(0,0,1,1))
rect3 = Rect(Layer('metal1'), Box(0,0,1,1))
rect4 = Rect(Layer('metal1'), Box(0,0,1,1))
# create compound component, and add rectangles
cc1 = CompoundComponent('Rectangles')
cc1.add(rect1)
cc1.add(rect2)
cc1.add(rect3)
cc1.add(rect4)
# check that compound component is not locked
cc1.isLocked()          # returns False
# move rectangles into new positions
rect2.moveBy(2,0)
rect3.moveBy(2,2)
rect4.moveBy(0,2)
# now lock this compound component
cc1.lock()
# try various operations on components, should all fail
rect1.moveBy(1,1)
rect2.rotate90()
rect3.mirrorX()
rect4.destroy()
# can not remove members from locked compound component
cc1.remove(rect1)
# but can operate on compound component as an entire unit
cc1.moveBy(1,1)
# ungroup compound component, rectangles now individual objects
cc1.ungroup()
# now try various operations on components, should all succeed
rect1.moveBy(1,1)
rect2.rotate90()
rect3.mirrorX()
rect4.destroy()
```

Derived CompoundComponent Classes

There are several classes which are derived from the CompoundComponent class. These classes include the Bar class, the ContactRing class, the DeviceContact class, the Path class and the RoutePath class. These classes are CompoundComponent classes, since they are composed of several different physical component objects. For example, the Bar class is composed of a Rect rectangle object, along with one or more Contact objects, which are used to make connections between different layers. The ContactRing class is a collection of Contact objects (and an optional fill rectangle object), while the DeviceContact class is a collection of Rect rectangle objects. The RoutePath class is made up of a collection of Rect rectangle objects (as well as Pin objects and Net objects). The MultiPath class is a collection of Path object shapes, which are combined to provide a MultiPath object, which can be used to connect a set of points.

Bar

Description: The Bar class is used to provide simple routing on a single Layer object or between different Layer objects. This Bar class represents a horizontal or vertical route, to which vertical or horizontal connections can be made. These different perpendicular connections can be made either on the same layer or on different layers. If the connection is on different layers, then one or more contacts will be needed to make the necessary connections. Thus, this Bar object consists of a horizontal or vertical route rectangle, along with one or more contacts (if connections are being made between layers).

Note that DRC rules are taken into account when this Bar object is created or modified. For example, minimum width and minimum area design rules for the layer on which this Bar is defined will be used. As a consequence, in some cases (depending upon the technology file used to construct the Bar) the created Bar may be larger than the size requested when the Bar was created. The width will be increased as necessary to meet minimum width rules, while the length will be increased as necessary to meet minimum area rules. Note that the width of the Bar will not be changed, unless it is necessary to meet minimum width design rules.

Creation:

Bar(Layer *layer*, Direction *dir*, string *node*='', Point *point1*=Point.INVALID, Point *point2*=Point.INVALID, Location *anchor* = Location.LOWER_LEFT, string *name* =) – creates a Bar object, where the underlying Bar rectangle is constructed on the given *layer* Layer object. The *dir* Direction of this Bar rectangle is specified using either the Direction `NORTH_SOUTH` (vertical) or `EAST_WEST` (horizontal). An exception will be raised for any other Direction objects. The *node* string specifies the name of the net to which this Bar object belongs in the current design; if this *node* parameter is an empty string, then no connectivity will be used. The two Point parameters *point1* and *point2* are used with the *anchor* Location to specify the size and/or position of the Bar rectangle. The Point parameters are the two diagonal points of the Bar rectangle bounding box, while the *anchor* Location is the reference point for the Bar object rectangle bounding box. This *anchor* remains stationary, whenever the Bar rectangle needs to be resized to meet DRC

rules (such as minimum area rules). The *name* parameter is an optional name for the grouping consisting of the Bar rectangle and any Contact objects.

Note that when this Bar object is created, it does not have any contacts. Contact objects are then added to this Bar object through the use of the "**addContact**()" method.

Methods:

addContact(Layer *layer*, Point *point1*, Point *point2* = Point.INVALID, Direction *anchor* = Direction.CENTER, Direction *routeDir* = Direction.NONE) – adds a new Contact object to this Bar, using the *layer* Layer object parameter. The *point1* and *point2* parameters are used to define this Contact object. This Bar will be grown, if this added contact is outside the current span of this Bar. The optional *anchor* Direction is used to specify the behavior of this Contact object whenever it needs to be grown. Note that only one layer needs to be specified for this Contact object, since the Bar object layer is used as the other layer when the contact is created.

clearContacts() – removes all of the Contact objects associated with this Bar. Note that this method can be used to reposition Contact objects, whenever a transistor is resized.

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this Bar object.

destroy() – destroys all of the components in this Bar (Contact objects and Rect rectangle object), as well as the Grouping object which contains all of these components. Note that the underlying OpenAccess object will also be destroyed and removed from the database.

extendTo(Point *point*) – extends this Bar in the Direction of this Bar object (horizontally or vertically) so that the end of this Bar is collinear with the specified *point* parameter. Note that if the x or y coordinate range of the specified *point* is contained within the x or y coordinate range of the current Bar rectangle, then this Bar will not be changed. Thus, unlike the **stretchTo**() or **stretchToCoord**() methods, this method will not shrink this Bar object; this method can only grow this Bar object.

getContacts() – this method returns a uniform list of all Contact objects which are contained in this Bar. Note that this list will be a `snapshot` of the contacts present when this method is called; this list will not be updated as any contacts are added to this Bar.

getDirection() – returns the Direction object for this Bar; this will be either the either the `NORTH_SOUTH` (vertical) or `EAST_WEST` (horizontal) Direction.

getLayer() – returns the Layer object on which this Bar is defined.

getRect() – returns the route rectangle for this Bar object.

getRoutePathIntersectBox(RouteTarget *fromTarg*) – returns the box which is the intersection of this Bar object and the RoutePath object which has been generated, if this Bar object is being used as the routing target for a straight-line RoutePath object (see the

section on the RoutePath object). This method is useful when the designer is generating their own Contact objects.

stretchTo(Point *point*) – stretches this Bar in the Direction of this Bar object (horizontally or vertically), so that the end of this Bar is collinear with the given Point *point* parameter. If the value of this *point* parameter causes this Bar to shrink, and this Bar has contacts, then this Bar will only shrink by an amount that keeps all of the Bar Contact objects connected. Note that this method does not change the width of the Bar, only the length. In addition, minimum width and minimum area design rules will be used to determine the size of the Bar rectangle.

stretchToCoord(Direction *dir*, Coord *value*) – stretches this Bar using the edge specifed by the *dir* direction, until the edge of this Bar aligns with the coordinate *value*. If the value of this *value* parameter causes this Bar to shrink, and this Bar has contacts, then this Bar will only shrink by an amount that keeps all of the Bar Contact objects connected. The specified *dir* direction must be a sub-direction of the Bar direction (NORTH or SOUTH for vertical, and SOUTH or EAST for horizontal), oran exception is raised. In addition, minimum width and minimum area design rules will be used to determine the size of the Bar rectangle.

trim() – trims this Bar to the size required to connect all of the contacts for this Bar. This can be used after all contacts are added to this Bar, to compact it to its minimum size. Note that this method does not change the width of the Bar, only the length. In addition, minimum width and minimum area design rules will be used to determine the size of the Bar rectangle. If there are no contacts, then this Bar will be trimmed to the minimum size for a rectangle, based upon minimum width and minimum area design rules.

Inherited Methods: This Bar class inherits all methods defined for the Grouping class, as well as the PhysicalComponent class.

Examples:

```
bar1 = Bar(Layer('metal1'), EAST_WEST, 'A')
bar1.addContact(Layer('metal3'), Point(1.0,1.0), Point(2.0,2.0))
bar1.getContacts()       # returns list of contacts
bar1.trim()              # trim Bar, but include contacts
bar1.clearContacts()
bar1.trim()              # trim Bar to minimum size
bar1.getLayer()          # returns Layer('metal1')
bar1.getDirection()      # returns Direction.EAST_WEST
bar1.getRect()           # returns Bar route rectangle
# extend Bar to Point(3.0, 3.0) to increase size of Bar
bar1.extendToPoint(Point(3.0, 3.0))
bar1.getRect()           # returns new Bar route rectangle
```

Feedback

ContactRing

Description: The ContactRing class provides the ability to create a guard ring, which can be used to isolate a set of devices in a design from external noise, as well as to also prevent latchup. This ContactRing object consists of four guard ring contacts, which are placed in the NORTH, SOUTH, EAST and WEST directions. Thus, this ContactRing object will consist of these four abutting contacts (top, bottom, left and right), along with an optional fill rectangle. These four contacts will be spaced using minimum spacing rules for the current technology, so that there will be no DRC errors. The user can specify that a fill layer be used to construct an optional fill rectangle on the specified fill layer, which will enclose this ContactRing object. This fill layer should be either a nwell or pwell well layer. In addition, the user has the ability to specify a gap parameter value when this ContactRing object is created. If this gap parameter is provided by the user, then a gap of the specified distance is inserted between the left (WEST) and top (NORTH) contact, on the top side of this contact ring. Note that the spacing of this gap will be constructed so that there will be no DRC errors for the given technology. The gap spacing would typically be used in cases where the contact ring would be large enough to cause a current loop to be induced from a magnetic field; this gap spacing would be used to `break` such an induced current loop.

There are a large number of DRC rules which are taken into account when this ContactRing object is created or modified. This includes such rules as minimum width, minimum area, minimum via spacing and minimum enclosure rules. These design rules are used when each of the contacts for this ContactRing is created. As a consequence, in some cases (depending upon the technology file used to construct the ContactRing) the created ContactRing may be larger than the size requested when the ContactRing was created. For example, via spacing and enclosure rules may necessitate that a wider Contact object be created to meet these minimum via spacing and enclosure design rules.

When this ContactRing object is created, it will be placed to enclose all of the design instances and shapes in the current design. This approach makes it easy to add a contact ring after the body of a design has already been implemented. It is also possible to create multiple contact rings, each of which encloses the previously created contact ring. In addition, the designer can specify the dimensions of an enclosing box, around which the contact ring will be generated. This approach allows the designer to handle the special cases where the contact ring has to be larger than would otherwise be indicated by the minimum enclosure design rule values.

Although the default ContactRing object consists of four contacts, it is possible to construct a contact ring consisting of only a singe contact strip, or only a subset of these four default contacts. This can be done by using the *locations* parameter when the ContactRing object is created. Note that if there are less than four contacts, then any gap spacing will be ignored, as there will be no need to prevent any possible current loops.

Feedback

Note that this contact ring can be `chopped`, using the **chop**() method. For example, it may be necessary to cut through this ContactRing object for routing purposes; this may be necessary for process technologies having a smaller number of metal layers.

Creation:

ContactRing(Layer *layer1*, Layer *layer2*, string *node*='', LayerList *addLayers* = None, Coord *width* = 0, Coord *gap* = 0, LayerList *fillLayers* = None, string *name* = ,

DirectionList *locations*=None, ShapeFilter *ruleFilter*=None, Box *encloseBox*=None, PhysicalComponent *encloseComp*=None, bool *overlapContact*=False, bool *fillToBoundary*=False, dict *options*=None) – creates a ContactRing object, using the *layer1*, *layer2*, *addLayers* and *width* parameters to create the four different Contact objects which make up this contact ring. The *width* parameter is the width of each contact; if not specified, then the minimum width required to construct a single row (or column) of contact cuts will be used by default. The *node* parameter is the name of the net to which these contacts are connected; each of these four contacts will be connected to the same net; if this *node* parameter is an empty string, then no connectivity will be used. The optional *gap* parameter specifies the width of a gap space between the left (WEST) and top (NORTH) contacts. The optional *fillLayers* parameter specifies the layers which should be used to generate enclosure rectangles for this ContactRing; these layers should be pwell or nwell well layers, diffusion layers, implant layers or high voltage layers. The optional *name* parameter is used to specify a name for this ContactRing (which will be the name of the Grouping object for this ContactRing). If this *name* parameter is not specified, then the default name Unique::Name(`ContactRing`) will be used. The optional *locations* parameter is used to specify the list of Directions (NORTH, SOUTH, EAST, WEST) for which contacts should be generated; if this list is empty, then contacts for all four directions will be generated. Note that if there are fewer than four contacts specified by this *locations* parameter, then the *gap* parameter will be ignored. The optional *ruleFilter* parameter is used to suppress layer design rules which should not be considered when constructing this contact ring; the use of this parameter will potentially generate a smaller contact ring. The optional *encloseBox* and *encloseComp* parameters are used to specify the enclosing box (or physical component) around which this ContactRing object should be generated. If both of these parameters are used at the same time, then the *encloseComp* parameter will be ignored. Note that there will be no checking for DRC minimum spacing rules when the optional *encloseBox* parameter is specified; the dimensions of the enclosing box will be used without performing any design rule checking. If the optional *overlapContact* parameter is specified, then this ContactRing can be overlapped with another mirror imaged ContactRing, by fully overlapping the mirror imaged contacts of each ContactRing. Using this *overlapContact* option will ensure that contact cuts for the overlapped contacts are constructed without DRC errors. If the optional *fillToBoundary* parameter is specified, then the generated fill layer shapes will be extended as necessary to fully enclose all of the components of this ContactRing. The optional *options* parameter is used to specify values for special options which should be used when the ContactRing object is constructed. This *options* parameter is a Python dictionary, where the keys are pre-defined option names, and the values are the values for these options. Valid option names include:

`infiniteLayers`. The `infiniteLayers` option is specified as a list of layers; these layers will be considered for DRC spacing calculations, but no shapes will be generated for these layers. The `optimizeFillLayers` option obsolete and will not produce any change in the layout.

Methods:

chop(Box *cutBox*, LayerList *cutLayers*, LayerList *mendLayers*=None, Layer *joinLayer*=None) – chops this ContactRing object, by cutting out the interconnect layers specified by the *cutLayers* parameter. The intersection of the contact ring contacts and the box defined by the *cutBox* parameter specifies the portion of the ContactRing object to be chopped. The *mendLayers* parameter specifies a list of layers which will be mended, by creating a new shape which reconnects the two new contacts which will be created by this chopping operation. If this *mendLayers* parameter is not specified, then all layers will be mended, except for the layers specified by the *cutLayers* parameter. The layers specified by this *mendLayers* parameter should be either interconnect layers or one of the additional layers specified when this contact ring was created. The *joinLayer* parameter specifies a layer which should be used to join the cut contacts; this should be a layer which is not one of the interconnect layers or the additional layers specified when this ContactRing object was created. This *joinLayer* is used for joining cut contacts, to provide connectivity for this contact ring. If this *joinLayer* parameter is not specified, then the cut contacts in the chopped ContactRing object will not be joined. In this case, the designer should make sure that connectivity for this contact ring is maintained, as will automatically be done when the *mendLayers* parameter contains a metal interconnect layer. If the layers specified by the *cutLayers*, *mendLayers* or *joinLayer* parameters are not as described above, then an exception will be generated. Note that this **chop**() method can not be undone; if the chopped contact ring is not cut as desired, then this contact ring should be destroyed and a new one created. It should also be noted that although this **chop**() method attempts to satisfy all applicable design rules, there is some possibility for design rule errors to result, especially if there are multiple chop operations performed on the same ContactRing object.

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this ContactRing object.

destroy() – destroys this ContactRing object, by destroying all of the contacts, optional fill rectangle, and the Grouping object which contains all of these components. Note that the underlying OpenAccess object will also be destroyed and removed from the database.

getContact(Direction *dir*) – returns the Contact object for the specified *dir* Direction. For example, the NORTH Direction would return the top Contact object for this ContactRing, while the SOUTH Direction would return the bottom Contact object. This Direction parameter *dir* should be one of NORTH, SOUTH, EAST or WEST; any other Direction value causes an exception to be raised.

Inherited Methods: This ContactRing class inherits all methods defined for the CompoundComponent and Grouping classes, as well as the PhysicalComponent class.

Feedback

Examples:

```
# create contact ring object
contactRing1 = ContactRing(Layer('metal3'), Layer('diff'), 'nbulk',
 width=1.0, gap=0.01,
addLayers=[Layer('nimp')], fillLayers=[Layer('nwell')])
# get the top contact
contact1 = contactRing1.getContact(NORTH)
# get the different layers for this contact
contact1.getLayer1()    # returns  Layer('metal3')
contact1.getLayer2()    # returns  Layer('diff')
# get all layers for this contact
contact1.getLayers()
# also get the reference box for this contact
contact1.getRefBox()
# construct contact ring with only three contacts
contactRing2 = ContactRing(Layer('metal1'), Layer('diff'), 'A',
 locations=[EAST,SOUTH,WEST])
# create contact ring, using an enclosure box
box1 = Box(0, 0, 10.0, 10.0)
contactRing3 = ContactRing(Layer('metal3'), Layer('diff'), 'B',
 encloseBox=box1)
# chop contact ring in the top contact
cutBox = Box(4,8,5,12)
contactRing1.chop(cutBox, cutLayers=[Layer('metal1')])
```

DeviceContact

Description: The DeviceContact class provides a connection between two vertically adjacent interconnect layers in a layout design. An interconnect layer is either a diffusion, polysilicon or metal layer in a design. Note that these interconnect layers are required to be vertically adjacent, unlike the interconnect layers used by the Contact class. Although the Contact class provides general purpose contacts which can be used for any purpose, the DeviceContact class provides more specialized contacts which are meant to be used for device construction, versus other applications, such as routing contacts. Note that while the Contact class is derived from the basic Instance class, this DeviceContact is derived from the base CompoundComponent class. As a result, shapes within a device contact can easily be directly assigned to nets or pins in the layout design. In addition, these shapes can also be named for use with the hierarchical reference classes.

The DeviceContact object can have one or more via cuts, which are shapes on the via layer, which are used to connect the two interconnect layers. This number of cuts can be increased in order to improve manufacturing yield, reduce parasitic resistance, or to increase current capacity for the device being designed. Note that when this device contact object is created, the minimum number of cuts in either the horizontal or vertical directions can be specified. These square fixed-size cut shapes will be constructed to meet the required DRC rules for the given technology. In addition, the spacing between these via cuts can also be specified (by means of the GapStyle parameter). Note that

unlike the Contact class, the number of via cuts will not be automatically adjusted whenever the contact rectangle shapes are modified; if desired, the designer would need to separately adjust the number of via cuts.

Note that there are a large number of DRC rules which are automatically taken into account when this DeviceContact object is created or modified. This includes such rules as minimum width, minimum area, minimum via spacing and minimum via enclosure (or extension) design rules. These minimum via spacing rules include adjacent neighbor via spacing rules, as well as large array via spacing rules. In addition, the minimum same net spacing rule will be used, if this design rule exists in the technology file. If desired, the designer has the option to override these default design rule values when the device contact is created, through the use of optional parameters in the device creation method. This approach provides maximum flexibility for the designer, so that this device contact can be more easily customized. In addition, this DeviceContact class is implemented in Python and the Python source code for this PyCell is shipped as part of the standard PyCell Studio product (in $CNI_ROOT/pylib/cni/devices/deviceContact.py). Thus, the designer can either modify this existing source code or use this DeviceContact class as a base class from which to derive their own more specialized contact classes.

Creation:

DeviceContact(Layer *layer1*, Layer *layer2*, Box *box*, GapStyle *gapStyle*=GapStyle.MIN_CENTER, ulist[int] *minCuts*=None, ulist[float] *layer1Ext*=None, ulist[float] *layer2Ext*=None, ulist[float] *viaSpace*=None, ulist[float] *viaSize*=None, Direction *routeDir1* = None, Direction *routeDir2* = None, string *name* =) – creates a DeviceContact object which connects *layer1* to *layer2*. These *layer1* and *layer2* Layer objects should be either diffusion, polysilicon or metal layers. In addition, these two layers must be adjacent interconnect layers, or else an exception will be generated. The box parameter specifies the area which will be completely filled on *layer1* to define this DeviceContact object. The *gapStyle* parameter specifies the distribution of spacing between via cuts; by default, these via cuts will be packed together as close as possible in the center. The *minCuts* parameter specifies the minimum number of cuts as a list of one or two values. If two values are specified, then the first value is used for the minimum number of horizontal cuts, while this second number is used for the minimum number of vertical cuts. Note that if one of these numbers is zero, then the number of cuts in that direction is not restricted. If only one value is specified, then this value is used for both horizontal and vertical cuts.

The *layer1Ext*, *layer2Ext*, *viaSpace* and *viaSize* parameters are provided as optional device construction parameters for the DeviceContact object. These parameters can be used to override the default design rule values which are used by the DeviceContact. If these parameters are not specified, the DeviceContact will look up and use the enclosure (or extension) design rule values for these layers from the current technology file. However, if the values for any of these parameters are specified, then these values will automatically override the values of any applicable enclosure (or extension) design rules which are defined for these layers. The *layer1Ext* parameter is a list of 1, 2 or 4 values for the enclosure (or extension) of the *layer1* rectangle over the via cuts for this

Feedback

DeviceContact. If one value is specified, then it will be used for all four sides; if two values are specified, then they will be used as dual x-y minimum enclosure values. If four values are specified, then these values will be used for the left, bottom, right and top sides respectively of the rectangle. In a similar manner, the *layer2Ext* parameter specifies the enclosure (or extension) values for the *layer2* rectangle. The *viaSpace* parameter is a list of 1 or 2 values for the horizontal and vertical spacing for the via cuts; if only a single value is given, it will be used for both horizontal and vertical spacing. (Note that this *viaSpace* value may not be exact when the *gapStyle* is DISTRIBUTE). The *viaSize* parameters is a list of 1 or 2 values for the width and height of each via cut; if only a single value is given, then it will be used for the both the width and height of via cuts.

The *routeDir1* parameter specifies the routing direction for *layer1*, while *routeDir2* specifies the routing direction for *layer2*. These routing direction values should be either EAST_WEST or NORTH_SOUTH; if not specified, the default value is EAST_WEST. Note that these routing directions are only used, if the default design rule lookup is needed to determine the enclosure (or extension) for the given layer, and the technology file uses min dual extension design rules for that layer. In that case, the first value is used for the horizontal enclosure, while the second value is used for the vertical enclosure.

The *name* parameter is an optional string parameter for the name of this DeviceContact component.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this DeviceContact object.

getLayer1() – returns the first interconnect layer for this device contact.

getLayer2() – returns the second interconnect layer for this device contact.

getNumHVCuts() – returns the current number of cuts in the horizontal direction, and the current number of cuts in the vertical direction. These numbers are returned as a tuple of integers: (numHorCuts, numVertCuts).

getRect1() – Returns rectangle which is defined on the first interconnect layer for this device contact. Note that this rectangle can be directly assigned to a Net or Pin object.

getRect2() – Returns rectangle which is defined on the second interconnect layer for this device contact. Note that this rectangle can be directly assigned to a Net or Pin object.

getViaBBox() – Returns the bounding box of the via cuts for this device contact. This specialized method is provided, since it is generally much faster than the more general-purpose **getBBox**(viaLayer) method.

getViaLayer() – returns the via layer for this device contact.

The following three methods (**stretch**(), **stretchToCoord**() and **stretchToPoint**()) are used to stretch the device contact rectangle defined on one of the device contact layers.

Feedback

Note that this stretching will not allow this device contact rectangle to violate the enclosure design rule values for the via cuts which were specified when this device contact was created. If it is necessary to modify the device contact rectangle such that enclosure design rules are violated, then the designer should directly resize this rectangle after using the **getRect1**() or **getRect2**() methods to obtain this device contact rectangle. In addition, unlike the **Contact** class, these stretch methods will not change the field of via cuts; only the specified device contact rectangle will be modified, and additional via cuts will not be automatically generated by these stretch methods.

stretch(Layer layer, Box *box*) – stretches the device contact rectangle on the specified layer *layer* to match the box which is passed using the *box* parameter. Note that no warnings will be generated if the device contact rectangle cannot be stretched as specified by the *box* parameter value.

stretchToCoord(Layer *layer*, Direction *dir*, Coord *value*) – stretch the device contact rectangle defined on the specified *layer* in the specified direction *dir*, until this rectangle is aligned with the specified coordinate *value*. Note that the *dir* direction can be specified as either a single direction (such as NORTH or SOUTH) or as two directions (such as NORTH_EAST or SOUTH_WEST). All specified sides of the device contact rectangle will be stretched as required to align with the specified *value* coordinate.

stretchToPoint(Layer *layer*, Direction *dir*, Point *point*) – stretch the device contact rectangle defined on the specified *layer* in the specified direction *dir*, until this rectangle is aligned with the specified *point*. The *dir* direction can be specified as either a single direction (such as NORTH or SOUTH) or as two directions (such as NORTH_EAST or SOUTH_WEST). All specified sides of the device contact rectangle will be stretched as required to align with the specified *point.*

Inherited Methods: This DeviceContact class inherits all methods defined for the CompoundComponent and Grouping classes, as well as the PhysicalComponent class.

Examples:

```
# create first device contact using default design rules
dc1 = DeviceContact(Layer('diff'), Layer('metal1'),
                    Box(0,1,0,1), gapStyle=GapStyle.DISTRIBUTE,
                    minCuts=[1,2], routeDir1=NORTH_SOUTH)
dc1.getLayer1()    # returns Layer('diff')
dc1.getLayer2()    # returns Layer('metal1')
dc1.getRect1()     # returns diffusion rectangle
dc1.getRect2()     # returns metal1 rectangle
dc1.getViaLayer()  # returns Layer('contact')
dc1.getViaBBox()   # returns bounding box for contact via cut
# get number of horizontal and vertical cuts
(numHorCuts, numVertCuts) = dc1.getNumHVCuts()
# create second device contact overriding design rule values
layer1Ext = [0.08, 0.1]   # dual x-y enclosure for diffusion
layer2Ext = [0.04, 0.02]  # dual x-y enclosure values for metal1
viaSpace = [0.11]         # horizontal and vertical spacing value
```

Feedback

```
viaSize = [0.18]          # fixed width and length of via cut
dc2 = DeviceContact(Layer('diff'), Layer('metal1'),
                    Box(0,0,0.5,1), gapStyle=GapStyle.MIN_CENTER,
                    layer1Ext=layer1Ext, layer2Ext=layer2Ext,
                    viaSpace=viaSpace, viaSize=viaSize,
                    minCuts=[1,2], routeDir1=NORTH_SOUTH)
# get number of horizontal and vertical cuts
(numHorCuts, numVertCuts) = dc2.getNumHVCuts()
# stretch second contact diffusion layer to meet first contact
dc2.stretchToCoord(Layer('diff'), NORTH, dc1.getBBox().getTop())
# Note that the number of cuts is not increased,
# even though the contact was increased in size.
(numHorCuts, numVertCuts) = dc2.getNumHVCuts()
```

MultiPath

Description: The MultiPath class provides the ability to group together a number of different Path Shape objects, based upon the Master Path object which is defined for this MultiPath CompoundComponent object. Once the Master Path object has been defined for this MultiPath class, then methods are provided to generate additional sub-paths, based upon this original Path object. For example, a new Path object can be generated by applying an offset to the set of points which define the original Path object. In addition, an additional sub-path Path object can be generated by creating a new sub-path which encloses the original Path object, by expanding the width of this original Path object.

Other capabilities for this MultiPath object include the ability to generate a field of sub-rectangles for the Master Path, as well as the ability to chop the MultiPath into many segments. Note that the construction of sub-rectangles for the Master Path is typically used when constructing via cuts.

Note that when a MultiPath object is created, the path style for the beginning and ending points which define the MultiPath object can be specified, using the PathStyle class. This PathStyle value can be one of the following enumerated values: TRUNCATE, EXTEND, ROUND or VARIABLE. The TRUNCATE path style provides no extension beyond the beginning and ending points, while the EXTEND path style will extend both beginning and ending points by one half the width of the MultiPath. The ROUND path style will generate an octagonal extension which results in three edges which extends both the beginning and ending points by one third the width of the MultiPath. The VARIABLE path style allows the beginning and ending points to have different extensions; the designer provides the desired extension values for the beginning and ending points.

Creation:

MultiPath(Layer *layer*, PointList *points*, Coord *width*, Direction *justify*=EAST_WEST, Coord *sep*=0, PathStyle *style*=PathStyle.TRUNCATE, Coord *beginExt*=0, Coord *endExt*=0, bool *choppable*=True, ulist[Box] *chopBoxes*=None, string *node*=, string *masterName*=, string *name*=) – creates a MultiPath object using the *points*, *layer* and *width* parameters, creating a Master Path object on the given layer, based upon this list of points, with the

given width. If the *width* parameter value is not specified, then the minimum width for the specified *layer* is used by default. The points specified by the *points* parameter are justified by the *justify* parameter value, relative to the Master Path. These points are also separated from the Master Path by the value of the *sep* separation parameter value, relative to the Master Path. Note that the *beginExt* and *endExt* extension parameter values are only used to specify path extensions when the *style* path style is set to VARIABLE. The *choppable* parameter specifies whether the Master Path can be chopped or not, while the *chopBoxes* parameter specifies a uniform list of boxes which should be used to chop the Master Path. The *node* parameter specifies the name of the net to which the Master Path should be connected. The *masterName* parameter specifies the name which should be used for the CompoundComponent which contains the Master Path. The string *name* parameter is used to give a name to this MultiPath object. If this *name* parameter is not specified, then a default name will be automatically generated.

Methods:

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this MultiPath object.

createEnclosureSubpath(Layer *layer*, Coord *encl*=0*,* Coord *beginEncl*=None, Coord *endEncl*=None, bool *choppable*=True, string *node*=*,* string *name*=) – creates an enclosure subpath for this MultiPath object. This subpath is created using the point list for this MultiPath, but the enclosure of this subpath is determined by the value of the *encl* parameter. The width of this enclosure subpath will be calculated as the width of the Master Path minus twice the value of the *encl* parameter. Note that for positive *encl* enclosure values, the subpath will be created inside the master path, while for negative enclosure values, the created subpath will enclose the master path. If this *encl* enclosure parameter is not specified, then the minimum enclosure design rule for this specified *layer* and the Master Path layer will be used by default. If this minimum enclosure design rule does not exist in the technology file, then an exception is raised. The *beginEncl* and *endEncl* parameters specify the begin-enclosure and end-enclosure of this enclosure subpath relative to the begin point and end point of the Master Path. If the *beginEncl* parameter is not specified, then *encl* will be used by default. If the *endEncl* parameter is not specified, then *encl* will be used by default. The *choppable* parameter specifies whether this enclosure subpath should be chopped or not, when the Multi Path object is chopped. The *node* parameter specifies the name of the net to which this subpath should be connected. The string *name* parameter is used to give a name to this subpath. If this *name* string parameter is not specified, then a default name will be automatically generated. Note that this enclosure subpath uses a truncated end style for each end of the subpath. The modified MultiPath object will be returned by this method.

createOffsetSubpath(Layer *layer*, Coord *width*, Direction *justify*=EAST_WEST, Coord *sep*=0, Coord *beginOffset*=0, Coord *endOffset*=0, bool *choppable*=True, string *node*=, string *name*=) – adds a new Path object to this MultiPath, which is created by using the point list for the Master Path, but shifting each point, by adding the value of the *sep* separation parameter, in the specified *justification* direction relative to the Master Path.

The *width* parameter specifies the width of the created subpath; if this *width* parameter is not specified, then the minimum width for this *layer* will be used by default. The *beginOffset* and *endOffset* parameters specify the additional extensions of this subpath, relative to the begin and end points for the Master Path. A positive offset value indicates that the begin or end point should be extended, while a negative value means that the point should be retracted. The *choppable* parameter specifies whether this offset subpath should be chopped or not, when the Multi Path object is chopped. The *node* parameter specifies the name of the net to which this subpath should be connected. The string *name* parameter is used to give a name to this subpath. If this *name* string parameter is not specified, then a default name will be automatically generated. The modified MultiPath object will be returned by this method. Note that the *justification* direction should be a direction defined by the X-axis (EAST, WEST or EAST_WEST); otherwise, an exception is raised.

createSubrectangles(Layer *layer*, Coord *width*=0, Coord *length*=0, Coord *space*=0, GapStyle *gapStyle*=GapStyle.MINIMUM, Direction *justify*=EAST_WEST, Coord *sep*=0, Coord *beginOffset*=None, Coord *endOffset*=None, bool *choppable*=True, string *node*= string *name*=, bool *diagonal*=False, PlaceStyle *parMainPath*=PlaceStyle.WIDTH_PARALLEL_X_AXIS, *fillSchema*=False, bool *distSingleRect*=True, int *layerGridSizeFactor*=2) – creates a field of subrectangles on the specified *layer* for this MultiPath. The *width* parameter specifies the width of each rectangle; if this *width* parameter is not specified, then the minimum width for the specified *layer* is used by default. The *length* parameter specifies the length of each rectangle; if this *length* parameter is not specified, then the minimum width for the specified *layer* is used by default. The *space* parameter specifies the minimum spacing for the rectangles; if this *space* parameter is not specified, then the minimum spacing for the specified *layer* will be used by default. The *gapStyle* parameter specifies how any excess spacing between two rectangles will be divided into gaps. If this *gapStyle* parameter is DISTRIBUTE, then excess spacing is distributed between rectangles; if the value is EVEN_DISTRIBUTE, then excess spacing is distributed evenly between rectangles; if the value is MINIMUM, then rectangles are spaced at minimum distances and excess spacing is located at the end of each subpath segment. The *justify* parameter specifies the justification value for the field of rectangles relative to the Master Path. The *sep* separation parameter specifies the separation of this subpath from the Master Path. The *beginOffset* and *endOffset* paarmeters specify additional extensions of this subpath relative to the begin point and end point for the Master Path. The *choppable* parameter specifies whether this subpath should be chopped or not, when the Multi Path object is chopped. The *node* parameter specifies the name of the net to which this subpath should be connected. The *name* parameter is used to give a name to this subpath. If this *name* parameter is not specified, then a default name will be automatically generated. The *diagonal* parameter specifies whether subrectangles should be created for diagonal (45-degree or 135-degree)

segments of the MultiPath. The modified MultiPath object will be returned by this method. The *parMainPath* parameter specifies how to place the subrectangles as follows:

- LENGTH_PARALLEL_CENTERLINE – the length of the subrectangle is parallel to the master path of the centerline.
- WIDTH_PARALLEL_CENTERLINE – the width of the subrectangle is parallel to the master path of the centerline.
- WIDTH_PARALLEL_X_AXIS – the width of the subrectangle is parallel to x-axis always.

The *fillSchema* parameter specifies whether to fix the violation of the rules for the subrectangle. If only a single subrectangle is created, the *distSingleRect* parameter specifies whether it should be centered and offsets ignored (True) or offset as specified by *beginOffset* and *endOffset* (False). The *layerGridSizeFactor* parameter specifies whether to get the rectangle's expected width and length according to grid resolution; if the *layerGridSizeFactor* parameter is not specified, then the minimum *layerGridSizeFactor* is used by default.

destroy() – destroys all of the components of this MultiPath object (all subpath objects, as well as possible subrectangles), as well as deleting this MultiPath object from the DloGen owner. Note that the underlying OpenAccess object will also be destroyed and removed from the database.

genJustifyPathPoints (PointList *refPoints*, Coord *width*=0, Direction *justify*= EAST_WEST, Coord *sep*=0, Coord *subWidth*=0, Coord *beginOffset*=0, Coord *endOffset*=0) – returns a justified point list which is constructed from the list of reference points specified by the *refPoints* parameter. The *justify* parameter specifies the direction of the justified point list relative to the *refPoints* point list, and the *sep* parameter specifies the separation between the justified point list and the *refPoints* point list. The *subWidth* parameter specifies the width of the subpath built from the list of justified points. The *beginOffset* and *endOffset* parameters specify the offsets for the first and last points of the subpath generated from the *refPoints* point list. If the *justify* direction is not one of the directions defined by the X-axis (EAST, WEST or EAST_WEST), then an exception is raised.

Note that this method is defined as a static method, so it can be called without directly constructing this MultiPath object; for example, `MultiPath.genJustifyPathPoints()`.

getChoppedSubpathPointLists(*name*=) – returns the list of points for the chopped subpath or Master Path specified by the *name* parameter. If this *name* parameter is not specified, then the point list for the Master Path will be returned. If the *name* parameter does not specify the name of an existing subpath or Master Path, then an exception is raised.

getMasterPathName() – returns name assigned to the Master Path for this MultiPath.

Feedback

getSubpathPoints(*name*=) – returns the list of points for the subpath or Master Path specified by the *name* parameter. If this *name* parameter is not specified, then the point list for the Master Path will be returned. If the *name* parameter does not specify the name of an existing subpath or Master Path, then an exception is raised.

mirrorX(Coord *yCoord* = 0) – mirrors all components in this MultiPath about the X coordinate axis. The optional *yCoord* parameter can be used to specify a displacement for the location of the X coordinate axis.

mirrorY(Coord *xCoord* = 0) – mirrors all components in this MultiPath about the Y coordinate axis. The optional *xCoord* parameter can be used to specify a displacement for the location of the Y coordinate axis.

moveBy(Coord *dx*, Coord *dy*) – moves all components in this MultiPath by *dx* units in the x-coordinate and *dy* units in the y-coordinate.

moveTowards(Direction *dir*, Coord *distance*) – moves all components in this MultiPath by *distance* coordinate units in the *dir* parameter direction.

moveTo(Point *destination*, Location *handle*=Location.CENTER_CENTER, ShapeFilter *filter*=ShapeFilter()) – moves all components in this MultiPath, so that the *destination* Point becomes the handle point for the bounding box for this MultiPath at the *handle* location. This bounding box is determined using the *filter* ShapeFilter parameter. If these components do not have any geometry on the layers specified by the *filter* ShapeFilter parameter, then an exception is raised.

rotate90(Point *origin*=None) – rotates all components in this MultiPath by 90 degrees in a counter-clockwise direction. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

rotate180(Point *origin*=None) – rotates all components in this MultiPath by 180 degrees in a counter-clockwise direction. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

rotate270(Point *origin*=None) – rotates all components in this MultiPath by 270 degrees in a counter-clockwise direction. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

setChopBoxes(BoxList *chopBoxes*) – chops the Master Path by the *chopBoxes* parameter.

setChoppable(bool *choppable*) – specifies whether the Master Path can be chopped by the *choppable* parameter.

snap(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) – snaps both the X and Y coordinates of the lower-left vertex point of the bounding box for this MultiPath object to the grid points defined by the *grid* parameter. This bounding box will be determined using the layers specified by the *filter* ShapeFilter parameter. The *snapType* parameter is used to specify the snapping type (CEIL, FLOOR, ROUND or TRUNC) which

Feedback

should be used; if this *snapType* parameter is not specifed, then the snapping type defined by the *grid* Grid object will be used by default. If the bounding box determined by the *filter* parameter is inverted, then an exception is raised.

snapX(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) – snaps only the X coordinate of the lower-left vertex point of the bounding box for this MultiPath object to grid points defined by the *grid* parameter. If the bounding box determined by the *filter* parameter is inverted, then an exception is raised.

snapY(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) – snaps only the Y coordinate of the lower-left vertex point of the bounding box for this MultiPath object to grid points defined by the *grid* parameter. If the bounding box determined by the *filter* parameter is inverted, then an exception is raised.

snapTowards(Grid *grid*, Direction *dir*, ShapeFilter *filter*=ShapeFilter()) – snaps the coordinates of the lower-left vertex point of the bounding box for this MultiPath object to the grid points defined by the *grid* parameter. This bounding box will be determined using the layers specified by the *filter* ShapeFilter parameter. The *dir* Direction parameter is used to specify the one-dimensional direction in which the snapping operation should take place. If the bounding box determined by the *filter* ShapeFilter parameter is inverted, then an exception is raised. If the *dir* direction is not one of the four primary directions (NORTH, SOUTH, EAST or WEST), then an exception is raised.

transform(Transform *trans*) – applies the transform *trans* passed as a parameter to all of the components in this MultiPath object.

Inherited Methods: This MultiPath class inherits all methods defined for the Grouping and CompoundComponent classes, as well as the PhysicalComponent class.

Examples:

```
p1 = PointList([Point(0,0), Point(100,0), Point(100,100),
                            Point(200,100), Point(200,200)])
# use this PointList to create diagonal routing MultiPath
mp1 = MultiPath(Layer('metal1'), width=.10, points=p1,
                masterName='MP', name='path1')
# create an offset subpath for this MultiPath
mp2 = mp1.createOffsetSubpath(Layer('metal2'), width=0.20,
                              sep=10, name='OFFSET_PATH')
# create an enclosure subpath for this MultiPath
mp3 = mp1.createEnclosureSubpath(Layer('metal3'), encl=-0.30)
# create field of subrectangles for this MultiPath object;
# field uses the default minimum width and minimum spacing
mp3.createSubrectangles(Layer('metal4'))
# get name of MultiPath, as well as master path name
mp3.getName()               # returns "path1"
mp3.getMasterPathName()     # returns "MP"
# get Master point list for this MultiPath object
mPointList = mp3.getSubpathPoints('MP')
mPointList == p1            # returns True
```

Feedback

```
mPointList = mp3.getSubpathPoints()
mPointList == p1            # returns True
# get point list for offset subpath
mPointList = mp3.getSubpathPoints('OFFSET_PATH')
# create justified point list from master path point list
p2 = MultiPath.genJustifyPathPoints(p1, justify=EAST, sep=1.0)
# illustrate use of different movement methods
mp3.mirrorX()
mp3.mirrorX()               # mp3 back to original position
mp3.rotate90()
mp3.rotate270()             # mp3 back to original position
mp3.moveBy(1,1)
```

RoutePath

Description: The RoutePath class provides the ability to make a connection between two different shapes within a design. These shapes can be either rectangle shapes or the shapes associated with a Pin or InstPin object. In general, this RoutePath class provides a connection between two different RouteTarget objects (see section on "RouteTarget"). Note that any Rect, RectRef, Pin or InstPin object will automatically be converted to the appropriate RouteTarget object. Thus, when RouteTarget objects are specified as parameters for any of these RoutePath methods, then a Rect, RectRef, Pin or InstPin object can be used instead, as this object will automatically be converted to the corresponding RouteTarget object. Alternatively, the RouteTarget object can be explictly constructed and used. The benefit of this latter approach is that it allows the designer to use the different methods for the RouteTarget class, so that additional information about the generated routing can be specified. For example, the designer can use the RouteTarget object to specify the point inside the shape which should be used for the RoutePath starting or ending point. By default, the center point for the bounding box of the shape is used for the RoutePath starting or ending point.

This RoutePath class provides the ability to generate different types of shapes to connect these RouteTarget objects. For example, a straight line (with an optional width) can be generated to make this connection. In addition, LShapes, ZShapes and CShapes can be generated as RoutePath objects between two different shapes in the design. A flight line can also be drawn between two different shapes, making use of the flightLine layer which is defined in the technology file for the current design. In the case of the straight line RoutePath object, there is an additional method provided to directly connect a RouteTarget shape to a Bar object. It should also be noted that a `Connect()` method is provided for this RoutePath class, which will generate the best type of RoutePath object to connect the two different shapes in the design. If a straight-line RoutePath can not be used to connect these two different shapes, then a ZShape or LShape will automatically be used. This is the preferred method to be used to create connections between shapes in different design objects, since it figures out the best RoutePath to be used, based upon the placement of the shapes for these different design objects. For example, this `Connect()` method can be used to connect together the corresponding source, gate and drain pins between two

transistors. Based upon the placement of these transistors (and their corresponding pins), the appropriate straight line, ZShape or LShape Route objects will be generated for each pin-to-pin connection.

The specification of access points by RouteTarget objects is used as a guideline for the RoutePath methods to generate starting and ending access points. When the routing shapes are generated by the RoutePath methods, then the passed RouteTarget objects will be updated with the exact values for these access points. In addition, when the generated routing shapes are on the same layer as the Route Targets, then these routing shapes will abut with these Route Target shapes. In the case that these generated routing shapes are on a different layer, then a Contact will be generated, so that these routing shapes may overlap with the RouteTarget shapes. If the designer would like to modify the routing shapes which are returned by the RoutePath methods, then the **flatten**() or **ungroup**() methods on the underlying base classes can be used to directly access these shapes.

Note that this RoutePath object will generate Contact objects when they are needed to connect different layers in the design. By default, these Contact objects will automatically be generated when the RoutePath connection is generated. However, the designer can choose to not generate these contacts, and can create any Contact objects themselves. This option would typically be used when the designer wants to use their own custom contacts, in place of the Contact objects which are provided by the Python API. When a RouteTarget contains multiple shapes, then the shape with the matching layer which will generate the shortest RoutePath distance will be chosen by the RoutePath.

Creation:

Note that almost all methods for this RoutePath class are static methods. Thus, there is no constructor method for this RoutePath class, and methods are invoked by directly invoking the class object name. That is, this RoutePath object will already exist in the Python environment and methods can be directly called as needed.

Methods:

Connect(RouteTarget *fromTarg*, RouteTarget *toTarg*, Layer *layer* = None, Coord *width* = 0, bool *genContact*=True, string *name*=) – this method first attempts to generate a straight-line route to connect the two RouteTarget objects. Note that these RouteTarget parameters can also be Rect, RectRef, Pin or InstPin objects; these objects will automatically be converted into the corresponding RouteTarget object.

If a straight-line can not be generated (eg: due to lack of intersection, or otherwise not being connectable by a straight line), then a ZShape will be generated as the route path to connect these two RouteTarget objects. If this ZShape can not be successfully generated, then an LShape will be generated. If the *width* parameter is not specified, then the minimum width value determined from the appropriate DRC rules in the technology file will be used. If the *width* parameter is specified, then it will be used, without checking that it meets any DRC minimum width rules. If the *name* parameter is not specified,

then a unique name string will be generated to be used as the name for this Route. This method will return the successful straight line RoutePath, ZShape RoutePath or LShape RoutePath, or will raise an exception if no RoutePath object can be generated.

Note that this method is defined as a static method, so that it can be called without directly constructing this RoutePath object; for example, `RoutePath.Connect()`.

CShape(RouteTarget *fromTarg*, RouteTarget *toTarg*, Layer *layer*=None, Point *position* = None, Direction *dir*=EAST, Coord *width* = 0, bool *genContact*=True, string *name*=) – generates an C-shaped RoutePath which connects these two RouteTarget objects. Note that these RouteTarget parameters can also be Rect, RectRef, Pin or InstPin objects; these objects will automatically be converted into the corresponding RouteTarget object. This CShape RoutePath is very useful when connecting multiple fingers of interdigitated devices in a design layout.

This CShape route path is generated on the given *layer* parameter, and has the width specified by the Coord *width* parameter. If the *layer* parameter is not specified, then the preferred routing layer for the RouteTarget objects will be used. Note that if the *genContact* parameter is True, then a contact will automatically be generated when necessary to connect two layers. If the *width* parameter is not specified, then the minimum width value determined from the appropriate DRC rules in the technology file will be used. Note that if this *width* parameter is specified, then it will be used, without checking that it meets any DRC minimum width rules. If the *dir* parameter is specified, it will be used to determine the direction of this CShape from the *fromTarg* RouteTarget to the *toTarg* RouteTarget. If the *position* parameter is specified, then the center segment of this CShape will align to this *position* value; otherwise, the center of this CShape will align next to the farthest edge of the two RouteTarget parameters. If the *name* parameter is not specified, then a unique name string will be generated to be used as the name for this Route. This method will return the CShape Route, or will raise an exception if the CShape route can not be generated. Note that the *width* parameter must be non-negative or an exception is raised.

Note that this method is defined as a static method, so that it can be called without directly constructing this RoutePath object; for example, `RoutePath.CShape()`.

FlightLine(RouteTarget *fromTarg*, RouteTarget *toTarg*, Layer *layer*=None, string *name* =) – generates the flight line between these two RouteTarget objects. Note that these RouteTarget parameters can also be Rect, RectRef, Pin or InstPin objects; these objects will automatically be converted into the corresponding RouteTarget object.

This flight line is drawn as a line between the two different RouteTarget objects. If access points have been defined by the user for either of these RouteTargets, then they will be used; otherwise the center point of the RouteTarget route box will be used as the starting or stopping point for this flight line. If the optional *layer* parameter is specified, then this layer is used as the layer on which to generate this flight line; otherwise, the flightLine layer as defined in the current technology file for this design will be used. If the *name* parameter is not specified, then a unique name string will be generated to be used as the

name for this RoutePath. This method will return the flight line RoutePath object, or will raise an exception if the flight line route can not be generated.

Note that this method is defined as a static method, so that it can be called without directly constructing this RoutePath object; for example, `RoutePath.FlightLine()`.

LShape(RouteTarget *fromTarg*, RouteTarget *toTarg*, Layer *layer*=None, Direction *dir* = EAST_WEST, Coord *width*=0, bool *genContact*=True, string *name*=) – generates an L-shaped RoutePath which connects these two RouteTarget objects. Note that these RouteTarget parameters can also be Rect, RectRef, Pin or InstPin objects; these objects will automatically be converted into the corresponding RouteTarget object. This LShape route path is generated on the given *layer* parameter, and has the width specified by the Coord *width* parameter. If the *layer* parameter is not specified, then the preferred routing layer for the RouteTarget objects will be used. Note that if the *genContact* parameter is True, then a contact will automatically be generated when necessary to connect two layers. If the *width* parameter is not specified, then the minimum width value determined from the appropriate DRC rules in the technology file will be used. Note that if this *width* parameter is specified, then it will be used, without checking that it meets any DRC minimum width rules. If the *dir* parameter is specified, it will be used (and possibly extended) to determine the direction of this LShape from the *fromTarg* RouteTarget to the *toTarg* RouteTarget. If the *name* parameter is not specified, then a unique name string will be generated to be used as the name for this Route object. This method will return the LShape Route, or will raise an exception if the LShape route can not be generated.

Note that the *width* parameter must be non-negative or an exception is raised.

Note that this method is defined as a static method, so that it can be called without directly constructing this RoutePath object; for example, `RoutePath.LShape()`.

StraightLine(RouteTarget *fromTarg*, RouteTarget *toTarg*, Layer *layer*=None, Coord *width* = 0, bool *genContact*=True, string *name* =) – generates a straight line route from one RouteTarget object to another RouteTarget object. Note that these RouteTarget parameters can also be Rect, RectRef, Pin or InstPin objects; these objects will automatically be converted into the corresponding RouteTarget object. This straight line route path is generated on the given *layer* parameter, and has the width specified by the Coord *width* parameter. If the *layer* parameter is not specified, then the preferred routing layer for the RouteTarget objects will be used. Note that if the *genContact* parameter is True, then a contact will automatically be generated when necessary to connect two layers. If the *width* parameter is not specified, then the minimum width value determined from the appropriate DRC rules in the technology file will be used. Note that if this *width* parameter is specified, then it will be used, without checking that it meets any DRC minimum width rules. If the *name* parameter is not specified, then a unique name string will be generated to be used as the name for this Route object. This method will return the straight line RoutePath object, or will raise an exception if the straight line route can not be generated. Note that the *width* parameter must be non-negative or an exception is raised.

Feedback

Note that this method is defined as a static method, so that it can be called without directly constructing this RoutePath object; for example, `RoutePath.StraightLine()`.

StraightLineToBar(RouteTarget *fromTarg*, Bar *bar*, Layer *layer*=None, Coord *width* = 0, bool *genContact*=True, string *name* =) – generates a straight line route from a RouteTarget object to a Bar object. Note that this RouteTarget parameter can also be Rect, RectRef, Pin or InstPin object; these objects will automatically be converted into the corresponding RouteTarget object. This straight line route path is generated on the given *layer* parameter, and has the width specified by the Coord *width* parameter. If the *layer* parameter is not specified, then the preferred routing layer for the RouteTarget objects will be used. Note that if the *genContact* parameter is True, then a contact will automatically be generated when necessary to connect two layers. If the *width* parameter is not specified, then the minimum width value determined from the appropriate DRC rules in the technology file will be used. Note that if this *width* parameter is specified, then it will be used, without checking that it meets any DRC minimum width rules. If the *name* parameter is not specified, then a unique name string will be generated to be used as the name for this Route object. This method will return the straight line RoutePath object, or will raise an exception if the straight line route can not be generated.

Note that the *width* parameter must be non-negative, or an exception is raised.

Note that this method is defined as a static method, so that it can be called without directly constructing this RoutePath object; such as `RoutePath.StraightLineToBar()`.

ZShape(RouteTarget *fromTarg*, RouteTarget *toTarg*, Layer *layer*=None, Point *position* = None, Direction *dir*=EAST_WEST, Coord *width* = 0, bool *genContact*=True, string *name*=) – generates an Z-shaped RoutePath which connects these two RouteTarget objects. Note that these RouteTarget parameters can also be Rect, RectRef, Pin or InstPin objects; these objects will automatically be converted into the corresponding RouteTarget object. This ZShape route path is generated on the given *layer* parameter, and has the width specified by the Coord *width* parameter. If the *layer* parameter is not specified, then the preferred routing layer for the RouteTarget objects will be used. Note that if the *genContact* parameter is True, then a contact will automatically be generated when necessary to connect two layers. If the *width* parameter is not specified, then the minimum width value determined from the appropriate DRC rules in the technology file will be used. Note that if this *width* parameter is specified, then it will be used, without checking that it meets any DRC minimum width rules. If the *dir* parameter is specified, it will be used (and possibly extended) to determine the direction of this ZShape from the *fromTarg* RouteTarget to the *toTarg* RouteTarget. If the *position* parameter is specified, then the center segment of this ZShape will align to this *position* value; otherwise, the center segment will align to the midpoint between the two RouteTarget parameters. If the *name* parameter is not specified, then a unique name string will be generated to be used as the name for this RoutePath. This method will return the ZShape RoutePath, or will raise an exception if the ZShape route can not be generated.

Note that the *width* parameter must be non-negative or an exception is raised.

Feedback

Note that this method is defined as a static method, so that it can be called without directly constructing this RoutePath object; for example, `RoutePath.ZShape()`.

checkLayerWidth(Layer *layer*, Coord *width*) – this method checks the minimum width DRC rule for the current technology file, and returns True, if the *width* parameter value is greater than or equal to this minimum width rule value, and returns False otherwise.

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this Route object.

destroy() – destroys all of the components of this RoutePath object (all route rectangles and Contact objects), as well as deleting this RoutePath object from the DloGen owner. Note that the underlying OpenAccess object will also be destroyed and removed from the database.

findAdjacentInterconnectLayer(LayerList *layers*, Layer *bestLayer*, bool *prefAbove*=True) – returns the closest interconnect layer to the *bestLayer* layer from the specified *layers* list of layers. The *prefAbove* parameter is only used when there is no exact match of the *bestLayer* layer in the specified list of layers. In this case, the closest layer in the list is determined by the layer number as defined in the technology file. If *prefAbove* is True, then the closest layer with a higher layer number will be returned; otherwise, the closest layer with a lower layer number will be returned

Note that this method is defined as a static method, so that it can be called without directly constructing this RoutePath object; for example, “RoutePath. findAdjacentInterconnectLayer ()”.

getLayer() – returns layer which is being used as routing layer for this RoutePath object.

Inherited Methods: This RoutePath class inherits all methods defined for the Grouping class, as well as the PhysicalComponent class.

Examples:

```
# assume pin1 and pin2 are Pin objects for existing design
# As an example, just use Pin objects from Contact objects
contact1 = Contact(Layer('metal1'), Layer('metal3'), 'A')
contact2 = Contact(Layer('metal1'), Layer('metal3'), 'B')
pin1 = contact1.findInstPin()
pin2 = contact2.findInstPin()

# create straight-line route between pin1 and pin2
contact1.place(WEST, contact2, 10.0)
RoutePath.StraightLine(pin1, pin2, Layer('metal1'))
# can also create straight-line route, specifying width
pinWidth = pin1.getBBox(Layer('metal1')).getHeight()
RoutePath.StraightLine(pin1, pin2, Layer('metal1'), pinWidth)
# draw flight line between pin1 and pin2, using flightLine Layer
RoutePath.FlightLine(pin1, pin2)
RoutePath.FlightLine(pin1, pin2, Layer('flightLine'))
# now create CShape, LShape, ZShape routes between pin1 and pin2
```

Feedback

```
RoutePath.CShape(pin1, pin2, Layer('metal1'), dir=NORTH)
RoutePath.CShape(pin1, pin2, Layer('metal1'), dir=SOUTH)
contact2.moveBy(dx=0, dy=5.0)
RoutePath.LShape(pin1, pin2, Layer('metal1'))
RoutePath.ZShape(pin1, pin2, Layer('metal1'))
# can also generate ZShape between pin1 and pin2 using "Connect"
RoutePath.Connect(pin1, pin2, Layer('metal1'))
# create Bar object to route pin to bar
bar1 = Bar(Layer('metal1'), EAST_WEST, 'C')
bar2 = Bar(Layer('metal1'), NORTH_SOUTH, 'C')
# connect pin1 and pin2 to Bar object using straight-line route
RoutePath.StraightLineToBar(pin1, bar2, Layer('metal1'))
RoutePath.StraightLineToBar(pin2, bar1, Layer('metal1'))
```

Boundary Classes

The Boundary class is the base class for the PRBoundary class. These boundary classes are used to define a boundary which encloses a design block, which is typically used for Place and Route applications. This boundary is used to constrain the area in which the contents of the design block may be placed. This Boundary class is derived from the PhysicalComponent class, and is the base class for the PRBoundary class.

Boundary

Description: The Boundary class is derived from the base PhysicalComponent class, and provides a generic boundary which can be used to define an enclosed area. This boundary includes a polygonal outline, which is defined by a set of points, which in turn define a set of polygon edges. Each of these edges can have an associated name which is specified by the PyCell developer.

Creation:

Since this Boundary class is an abstract base class, it does not have a creation method. Instead, the creation method for the PRBoundary class should be used. The PRBoundary derived class will inherit all methods which are defined for this base Boundary class.

Methods:

getBBox(ShapeFilter *filter*=ShapeFilter()) – returns the bounding box for this Boundary. Note that this Boundary does not have any layers associated with it, so that any layers specified in the *filter* ShapeFilter object will necessarily be ignored.

getEdgeNames( ) – returns the list of names for each edge in the polygonal boundary for this Boundary object.

getNumEdges( ) – returns the total number of edges for the polygonal boundary for this Boundary object.

Feedback

getPoints( ) – returns the uniform list of points which define the polygonal boundary for this Boundary object. Note that the ordering of the points in the list of points which is returned is not guaranteed (by OpenAccesss) to be the same as the ordering which was used for the list of points when this Boundary object was created.

setEdges(PointList *points*, ulist[string] *edgeNames*) – Uses the specified list of points and edge names to set the edges and edge names for this Boundary object. Note that it is assumed that the number of edge names is the same as the number of points.

Inherited Methods: This Boundary class inherits all methods which are defined for the PhysicalComponent class.

PRBoundary

Description: The PRBoundary class is derived from the base Boundary class, and provides a boundary for a block which is typically used for Place-and-Route operations. Note that there can be only be a single PRBoundary object within the current DloGen design object (as this is an OpenAccess restriction).

Creation:

PRBoundary(PointList *points*, ulist[string] *edgeNames*) – creates a PRBoundary object, using the specified list of points to define the polygonal boundary. In addition, the list of edge names is used to name each of the edges defined by the polygonal boundary. Note that if a PRBoundary object already exists in the current DloGen design object, then an exception is raised.

Methods:

find() – returns the single PRBoundary object which exists in the current DloGen design object. If there is no PRBoundary object defined in the current design, then None will be returned.

Note that this method is a static method, so that it can be called without explicitly creating a PRBoundary object. For example, it can be directly called using the syntax `PRBoundary.find()`.

Inherited Methods: This PRBoundary class inherits all methods which are defined for the Boundary class.

Examples:

```
# create design using MOSFET transistor from IPL Library
P = ParamArray()
inst1 = Instance('IPL_cni130/Nmos', p)
# now use bounding box of design to create PR Boundary
pts = PointList([self.getBBox().lowerLeft(),
                 self.getBBox().upperLeft(),
                 self.getBBox().upperRight(),
```

```
                self.getBBox().lowerRight()])
edges = ['A', 'B', 'C', 'D']
prb = PRBoundary(pts, edges)
# use methods to obtain information about PRBoundary object
prb.getBBox()           # returns bounding box
prb.getNumEdges()       # returns 4
prb.getEdgeNames()      # returns ['A', 'B', 'C', 'D']
prb.getPoints()         # returns list of points
# now change assignment of edges and edge names
prb.setEdges(pts, ['one', 'two', 'three', 'four'])
prb.getEdgeNames()      # returns ['one', 'two', 'three', 'four']
PRBoundary.find()       # returns current PRBoundary object
# Can not create additional PRBoundary object,
# as only single PRBoundary allowed per design.
# Check that exception is raised.
prb2 = PRBoundary(pts, ['alpha', 'beta', 'gamma', 'delta'])
```

Via Classes

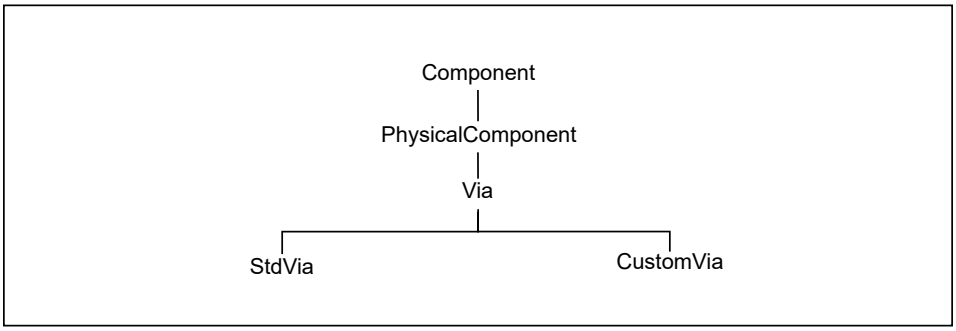
The Via class is the base class for the StdVia and the CustomVia classes. These via classes are used to represent a connection between two different layers. The StdVia standard via class provides a standard via, which is an instance of a standard via definition specified in the associated technology file. This standard via has a fixed number of different parameters, whose values can be specified to create the standard via. The CustomVia custom via class provides a custom via, which is an instance of a custom via definition specified in the technology file. This custom via has a variable number of different custom parameters, whose values are specified to create the custom via.

The Via base class provides basic functionality which is used by both the StdVia standard via class and the CustomVia custom via class.

Note that the use of these StdVia and CustomVia classes has an important pre-requisite. Namely, the appropriate standard via definition (oaStdViaDef) or custom via definition (oaCustomViaDef) must already be present in the OpenAccess technology file which is being used by the PyCell. This standard or custom via definition needs to be created, using available OpenAccess development tools. If the PyCell source code attempts to use a standard or custom via definition which does not already exist in the technology file, then an exception is raised.

Feedback

These different via classes can be represented using the following class inheritance diagram:



Via

Description: The Via class is derived from the base PhysicalComponent class, and provides an electrical connection between two adjacent conducting layers on a chip. This via is associated with three different layers: 1) the top wiring layer (layer2), 2) the middle connecting cut layer, and 3) the bottom wiring layer (layer1). This Via class is a base class, from which is derived the StdVia and CustomVia classes to create standard and custom vias which are defined in a standard manner in the associated technology file.

Creation:

Since this Via class is an abstract base class, it does not have a creation method. Instead, the creation methods for the StdVia class or the CustomVia class should be used. These StdVia and CustomVia derived classes will inherit all methods which are defined for this base Via class.

Methods:

getBBox(ShapeFilter *filter*=ShapeFilter()) – returns the bounding box for this Via. Any layers specified in the ShapeFilter *filter* parameter are used to perform this bounding box calculation. The merged bounding boxes for all specified layers will be returned.

getColorMask( ) – returns coloring information (string *anchorString*, int *layer1ColorShift*, int *layer2ColorShift*, int *cutLayerColorShift*) for this Via for multi-patterning. String *anchorString* can be either `ANCHORED` or `UNANCHORED`. Integers *layer1ColorShift*, *layer2ColorShift*, and *cutLayerColorShift* are 2-bit values representing color shifts from corresponding shapes in the master design. Values 0, 1, 2, 4, 5, 6 are for no shift, shift by 1, shift by 2, shift by 4, shift by 5, and shift by 6 respectively. Special value 3 means no coloring for the appropriate layer (even if the master design shape has an MP color attribute). By default, this method returns (`UNANCHORED`, 0, 0, 0) for a Via which does not have an MP color attribute.

getLayer1( ) – returns the bottom layer for this Via

getLayer2( ) – returns the top layer for this Via

getMaster(bool *buildPyObj*=False) – returns the master design for this Via; when via parameters are changed, then this master design is updated. If the *buildPyObj* parameter is True, then PhysicalComponent, Instance, Term, Pin, and Grouping objects are created on Dlo from the master design.

getName( ) – Note that Via objects are unnamed objects in the Python API; thus, this method always returns None. This method is only provided to override the **getName**() method which is provided by the base PhysicalComponent class.

getNet( ) – returns any net which is assigned to this Via

getOrientation( ) – returns the orientation for this Via

getOrigin( ) – returns the origin point for this Via

getTransform( ) – returns any transform which has been defined for this Via

getViaDefName( ) – returns the name of the via definition associated with this Via; this is the via definition which is defined in the associated OpenAccess technology file.

setColorMask(string *anchorString*, int *layer1ColorShift*, int *layer2ColorShift*, int *cutLayerColorShift*) – sets coloring information for this Via for multi-patterning. String *anchorString* can be either `ANCHORED` or `UNANCHORED`. Integers *layer1ColorShift*, *layer2ColorShift*, and *cutLayerColorShift* are 2-bit values representing color shifts from corresponding shapes in the master design. Values 0, 1, 2, 4, 5, 6 are for no shift, shift by 1, shift by 2, shift by 4, shift by 5, and shift by 6, respectively. Special value 3 means no coloring for the appropriate layer (even if the master design shape has an MP color attribute).

setName(string *name*) – Note that Via objects are unnamed objects in the Python API; thus, this method expects the empty string for the value of the *name* parameter, otherwise no operation will be performed. This method is only provided to override the **setName**() method which is provided by the base PhysicalComponent class.

setNet(Net *net*) – assigns the specified *net* to this Via. Note that if the *net* parameter is set to None, then this Via is removed from any assigned net.

setOrientation(Orientation *orient*) – sets an orientation for this Via

setOrigin(Point *origin*) – sets origin point for this Via

setTransform(Transform *trans*) – sets transform to be applied to this Via

---

StdVia

Description: The StdVia class is derived from the base Via class, and represents a standard via. This standard via should be defined in the associated technology file.

This standard via has a fixed number of pre-defined parameters which can be changed to modify the default standard via. These parameter values are set using the ViaParam class (as described in the `Parameter Classes` section).

Creation:

StdVia(string *viaDefName*, ViaParam *params* = None, string *node*= , Transform *trans*=None) – creates a standard via object, where the *viaDefName* parameter specifies the name of the standard via definition which should be contained in the associated OpenAccess technology file. The *params* parameter is a ViaParam object which contains all of the parameter values which should be set for this particular instance of the standard via. Any parameters which are not specified in this ViaParam object will automatically use their default values. If this *params* parameter is not specified, then the default values for all via parameters will be used. The optional *node* parameter is the name of the net which should be assigned to this standard via. The optional *trans* parameter specifies a transform which should be applied when this standard via is instantiated in the design.

Methods:

getCutLayer() – returns the cut layer for this standard via

getImplantLayer1( ) – returns the implant layer for the bottom layer (layer1) for this standard via. If there is no implant layer for the bottom layer, then None is returned.

getImplantLayer2( ) – returns the implant layer for the top layer (layer2) for this standard via. If there is no implant layer for the top layer, then None is returned.

getParams( ) – returns the ViaParam object which was used to set the via parameters for this standard via.

setParams(ViaParam *params*) – sets the via parameters for this standard via, using the value of the *params* parameter. Note that only the non-default parameter values will be updated by this method.

Inherited Methods: This StdVia class inherits all methods which are defined for the Via class.

Examples:

```
# assume that standard via "contact" exists in technology file
# first create standard via using all default parameter values
via1 = StdVia("contact")
# now modify some of these default parameter values,
# and then use them to create another standard via.
defaultViaParams = via1.getParams()
vp2 = ViaParam(params = defaultViaParams,
               cutSize = 0.10,
               cutSpace = 0.12)
via2 = StdVia("contact", params = vp2)
# can also use these parameter values to change
# via parameters for an existing standard via
```

```
via1.setParams(vp2)
via1.getCutLayer()        # returns cut layer
via1.getImplantLayer1()   # returns implant layer for bottom layer
via1.getImplantLayer2()   # returns implant layer for top layer
```

CustomVia

Description: The CustomVia class is derived from the base Via class, and represents a custom via. This custom via is defined in the associated technology file. This custom via has a number of parameters which can be changed to modify the custom via which is generated. These parameter values are set using the ParamArray class (as described in the “Parameter Classes” section).

Creation:

CustomVia(string *viaDefName*, ParamArray *params* = None, string *node*= , Transform *trans*=None, bool *checkParams*=False) – creates a custom via object, where the *viaDefName* parameter specifies the name of the custom via definition which should be contained in the associated OpenAccess technology file. The *params* parameter is a ParamArray object which contains all of the parameter values which should be set for this particular instance of the custom via. Any parameters which are not specified in this ParamArray object will automatically use their default values. If this *params* parameter is not specified, then the default values for all custom via parameters will be used. The optional *node* parameter is the name of the net which should be assigned to this custom via. The optional *trans* parameter specifies a transform which should be applied when this custom via is instantiated in the design. The optional *checkParams* parameter allows you to check whether all parameters provided in the *params* are actually the custom via parameters. By default, non-custom via parameters are converted to properties and stored on the created CustomVia object. If *checkParams*=True, an exception is thrown when a non-custom via parameter is provided.

Methods:

getParams(ParamArray *params*=None, bool *all*=True) – returns the **ParamArray** object which provides the explicit parameters and values which were used when this custom via was created. If the Boolean *all* parameter is True, then all parameters (both explicit and default) will be returned. If the *params* parameter is specified, then it will be set to the same values as are returned by this method.

setParams(ParamArray *params*, bool *checkParams*=False) – sets the parameter values for this custom via, using the value of the *params* parameter. The optional *checkParams* parameter allows you to check whether all parameters provided in the *params* are actually the custom via parameters. By default, non-custom via parameters are converted to properties and stored on this CustomVia object. Note that already existing properties with the same name will be overridden. If *checkParams*=True, an exception is thrown when a non-custom via parameter is provided.

Feedback

Inherited Methods: This CustomVia class inherits all methods which are defined for the Via class.

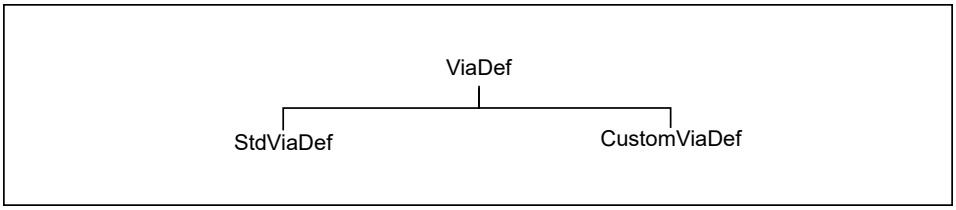
Examples:

```
# assume custom via "customContact" exists in technology file
# first create custom via using all default parameter values
via1 = CustomVia("customContact")
# now modify some of these default parameter values,
# and then use them to create another standard via.
vp2 = via1.getParams()
minCutSize = self.tech.getPhysicalRule('minWidth',
                                       Layer('contact'))
minCutSpace = self.tech.getPhysicalRule('minSpacing',
                                        Layer('contact'))
vp2['cutSize'] = minCutSize
vp2['cutSpace'] = minCutSpace
via2 = CustomVia("customContact", params = vp2)
# can also use these parameter values to change
# via parameters for an existing custom via
via1.setParams(vp2)
```

ViaDef Classes

The ViaDef class is the base class for the StdViaDef and the CustomViaDef classes. These classes represent via definitions present in the technology library which are used for creation of StdVia and CustomVia objects. Objects of these classes are never created directly in the API but returned by getStdViaDef and getCustomViaDef methods of Tech object. The StdViaDef standard via definition class represents a standard via definition existing in the technology library. The CustomViaDef custom via definition class represents custom via definition existing in the technology library. The ViaDef base class provides basic functionality which is used by both the StdViaDef standard via definition class and the CustomViaDef custom via definition class.

These different via classes can be represented using the following class inheritance diagram:



Feedback

ViaDef

Description: The ViaDef class represents via definitions present in the technology library which are used for creation of StdVia and CustomVia objects. This ViaDef class is a base class, from which is derived the StdViaDef and CustomViaDef classes of standard and custom via definitions which are defined in a standard manner in the associated technology file.

Creation:

Since this ViaDef class is an abstract base class, it does not have a creation method. The StdViaDef and CustomViaDef derived classes inherit all methods which are defined for this base ViaDef class.

Methods:

getName( ) – returns the name of this via definition

getLayer1( ) – returns the first layer associated with this via definition

getLayer2( ) – returns the second layer associated with this via definition

StdViaDef

Description: The StdViaDef class is derived from the base ViaDef class, and represents a standard via definition existing in the technology library.

Creation:

Objects of this class are never created directly in the API but returned by the getStdViaDef method of Tech object.

Methods:

getImplantLayer1( ) – returns the first implant layer associated with this standard via definition. If there is no implant layer, then None is returned.

getImplantLayer2( ) – returns the second implant layer associated with this standard via definition. If there is no implant layer, then None is returned.

getParams( ) – returns the ViaParam object representing default parameters associated with this standard via definition.

Inherited Methods: This StdViaDef class inherits all methods which are defined for the ViaDef class.

Feedback

CustomViaDef

Description: The CustomViaDef class is derived from the base ViaDef class, and represents a custom via definition existing in the technology library.

Creation:

Objects of this class are never created directly in the API but returned by the getCustomViaDef method of Tech object.

Methods:

getLibName() – returns the library name of the master design referenced by this custom via definition.

getCellName() – returns the cell name of the master design referenced by this custom via definition.

getViewName() – returns the view name of the master design referenced by this custom via definition.

getParams() – returns the **ParamArray** object representing default parameters associated with this custom via definition.

Inherited Methods: This CustomViaDef class inherits all methods which are defined for the ViaDef class.

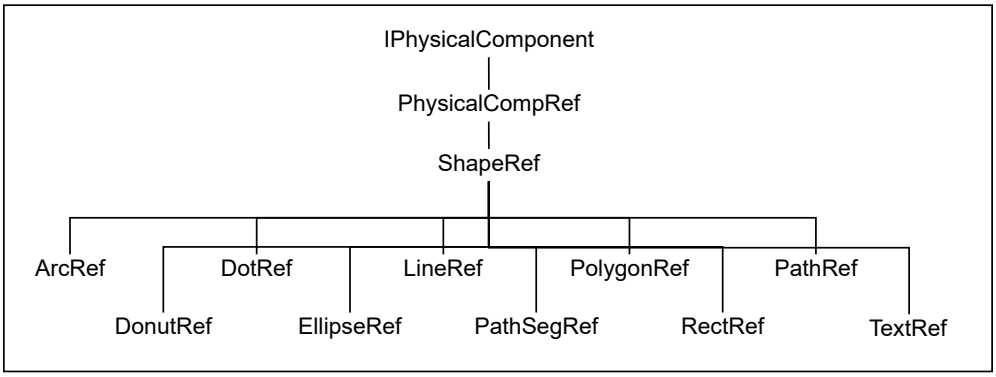
3

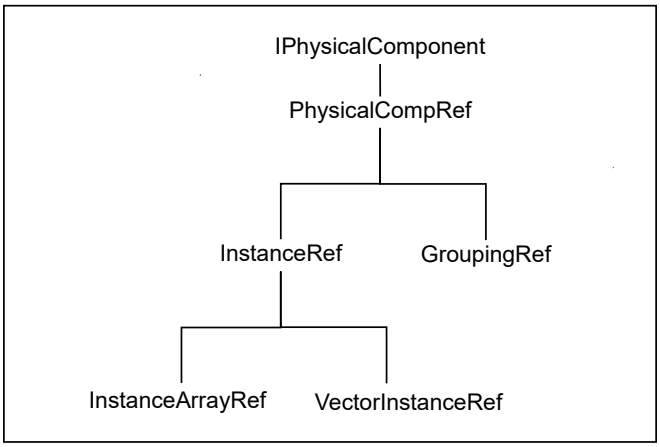
PHYSICAL COMPONENT REFERENCE CLASSES

There are several reference classes based upon the PhysicalComponent class. These Hierarchical Object Reference classes are used to reference layout objects in the lower-level hierarchy in a hierarchical layout design. For example, the poly gate rectangle of a transistor instance is a shape in the lower-level hierarchy of a design which contains an instance of this transistor. It is important to be able to access information concerning the location of this lower-level gate rectangle, in order to more easily perform geometric operations on the layout. For example, these reference classes allow the designer to easily align this poly gate rectangle with another physical component in the design layout.

A large number of methods are provided for these reference classes, which make it much easier to perform various types of geometric operations, by using these lower-level physical components as reference points.

These different physical component reference classes can be represented using the following class inheritance diagrams. Note that the PhysicalCompRef class is based upon the IPhysicalComponent class, which is an interface class representing either an actual physical component or a reference to a physical component:





This class inheritance hierarchy allows the designer to access information about Shapes, Instances, members of InstanceArray objects and Groupings in any lower-level hierarchy for the design layout. The designer can simply refer to these lower-level objects in the layout hierarchy by name, when using the different methods which have been provided for these physical component reference classes.

In order to conveniently refer to reference components in a design, a specific naming convention is followed. Any such reference object will have a name of the form `Level1_InstanceName/…/LevelN_InstanceName/PhysicalCompName`. That is, the full hierarchical name consists of the name of each enclosing instance, separated by slashes, and ending with the name of the physical component. For example, the hierarchical name `INV1/polyGate` can be used to refer to the poly gate rectangle shape contained within an instance of an inverter named `INV1`. Unlike an Instance or Shape object which actually generates real geometry in the design, these reference components are virtual objects, and so do not generate any layout geometries. Note that as the containing higher-level instances are changed in the design, the properties and coordinates of these reference components are dynamically updated.

PhysicalCompRef

Description: The PhysicalCompRef class is the base class used for all Hierarchical Object Reference classes. This class allows any physical component contained within an instance in the current design to be used as a reference object. Note that these reference objects can only be created when they exist within an instance; otherwise, they would just be physical components in the design which could be directly accessed.

This PhysicalCompRef class provides a large number of methods to conveniently make use of these basic reference objects. The first group of methods are used to access the hierarchical names used for these reference objects. The second group of methods are geometric methods, which are used to manipulate and modify the physical component which is being referred to, so that the reference object is modified as desired. This

approach makes it much easier to perform various geometric operations on lower-level physical components in the design.

Creation:

Note that there is no creation method required for this PhysicalCompRef class. These reference objects are created and managed by the DloGen design object, which contains the current design. This is handled through the use of the **findCompRef**() method for the DloGen class. The first call to this **findCompRef**() method for a specific hierarchical physical component name will create the PhysicalCompRef object, while subsequent calls to this method using the same hierarchical name will refer to the existing reference object which was automatically created by the first call. Note that when a physical component is destroyed, then any associated reference component will also be automatically destroyed.

Methods:

getBBox(ShapeFilter *filter*=ShapeFilter()) – returns the bounding box for this PhysicalCompRef object. Note that this bounding box is always computed relative to the current DloGen design.

getLocationPoint(Location *loc*, Point *default*=None) – returns a point corresponding to the *loc* location which can be one of nine standard (pre-defined) locations, or a custom location previously created by calling setCustomLocation(). *default* is an optional parameter that is returned if the custom location does not exist. When *default* is None, an exception is thrown if the custom location does not exist.

getName() – returns the leaf-level name for this PhysicalCompRef within the current DloGen design. If this PhysicalCompRef refers to an un-named shape in the design, then None will be returned.

getParentPathName() – returns the full hierarchical path name to the parent instance which contains this PhysicalCompRef within the current DloGen design. This method can be used to get the parent name for an un-named shape in the design. For example, an un-named rectangle shape within the instance hierarchy `I1/I2` would have `I1/I2` as the parent path name.

getParentPathTransform() – returns the transform of the parent Instance, which is identified by the **getParentPathName**() method. Note that if this parent name does not exist, then the identity transform will be returned.

getPathName() – returns the full hierarchical name for this PhysicalCompRef within the current DloGen design. Note that if this PhysicalCompRef refers to an un-named shape in the design, then an internal identification string will be used for the leaf name which corresponds to the un-named shape.

getProps() – returns the set of properties which have been defined for the physical component which is being referenced by this PhysicalCompRef. This set of properties will be returned as a Python dictionary-like PropSet object.

Feedback

getTopInst() – returns the top-level instance in the current DloGen design which contains this PhysicalCompRef object. For example, for the PhysicalCompRef `I1/I2/rect1`, this method would return the I1 instance.

The following are the geometric methods for the PhysicalCompRef class, which allow geometric operations to be performed on the reference physical component, by simply performing the required geometric operations on the physical component which is referred to by the PhysicalCompRef object. Note that the reference component parameter *refComp* used in these methods can be either a physical component or a reference physical component; this is indicated by the interface class name `IPhysicalComponent`.

abut(Direction *dir*, IPhysicalComponent *refComp*, ShapeFilter *filter* = ShapeFilter(), bool *align*=True, ShapeFilter *refFilter*=None) – moves the physical component referred to by this PhysicalCompRef in the given direction *dir*, so that the bounding box for this PhysicalCompRef just touches the bounding box for the *refComp* physical component.

alignEdge(Direction *dir*, IPhysicalComponent *refComp*, Direction *refDir*=None, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *refFilter*=None, Coord *offset*=None) – moves the physical component referred to by this PhysicalCompRef, so that this PhysicalCompRef is edge aligned with the *refComp* physical component.

alignEdgeToCoord(Direction *dir*, Coord *coord*, ShapeFilter *filter*=ShapeFilter()) – moves the physical component referred to by this PhysicalCompRef, so that this PhysicalCompRef is edge aligned in the specified *dir* direction with the *coord* value.

alignEdgeToPoint(Direction *dir*, Point *point*, ShapeFilter *filter*=ShapeFilter()) – moves the physical component referred to by this PhysicalCompRef, so that this PhysicalCompRef is edge aligned in the *dir* direction with the specified *point* value.

alignLocation(Location *loc*, IPhysicalComponent *refComp*, Location *refLoc*=None, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *refFilter*=None, Point *offset*=None) – moves the physical component referred to by this PhysicalCompRef, so that this PhysicalCompRef is location aligned with the *refComp* physical component. Any of the nine different location points for a bounding box, or a custom location point set by setCustomLocation(), are valid values for the *loc* and *refLoc* parameters.

alignLocationToPoint(Location *loc*, Point *point*, ShapeFilter *filter*=ShapeFilter()) – moves the physical component referred to by this PhysicalCompRef, so that this PhysicalCompRef is location aligned with the specified *point* value. Any of the nine different location points for a bounding box, or a custom location point set by setCustomLocation(), are valid values for the *loc* parameter.

getSpacing(Direction *dir*, IPhysicalComponent *refComp*, ShapeFilter *filter*=ShapeFilter(), ShapeFilter *refFilter* = None) – calculates current distance between this PhysicalCompRef and the *refComp* physical component in the specified *dir* direction.

keepRelativePosition(IPhysicalComponent *comp*, bool *keep* = True) – keeps the alignment of physical components throughout PyCell evaluation. After calling this method

with *keep* parameter = True (default), the relative position of this PhysicalCompRef and *comp* parameter (which can be PhysicalComponent or PhysicalCompRef) is automatically maintained: when one of them moves, another moves accordingly. To terminate this relationship, call this method with *keep* parameter = False. Or to terminate all relationships between this object and all objects, call this method with *keep* parameter = False without *comp* parameter.

mirrorX(Coord *yCoord*=0) – mirrors the physical component referred to by this PhysicalCompRef , so that this PhysicalCompRef is mirrored about the X axis.

mirrorY(Coord *xCoord*=0) – mirrors the physical component referred to by this PhysicalCompRef, so that this PhysicalCompRef is mirrored about the Y axis.

moveBy(Coord *dx*, Coord *dy*) – moves the physical component referred to by this PhysicalCompRef, so that this PhysicalCompRef is moved by *dx* units in the x-direction and *dy* units in the y-direction.

moveTo(Coord *x*, Coord *y*, Location *loc*=Location.CENTER_CENTER, ShapeFilter *filter*=ShapeFilter()) – moves the physical component referred to by this PhysicalCompRef, so that this PhysicalCompRef is moved such that the handle of the bounding box for this PhysicalCompRef aligns with the *x* and *y* coordinate values. Any of the nine different location points for a bounding box, or a custom location point set by setCustomLocation(), are valid values for the *loc* parameter.

moveTowards(Direction *dir*, Coord *d*) – moves the physical component referred to by this PhysicalCompRef, so that this PhysicalCompRef is moved by *d* distance units in the direction provided by the *dir* Direction parameter.

place(Direction *dir*, IPhysicalComponent *refComp*, Coord *distance*, ShapeFilter *filter*=ShapeFilter(), bool *align*=True, ShapeFilter *refFilter*=None) – explicitly places the physical component referred to by this PhysicalCompRef, so that this PhysicalCompRef is spaced *distance* units from the *refComp* physical component.

overlaps(Box *box*) – returns True if this PhysicalCompRef overlaps the bounding box object passed as the *box* parameter, and returns False otherwise.

rotate90(Point *origin*=None) – rotates the physical component referred to by this PhysicalCompRef, so that this PhysicalCompRef is rotated by 90 degrees.

rotate180(Point *origin*=None) – rotates the physical component referred to by this PhysicalCompRef, so that this PhysicalCompRef is rotated by 180 degrees.

rotate270(Point *origin*=None) – rotates the physical component referred to by this PhysicalCompRef, so that this PhysicalCompRef is rotated by 270 degrees.

snap(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) – snaps the physical component referred to by this PhysicalCompRef, so that the lower-left vertex of the bounding box for this PhysicalCompRef lies on grid.

snapX(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) – snaps the physical component referred to by this PhysicalCompRef , so that the X coordinate of the lower-left vertex point of the bounding box for the PhysicalCompRef lies on grid.

snapY(Grid *grid*, SnapType *snapType*=None, ShapeFilter *filter*=ShapeFilter()) – snaps the physical component referred to by this PhysicalCompRef , so that the Y coordinate of the lower-left vertex point of the bounding box for the PhysicalCompRef lies on grid.

snapTowards(Grid *grid*, Direction *dir*, ShapeFilter *filter*=ShapeFilter()) – snaps the physical component referred to by this PhysicalCompRef, in the *dir* direction.

transform(Transform *trans*) – applies the transform *trans* passed as a parameter to the physical component referred to by this PhysicalCompRef.

Attributes:

In addition to these methods for the PhysicalCompRef class, there are also two attributes (or properties) defined for this PhysicalCompRef class, described as follows:

bbox – the bounding box for this physical component reference object

props – properties associated with the referenced physical component

Note that these two attributes will return the same values as are returned by the "**getBBox**()" and "**getProps**()" methods.

Examples:

```
# Assume hierarchical design with three levels of hierarchy;
# want to align lower-level instances "I1/I2/I3" and "I5/I6",
# which are not visible at top level of design hierarchy.
# create reference components, which can be aligned
rc1 = self.findCompRef('I1/I2/I3')
rc2 = self.findCompRef('I5/I6')
rc1.getName()              # returns leaf-level name "I3"
rc2.getName()              # returns leaf-level name "I6"
rc1.getPathName()          # returns "I1/I2/I3"
rc2.getPathName()          # returns "I5/I6"
rc1.getParentPathName()    # returns "I1/I2"
rc2.getParentPathName()    # returns "I5"
rc1.getTopInst()           # returns instance I1
rc2.getTopInst()           # returns instance I2
# align these reference components; note that top-level
# instances will be moved to align reference components.
rc2.alignEdge(WEST, rc1)
rc2.alignLocation(Location.CENTER_CENTER, rc1)
# can also perform other geometric operations on reference
# components; note that operation will be performed on the
# top-level instances to operate upon reference components
rc1.mirrorX()
rc1.mirrorY()
rc2.rotate90()
```

Feedback

```
rc2.rotate270()
rc1.moveBy(2.0, 2.0)
rc2.moveTowards(NORTH, 3.0)
rc2.getSpacing(NORTH, rc1)
# create instances of basic Python transistor example,
# to make use of physical component reference objects
# which are automatically generated for connectivity.
# Note that un-named shapes can be used as reference
# objects, as internal name will automatically be used.
p = ParamArray()
# define connectivity for this instance
conn = {'D':'d', 'G':'g', 'S':'s', 'B':'b'}
inst1 = Instance('cnPyBasicDlo/PyTransistor', p, conn, 'tran1')
inst2 = Instance('cnPyBasicDlo/PyTransistor', p, conn, 'tran2')
# get instance pins for Source and Drain
instPinS1 = inst1.findInstPin('S')
instPinS2 = inst2.findInstPin('S')
instPinD1 = inst1.findInstPin('D')
instPinD2 = inst2.findInstPin('D')
# obtain reference objects for these instance pins
rcS1 = instPinS1.getShapeRefs()[0]
rcS2 = instPinS2.getShapeRefs()[0]
rcD1 = instPinD1.getShapeRefs()[0]
rcD2 = instPinD2.getShapeRefs()[0]
# now align Source and Drain pin shapes in various ways;
# note that instances will be moved to align pin shapes.
rcS2.alignEdge(WEST, rcS1)
rcD2.alignEdge(WEST, rcD1)
rcS2.alignEdge(WEST, rcD1)
rcD2.alignEdge(WEST, rcS1)
rcS2.alignLocation(Location.CENTER_CENTER, rcS1)
rcD2.alignLocation(Location.CENTER_CENTER, rcD1)
```

GroupingRef

Description: The GroupingRef class is derived from the PhysicalCompRef class, and is used to reference a lower-level Grouping object, which is contained within the hierarchy of a hierarchical physical layout design. Note that this Grouping should be persistent; this is done by means of the **makePersistent**() method for the Grouping class. Also note that these GroupingRef objects can be nested, and contain internal lower-level GroupingRef objects.

Creation:

Note that there is no creation method used for this GroupingRef class. These reference objects are created and managed by the DloGen class which contains the layout design. These GroupingRef reference objects are created in the same manner as the base PhysicalCompRef class objects.

Feedback

Methods:

getCompRefs() – returns a uniform list of all of the reference physical components which are contained inside this GroupingRef object.

Inherited Methods: This GroupingRef class inherits all methods defined for the PhysicalCompRef class.

InstanceRef

Description: The InstanceRef class is derived from the PhysicalCompRef class, and is used to provide an Instance object which can be used as a reference component. Thus, a InstanceRef object refers to a lower-level Instance object, which is contained within the hierarchy of a hierarchical physical layout design.

Creation:

Note that there is no creation method used for this InstanceRef class. These reference objects are created and managed by the DloGen class which contains the layout design. These InstanceRef reference objects are created in the same manner as the base PhysicalCompRef class objects.

Methods:

getCompRefs() – returns a uniform list of all of the reference physical components which are contained inside this InstanceRef object. Note that if this InstanceRef contains a GroupingRef object, then the GroupingRef object will be returned; the lower-level PhysicalCompRef objects contained inside the GroupingRef will not be returned.

getMaster(bool *buildPyObj*=False) – returns the master design object for this InstanceRef. This master design object is a Dlo object. If the *buildPyObj* parameter is True, then PhysicalComponent, Instance, Term, Pin, and Grouping objects are created on Dlo from the master design. If this InstanceRef is not bound to a master design object, then an exception is raised.

getParams(ParamArray *params*=None, bool *all*=True) – returns the ParamArray which provides the explicit parameters and values. If the Boolean *all* parameter is True, then all parameters (both explicit and default) will be returned. If the *params* parameter is specified, then this ParamArray will be set to the same values as are returned by this method. If this InstanceRef is not bound to a master design object, then an exception is raised.

Inherited Methods: This InstanceRef class inherits all methods defined for the PhysicalCompRef class.

Feedback

InstanceArrayRef

Description: The InstanceArrayRef class is derived from the PhysicalCompRef class, and is used to provide an InstanceArray object which can be used as a reference component. Thus, an InstanceArrayRef object refers to a lower-level InstanceArray object, which is contained within the hierarchy of a hierarchical physical layout design.

Creation:

Note that there is no creation method used for this InstanceArrayRef class. These reference objects are created and managed by the DloGen class which contains the layout design. These InstanceArrayRef reference objects are created in the same manner as the base PhysicalCompRef class objects.

Methods:

getMemberRefs() – returns a uniform list of all of the reference physical components which are members of this InstanceArrayRef object.

Inherited Methods: This InstanceArrayRef class inherits all methods defined for the PhysicalCompRef class.

VectorInstanceRef

Description: The VectorInstanceRef class is derived from the InstanceRef class, and is used to provide a reference to a lower-level vector instance object which is contained within the hierarchy of a hierarchical physical layout design.

Creation:

Note that there is no creation method used for this VectorInstanceRef class. These reference objects are created and managed by the DloGen class which contains the layout design. These VectorInstanceRef reference objects are created in the same manner as the base PhysicalCompRef class objects.

Methods:

getCompRefs() – returns a uniform list of all of the reference physical components which are contained inside this VectorInstanceRef object. Note that the number of references to physical components in the resulting list depends on the number of bits in this vector instance. See also **VectorInstance.getCompRefs**().

Inherited Methods: This VectorInstanceRef class inherits all methods defined for the PhysicalCompRef class.

Feedback

ShapeRef

Description: The ShapeRef class provides reference objects for any hierarchical shape object in a layout design. In addition, this ShapeRef class can be used to easily access the shape information which is associated with connectivity. For purposes of connectivity, this ShapeRef class provides a read-only representation of shape information, including the stored information about shapes in the instance master or submaster. Since this information is read-only, there is no ability to change any of these connectivity associated shapes through this class. As a convenience, coordinate information is automatically translated into the context of the current design.

In addition to this ShapeRef class, there are also derived reference classes for all of the basic shape objects which are defined in the Python API, such as the Rect and Polygon shape classes. The RectRef and PolygonRef classes are derived from this base ShapeRef class to provide further shape information for specific referenced shape objects. For example, the PolygonRef class provides class methods to obtain information about the points which define the vertices of the polygon shape. Note that the built-in Python `isinstance` command can be used to check the type for a particular ShapeRef object.

Creation:

There is no creation method required for this ShapeRef class, as is also the case for the base PhysicalCompRef class. Note that all ShapeRef objects corresponding to named shapes can be searched for using the **findCompRef**() method for the DloGen design class. In addition, all ShapeRef objects can be found through iteration, using the **getCompRefs**() method for the Instance, InstanceRef and GroupingRef classes. Note that in addition, this ShapeRef class also provides read-only connectivity information for the shapes which are associated with an instance pin This shape connectivity information is automatically created when shapes are associated with an instance pin, and is maintained by the Santana system. This connectivity information can also be directly obtained using the “**getShapeRefs**()” method for the InstPin class. Note that there is no direct OpenAccess equivalent for this ShapeRef class object.

Methods:

getBBox(ShapeFilter *filter*=ShapeFilter()) – returns the bounding box for this ShapeRef object. Note that this bounding box is originally calculated in the coordinate system of the instance submaster, and is then converted to the coordinate system being used for the current DloGen design. This bounding box is calculated for all layers specified by the *filter* ShapeFilter parameter.

getColorMask() – returns coloring information (string *anchor*, string *color*) for this Shape for double and triple patterning. String *anchor* = `"ANCHORED" | "UNANCHORED`. String *color* = `"GRAY"` | “MASKCOLOR*n*”, where *n* is an integer from 1 to 127. Getting color for Dot, Text, and TextDisplay always returns (`"UNANCHORED"`, None).

getInst() – returns the associated instance for this ShapeRef object. Note that this method is being deprecated; the preferred approach is to use the **getTopInst**() method for the PhysicalCompRef class.

getInstPin() – returns the associated instance pin for this ShapeRef object.

getLayer() – returns the layer on which the referenced shape has been drawn. Note that this Layer object includes both layer and purpose information.

getName() – returns the name for the referenced shape; note that None is returned for an un-named shape.

getNetRef() – returns the associated net for this ShapeRef object.

getPinRef() – returns the associated pin for this ShapeRef object.

getTransform() – returns any Transform which has been defined for the associated instance for this ShapeRef object. Note that this method is being deprecated; the preferred approach is to use the **getParentPathTransform**() method for the PhysicalCompRef class.

Attributes:

In addition to these methods for the ShapeRef class, there are also three read-only attributes (or properties) defined for this ShapeRef class, described as follows:

bbox – the bounding box for this ShapeRef object

layer – the layer for this ShapeRef object

name – the name for this ShapeRef object

Note that these three attributes will return the same values as are returned by the "**getBBox**()", "**getLayer**()" and "**getName**()" methods.

Inherited Methods: This ShapeRef class inherits all methods from the PhysicalCompRef class.

Examples:

```
# create instance of basic Python transistor example,
# to obtain automatically generated instance pins,
# along with referenced shape for an instance pin.
p = ParamArray()
# define connectivity for this instance
conn = {'D':'d', 'G':'g', 'S':'s', 'B':'b'}
inst1 = Instance('cnPyBasicDlo/PyTransistor', p, conn, 'tran1')
# get instance pin
instPin1 = inst1.getInstPins()[0]
# now obtain referenced shape for this instance pin
sr1 = instPin1.getShapeRefs()[0]
sr1.getLayer()                  # returns Layer('metal4')
```

Feedback

```
sr1.getName()                    # returns None
sr1.getInst()                    # returns transistor instance
sr1.getInstPin()                 # returns instance pin
sr1.getTransform()               # returns transform
isinstance(sr1, RectRef)         # returns True
isinstance(sr1, PolygonRef)      # returns False
```

Derived ShapeRef Classes

There are also several classes which are derived from the base ShapeRef class. These classes include the ArcRef class, the DonutRef class, the DotRef class, the EllipseRef class, the LineRef class, the PathRefRef class, the PathSegRef class, the PolygonRef class, the RectRef class, the TextRef class, the TextDisplayRef class and the AttrDisplayRef class. That is, there is a reference class defined for each of the derived Shape classes defined in the Python API. Each of these derived reference classes would be used whenever the referenced shape is one of the basic Shape objects. Note that the Python `isinstance` command can be used to determine if a particular ShapeRef object is one of these derived ShapeRef class objects. With the exception of the PathRef and PolygonRef classes, these derived ShapeRef classes do not currently provide any additional methods. The PathRef class provides methods to allow the designer to access the points which define the path, the width, path style, and beginning and ending extension values. The PolygonRef class provides methods to allow the designer to access the vertex points which are used to define the polygon shape.

ArcRef

Description: The ArcRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is an arc shape. The Python `isinstance` command can be used to determine if a particular ShapeRef object is an ArcRef object.

Methods: There are no methods currently defined for this ArcRef class.

Inherited Methods: This ArcRef class inherits all methods from the ShapeRef class, as well as from the PhysicalCompRef class.

DonutRef

Description: The DonutRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a donut shape. The Python `isinstance` command can be used to determine if a particular ShapeRef object is a DonutRef object.

Methods: There are no methods currently defined for this DonutRef class.

Inherited Methods: This DonutRef class inherits all methods from the ShapeRef class, as well as from the PhysicalCompRef class.

Feedback

DotRef

Description: The DotRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a dot shape. The Python `isinstance` command can be used to determine if a particular ShapeRef object is a DotRef object.

Methods: There are no methods currently defined for this DotRef class.

Inherited Methods: This DotRef class inherits all methods from the ShapeRef class, as well as from the PhysicalCompRef class.

EllipseRef

Description: The EllipseRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is an ellipse shape. The Python `isinstance` command can be used to determine if a particular ShapeRef object is a EllipseRef object.

Methods: There are no methods currently defined for this EllipseRef class.

Inherited Methods: This EllipseRef class inherits all methods from the ShapeRef class, as well as from the PhysicalCompRef class.

LineRef

Description: The LineRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a line shape. The Python `isinstance` command can be used to determine if a particular ShapeRef object is a LineRef object.

Methods: There are no methods currently defined for this LineRef class.

Inherited Methods: This LineRef class inherits all methods from the ShapeRef class, as well as from the PhysicalCompRef class.

PathRef

Description: The PathRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a path shape. The Python `isinstance` command can be used to determine if a particular ShapeRef object is a PathRef object.

Methods:

getBeginExt() – returns the beginning extension value for the path which is being referenced by this PathRef object.

getBoundary(bool *usePathOrder*=False) – returns a list of the points which define the boundary for the path which is being referenced by this PathRef object. Note that the list

of boundary points returned by this method does not use a specific ordering. If a specific ordering is required, then *usePathOrder* should be set to True, and the boundary points will use the same ordering as the points which define the path.

getEndExt() – returns the ending extension value for the path which is being referenced by this PathRef object.

getNumPoints() – returns the number of points for the path which is being referenced by this PathRef object.

getPoints() – returns the list of points that define the path which is being referenced by this PathRef object. This list of points is returned as a PointList object.

getStyle() – returns the end point style for the path which is being referenced by this PathRef object. This end point style will be a string having one of the four PathStyle enumerated values TRUNCATE, EXTEND, ROUND or VARIABLE.

getWidth() – returns width of the path which is being referenced by this PathRef object.

isOrthogonal() – returns True, if all the points in the path which is being referenced by this PathRef object.are orthogonal, and returns False, otherwise.

Inherited Methods: This PathRef class inherits all methods from the ShapeRef class, as well as from the PhysicalCompRef class.

PathSegRef

Description: The PathSegRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a path segment shape. The Python `isinstance` command can be used to determine if a particular ShapeRef object is a PathSegRef object.

Methods:

getBoundary() – returns a list of the points which define the boundary for the path segment which is being referenced by this PathSegRef object. These points can be used to construct a polygon, which will represent the boundary for the path segment which is being referenced by this PathSegRef object.

getPoints() – returns the two points which define the path segment which is being referenced by this PathSegRef object. These two points are returned as a tuple, such as `(beginPoint, endPoint)`.

getWidth() – returns width of the path segment which is being referenced by this PathSegRef object.

isOrthogonal() – returns True, if the two points for the path segment which is being referenced by this PathSegRef object.are orthogonal, and returns False, otherwise.

Feedback

Inherited Methods: This PathSegRef class inherits all methods from the ShapeRef class, as well as from the PhysicalCompRef class.

PolygonRef

Description: The PolygonRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is actually a polygon shape. The Python `isinstance` command can be used to determine if a particular ShapeRef object is a PolygonRef object. This PolygonRef class provides additional methods, to allow the designer to access specific information about the polygon which is being referenced.

Methods:

getNumPoints() – returns the number of points that define the referenced polygon.

getPoints() – returns the list of points that define the referenced polygon.

isOrthogonal() – returns True, if all the points in the referenced polygon are orthogonal, and returns False, otherwise.

Inherited Methods: This PolygonRef class inherits all methods from the ShapeRef class, as well as from the PhysicalCompRef class.

Attributes:

In addition to these methods for the PolygonRef class, there is also a read-only attribute (or property) defined for the PolygonRef class, described as follows:

points – the list of points which define the referenced polygon

Note that this **points** attribute will return the same value as the "**getPoints**()" method.

RectRef

Description: The RectRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a rectangle shape. The Python `isinstance` command can be used to determine if a particular ShapeRef object is a RectRef object.

Methods: There are no methods currently defined for this RectRef class.

Inherited Methods: This RectRef class inherits all methods from the ShapeRef class, as well as from the PhysicalCompRef class.

Feedback

TextRef

Description: The TextRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a text shape. The Python `isinstance` command can be used to determine if a particular ShapeRef object is a TextRef object.

Methods:

getText() – returns the text string from the Text object which is being referenced by this TextRef object.

Inherited Methods: This TextRef class inherits all methods from the ShapeRef class, as well as from the PhysicalCompRef class.

TextDisplayRef

Description: The TextDisplayRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is a text display shape. The Python `isinstance` command can be used to determine if a particular ShapeRef object is an TextDisplayRef object.

Methods: There are no methods currently defined for this TextDisplayRef class.

Inherited Methods: This TextDisplayRef class inherits all methods from the ShapeRef class, as well as from the PhysicalCompRef class.

AttrDisplayRef

Description: The AttrDisplayRef class is derived from the base ShapeRef class, and would be used whenever the referenced shape is an attribute display shape. The Python `isinstance` command can be used to determine if a particular ShapeRef object is an AttrDisplayRef object.

Methods: There are no methods currently defined for this AttrDisplayRef class.

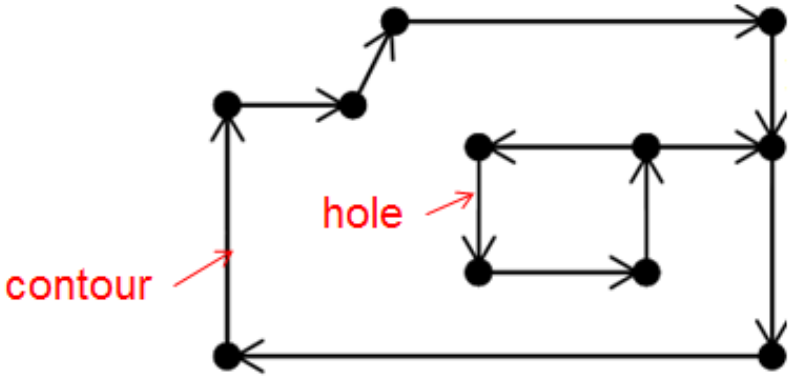
Inherited Methods: This AttrDisplayRef class inherits all methods from the ShapeRef class, as well as from the PhysicalCompRef class.

4

POINT LIST GEOMETRY OPERATIONS

Geometric operations ultimately degenerate to point operations. Therefore, providing points-in and points-out geometric operations in Python can be more useful. The points in a point list can be closed points or unclosed points.

Closed points indicate that the first and last points are treated as connected. The closed points are a one-shot polygon representing the contour and holes in the closed points. The following figure represents the one-shot polygon.



Unclosed points (that is, a polyline) indicate that the first and last points should be treated as unconnected. The unclosed points are not limited to one specific object. They can represent the centerline of the path, the partial contour of the polygon, and so on.

Whether the input uses closed points or unclosed points, they should be non-self-intersecting. If the input points are self-intersecting, the tool drops them.

Point List Geometry Methods

poAnd(Collection *col1*, Collection *col2*) – A two-operand (Collection1 and Collection2) Boolean AND operation. The operand is a closed points-collection representing multiple one-shot polygons. The resulting geometry for the Boolean AND operation is generated as a Collection object, which is the return value for this method.

poInside(Collection *col1*, Collection *col2*) – Select all shapes from Collection1 that share their entire areas with shapes of Collection2. If either Collection1 or Collection2 is empty,

Feedback

there is no output. The resulting geometry is generated as a Collection object, which is the return value for this method. In the following example, shapes s6 and s7 are inside in shape s1.

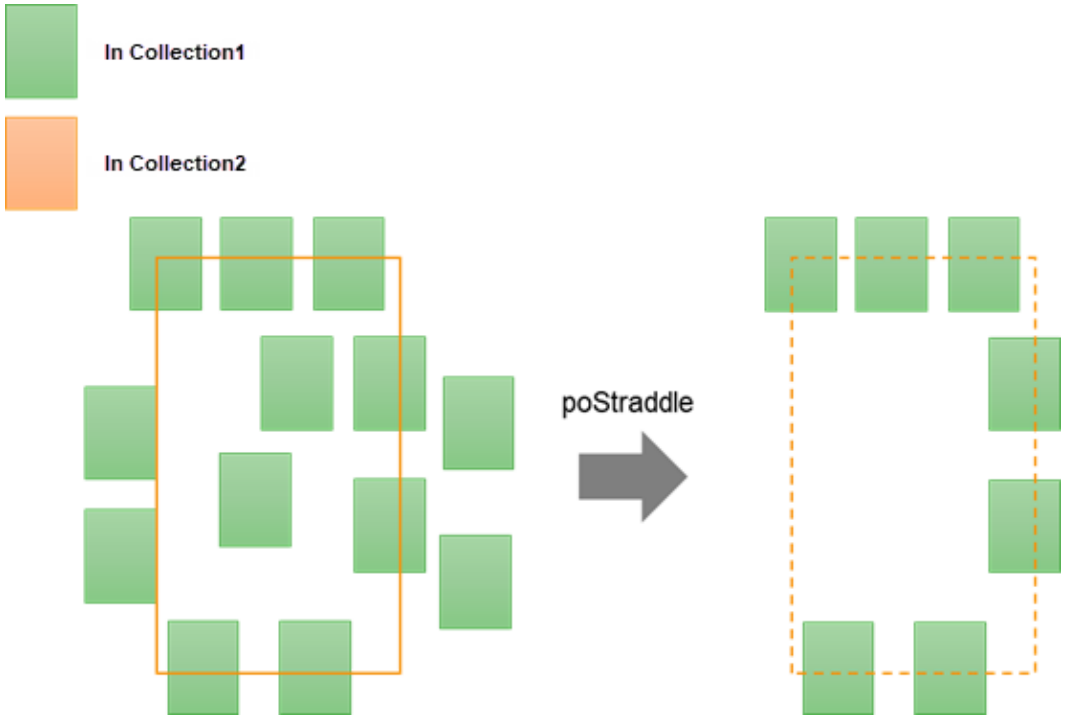
poNot(Collection *col1*, Collection *col2*) – A two-operand (Collection1 and Collection2) Boolean NOT operation. The operand is a closed points-collection representing multiple one-shot polygons. The resulting geometry for the Boolean NOT operation is generated as a Collection object, which is the return value for this method.

poOr(Collection *col1*, Collection *col2*) – A two-operand (Collection1 and Collection2) Boolean OR operation. The operand is a closed points-collection representing multiple one-shot polygons. The resulting geometry for the Boolean OR operation is generated as a Collection object, which is the return value for this method.

poOutside(Collection *col1*, Collection *col2*) – Select all shapes from Collection1 with areas that lie completely outside all shapes of Collection2, where butting shapes should be considered outside. If Collection2 is empty, output all shapes of Collection1. The resulting geometry is generated as a Collection object, which is the return value for this method. In the following example, shapes s2, s3, s4, and s8 are outside of shape s1.

poSize(Collection *col*, Coord *size*) – A one-operand Boolean SIZE operation. The operand is a closed points-collection representing multiple one-shot polygons. Expand or shrink all merged polygons by the specified value. The specified value is a floating-point number in user units. If the specified value is positive, expand the merged polygons and then merge them. If the specified value is negative, shrink the merged polygons. The resulting geometry for the Boolean SIZE operation is generated as a Collection object, which is the return value for this method.

poStraddle(Collection *col1*, Collection *col2*) – Collection1 and Collection2 should contain shapes. Select the shapes in Collection1 that the edges in Collection2 would be across. If the shapes in Collection1 and the shapes in Collection2 only touch along their edges, exclude these shapes.



poXor(Collection *col1*, Collection *col2*) – A two-operand (Collection1 and Collection2) Boolean XOR operation. The operand is a closed points-collection representing multiple one-shot polygons. The resulting geometry for the Boolean XOR operation is generated as a Collection object, which is the return value for this method.

poHull(Collection *col*) – A one-operand (Collection) Boolean HULL operation. The operand is a closed points-collection representing multiple one-shot polygons. The resulting geometry for the Boolean HULL operation is generated as a Collection object, which is the return value for this method.

Points-Collection

Description: The points-collection is the container containing multiple closed points or multiple unclosed points. Closed points represent a one-shot polygon, while unclosed points represent a polyline. A one-shot polygon and a polyline cannot be put in the same container, so a points-collection contains either multiple one-shot polygons or multiple polylines.

If closed points are input for a one-operand operation, the closed points are first merged. If closed points are input for a two-operand operation, the individual operands are first merged, then after merging individually, the corresponding operation is performed.

Feedback

Creation:

Collection(PointList *points*=None, *objects*=None) - creates a Collection object, using the *points* or *objects* parameters. The *points* parameter is passed as uniform list of Point objects. The *objects* parameter is passed as uniform list of PhysicalComponent(s) or PhysicalCompRef(s).

Methods:

addPointLists(PointList | ulist [PointList] *points*) – adds the point(s) in this collection.

addPointListsByObjects(PhysicalComponent | PhysicalCompRef | ulist [PhysicalComponent] | ulist [PhysicalCompRef] *objects*) - adds the physical component(s) or physical component reference(s) in this collection.

createPointListsToShapes(Layer layer, float grid=None, bool asRectangle=False) - returns the list of the Shape. The resulting shapes are generated on the layer when the layer parameter is specified. If the grid parameter is specified, then the points of the shapes are snapped with grid value to the nearest grid point. If the asRectangle parameter is True, the resulting shapes are generated for Rectangle instead if the points are a box.

getPointLists(grid=None) - returns the list of PointList which is being contained in this Collection object. If the grid parameter is specified, then the points are snapped with grid value to the nearest grid point.

5

CONNECTIVITY CLASSES

There are a number of connectivity classes which are provided to allow the designer to model connectivity within their parameterized cell designs. This connectivity information can be used by the designer to trace connectivity through a design, perform manual routing, create netlists for physical verification or circuit simulation, or perform cross-probing between schematic and layout views for a design. In addition, connectivity information can be used in the construction of hierarchical parameterized cell designs.

The Python API provides five major classes for the handling of connectivity information. These classes include the Net, Pin, Term, InstTerm and InstPin classes. In addition, these class objects are used in several other classes, such as the Instance classes and classes derived from the CompoundComponent class, such as the RoutePath class. For example, when a design instance is created, the designer can optionally specify the connectivity which should be associated with the design instance. The RoutePath class is used to construct routes between different pins in a design, such as to connect the source, drain or gate pins between two different transistor instances.

In addition to the five major connectivity classes (Net, Pin, Term, InstTerm and InstPin), there are also two smaller auxiliary classes, which are used by the Net and Term classes to construct connectivity objects with the appropriate properties. The SignalType class is used by the Net class, and the TermType class is used by the Term class.

SignalType

Description: There are several different signal types which can be associated with a Net object. These different signal types are specified by one of the following constant objects, which are defined as attributes for this SignalType class: SIGNAL, POWER, GROUND, CLOCK, TIEOFF, TIEHI, TIELO, ANALOG, SCAN, RESET.

These SignalType objects can be used in conjunction with various functions within the Python API. This SignalType object is typically used to specify the type of signal which should be used when creating a Net object.

There is no need to `construct` any SignalType objects; simply use the SignalType constants such as SignalType.SIGNAL or SignalType.CLOCK.

Methods:

Note that there currently are no methods defined for this SignalType class.

Feedback

Examples:

```
sig1 = SignalType.SIGNAL
sig2 = SignalType.SIGNAL        # same SignalType object
sig1 == sig2                    # returns True
```

TermType

Description: There are several different terminal types which can be defined for a Term object. These different terminal types are specified by one of the following constant objects, which are defined as attributes for this TermType class: INPUT, OUTPUT, INPUT_OUTPUT, SWITCH, JUMPER, UNUSED, TRISTATE.

These TermType objects can be used in conjunction with various functions within the Python API. This TermType object is typically used to specify the type of terminal when a Term terminal object is created.

There is no need to `construct` any TermType objects; simply use the TermType constants such as TermType.INPUT or TermType.OUTPUT.

Methods:

Note that there currently are no methods defined for this TermType class.

Examples:

```
trm1 = TermType.INPUT_OUTPUT
trm2 = TermType.INPUT_OUTPUT    # same TermType object
trm1 == trm2                    # returns True
```

Net

Description: The Net class is the base class for all types of net classes, for example, BusNet. The Net class is used to represent the basic connectivity within a design. The Net object is a non-physical object which associates other connected objects in a design, and so provides the logical connectivity for a design. These Net objects can be connected to terminals on a design instance; these terminals represent logical connection points for the instance. Note that these Net objects can either be created by the designer, or automatically created by the Santana system as necessary. In either case, there are a large number of methods provided for this Net class, which allows the designer to easily use nets in their design. For example, there are methods provided to allow the designer to associate physical shapes with this logical Net object.

Note that the name of any terminal associated with the Net object will always be the same as the name of this net, so that net names and terminal names will always be the same.

Feedback

Creation:

Net(string *netName*, SignalType *sigType*=SignalType.SIGNAL, bool *isGlobal*=False) – creates a Net object in the current design, using the *netName* parameter to name this newly created Net object. If this *netName* string parameter is empty, or is the name of an existing net in the design, then an exception is raised. In addition, the *sigType* parameter is used to specify the signal type for this net, and the *isGlobal* Boolean flag parameter should be set to True, when this net is a global net for the current design.

Methods:

addShape(Shape *shapes*)

addShape(ulist[Shape] *shapes*) – adds the single Shape or each Shape in the list of shapes to this net. If this shape has already been associated with another net, then an exception will be raised, since a shape can only be associated with a single net.

destroy() – destroys this Net object, by first destroying any Term terminal objects associated with this Net object. In addition, any InstTerm objects owned by this Net object are also disconnected from this Net object. The underlying OpenAccess database object (oaNet) is also destroyed.

find(string *name*=) – returns the Net object for a Net having this *name* in the current DloGen design. This is done by searching through all nets in the current DloGen design. If the *name* parameter is the empty string, then the first Net in the design is returned. If there are no nets having this *name* in the current DloGen design, then None is returned.

Note that this method is a static method, so that it can be called without explicitly creating a Net object; it can be directly called using the syntax `Net.find('net1').`

findCreate(string *name*) – returns the Net object for a Net having this *name* in the current DloGen design. This is done by searching through all nets in the current DloGen design. If there are no nets having this *name* in the current DloGen design, then a new Net object is created using the *name* parameter. This method supports BusNet and BundleNet creation, as well.

Note that this method is a static method, so that it can be called without explicitly creating a Net object; it can be directly called using the syntax `Net.findCreate('net1').`

getBit(unsigned int *index*) – for a single-bit net, always returns this Net object.

getInstPins() – returns a list of InstPin objects associated with this net. This is done by finding all InstTerm objects associated with this net, and then finding all InstPin objects associated with these instance terminals.

getInstTerms() – returns a list of InstTerm objects associated with this net.

getName() – returns the name for this Net object.

getNumBits() – returns the number of bits of this Net. This function always returns 1 for scalar nets, but it can return 1 or more for bus nets.

getShapes() – returns a uniform list of the shapes which are currently associated with this net. These shapes will be the ones which were explicitly associated with this net, through the use of the **addShape**() method.

getSignalType() – returns the signal type value assigned to this Net object. This signal type value will be one of the constant enumerated values defined by the SignalType class.

getPins() – returns a list of pins which are connected to any terminal associated with this Net object. All Pin objects associated with this terminal are returned in this list.

getTerm() – returns any terminal that is associated with this net. If there is no terminal associated with this net, then None will be returned. Note that unlike OpenAccess, there can be at most one terminal associated with this Net object.

getVias() – returns a list of all vias which are associated with this net. If there are no vias associated with this net, then None will be returned.

isGlobal() – returns the Boolean flag value indicating that this net is a global net.

removeShape(Shape *shapes*)

removeShape(ulist[Shape] *shapes*) – removes the single Shape or each Shape in the list of shapes from this net. If the specified shape is not associated with this net, then an exception is raised.

setGlobal(bool *isGlobal*) – sets the Boolean flag indicating that this net is a global net.

setName(string *name*) – sets the name for this Net object. Note that the name of any associated terminal for this Net will also be changed to use this new name. This is done in order to ensure that any terminal connected to a net will always have the same name as the net. If this *name* string parameter is empty, or is the name of an existing net or terminal in the current design, then an exception is raised.

setNetOverride(string *assignmentName*, string *netName*) – creates an oaNetConnectDef associating this single-bit net with the oaAssignment named *assignmentName*. If the oaAssignment referred to does not exist, this single-bit net is associated with the net named *netName*. An exception is thrown if the net named *netName* does not exist.

setSignalType(signalType *sigType*) – sets the signal type value for this Net object. This *sigType* signal type value should be one of the constant enumerated values defined by the SignalType class.

Feedback

Attributes:

In addition to these methods for the Net class, there are also four attributes (or properties) defined for this Net class, described as follows:

name – the name for this net

type – the signal type for this net

globalNet – the value of the Boolean flag indicating that this is a global net

AttrType – the display attribute type for this net

The **AttrType** attribute is used with the AttrDisplay class, and has the attributes IS_EMPTY, IS_GLOBAL, IS_IMPLICIT, NAME, NUM_BITS and SIG_TYPE as well as the **getMembers**() class method defined for the Net.AttrType class.

Note that the **name**, **type** and **globalNet** attributes will return the same values as returned by the "**getName**()", "**getSignalType**()" and "**isGlobal**()" methods. In addition, these attributes are settable, so that they can be used to set these values in the same manner as the "**setName**()", "**setSignalType**()" and "**setGlobal**()" methods.

Examples:

```
# use connectivity created by Santana system
# to illustrate basic connectivity structures
# first create basic Contact object
contact1 = Contact(Layer('diff'), Layer('metal1'), 'A')
# create single terminal and pin for our design
term1 = self.addTerm('A', TermType.INPUT_OUTPUT)
pin1 = self.addPin('A', 'A',
                   contact1.getBBox(Layer('metal1')),
                   Layer('metal1'))
# observe that Net object was automatically created
net1 = term1.getNet()
net1.getName()         # returns 'A'
net1.getSignalType()   # returns SignalType.SIGNAL
net1.isGlobal()        # returns False
net1.getTerm()         # returns term1
# check that net names and terminal names are the same
net1.getName()         # returns 'A'
term1.getName()        # returns 'A'
net1.setName('B')      # change net name
net1.getName()         # returns 'B'
term1.getName()        # returns 'B', terminal name changed
net1.getPins()         # returns pin1
```

Feedback

BusNet

Description:A bus net represents a logical connection by one or more bits which are associated with common base name and vector range specification. The BusNet class is derived from the Net class. Bus nets have vector names, for example, G[1:5:2].

Creation:

BusNet(string *baseName*, unsigned int *startIndex*, unsigned int *stopIndex*, unsigned int *indexStep*, SignalType *sigType* = SignalType.SIGNAL, bool *isGlobal* = False) – creates a bus net object, using the *baseName* parameter to name the base part (that is, without bit indexing) of this newly created BusNet object. If this *netName* string parameter is empty, or is the name of an existing net in the design, then an exception is raised. The *startIndex* parameter is the starting bit index, the *stopIndex* is the stopping bit index, and the *indexStep* parameter is the step of the bit index. In addition, the *sigType* parameter is used to specify the signal type for this bus net, and the *isGlobal* Boolean flag parameter should be set to True, when this bus net is a global net for the current design.

Methods:

find(string *name* =) – returns the bus net object having this name in the current DloGen design. This is done by searching through all bus net objects in the current DloGen design. If the *name* parameter is the empty string, then the first bus net in the design is returned. If there is no bus net having this name in the current design, then None is returned.

Note that this method is a static method, so that it can be called without explicitly creating a BusNet object. For example, it can be directly called using the syntax `BusNet.find('G[1:3]')`.

getBaseName() – returns the base part of the name of this bus net (without bit indexing).

getBit(unsigned int *index*) – returns a bus net bit by its index. For example, for a bus net with name "G[3:0]", and *index* = 0, returns the bus net bit with name "G[3]".

getIndexStep() – returns the step of index of the bus net bits.

getNumBits() – returns the number of bits of this bus net.

getStartIndex() – returns the starting index of the bus net bits.

getStopIndex() – returns the stopping index of the bus net bits.

setBaseName(string *baseName*) – sets the base part of this bus net name.

setIndexRange(unsigned int *startIndex*, unsigned int *stopIndex*) – sets the starting and stopping indexes of the bus net bits.

Inherited Methods: This BusNet class inherits all methods defined for the Net class.

Feedback

BundleNet

Description: A bundle net represents a logical connection by a multi-bit net composed of one or more scalar nets, bus nets, or a combination of both, each member net in a bundle net can repeat. The BundleNet class is derived from the Net class.

A bundle net with the name of '2*A,2*B[5],2*C[1:0]' that consists of 8 single bit members: A A B[5] B[5] C[1] C[0] C[1] and C[0]. The single bit net 'C[1]' is the 5th member of that bundle net, it is also the 7th member because the busName representation 'C[1:0]' is set to have a repeat value of 2.

Creation:

BundleNet(string *bundleName*, SignalType *sigType* = SignalType.SIGNAL, bool *isGlobal* = False) – creates a bundle net object, using the *bundleName* parameter. If this *bundleName* string parameter is empty or is the name of an existing net in the design, then an exception is raised. The *sigType* parameter is used to specify the signal type for this bus net, and the *isGlobal* Boolean flag parameter should be set to True, when this bus net is a global net for the current design.

Methods:

find(string *name* =) – returns the bundle net object having this name in the current DloGen design. This is done by searching through all bundle net objects in the current DloGen design. If the *name* parameter is the empty string, then the first bundle net in the design is returned. If there is no bundle net having this name in the current design, then None is returned.

Note that this method is a static method, so that it can be called without explicitly creating a BundleNet object. For example, it can be directly called using the syntax `Bundle.find('A,2*B[5]')`.

getMember(unsigned int *index*) – returns the specified member of this net at the specified bundle member index.

getMembers() – returns a uniform list of the member nets which are contained in this bundle net.

getName() – returns the name of this bundle net.

getNumMembers() – returns the number of members in this bundle net.

getRepeat(unsigned int index) – returns the repeat count for the specified member of this bundle net.

Feedback

Term

Description: The Term class is the base class for all types of terminal classes, for example, BusTerm. The Term class is used to represent the logical connection points within a design. The Term object is a non-physical object which is logically associated with a Net object to logically export the connectivity to the next higher-level in the design hierarchy. This Term object is used to indicate how a net should be connected at an upper level of the hierarchy of the design. In addition, this logical terminal is associated with one or more physical Pin objects, which represent the physical connection points in a design.

Note that the physical Pin objects associated with this terminal represent resistive equivalent physical connection points. For example, in the case of strong connectivity,

there are multiple shapes associated with a single Pin object, and these shapes can be directly routed to provide this strong connection (eg: metal routing). In the case of weak connectivity, there are multiple Pin objects associated with a single terminal (eg: poly routing). In the third case of disjoint connectivity, there are two terminals which need to be connected at a higher level of the design hierarchy. These types of disjoint terminals are collected into a `must join` set of terminals.

Note that the name for a terminal will be the same as any net associated with this terminal, so that terminal names and net names will always be the same.

Creation:

Term(string *termName*, TermType *termType*=TermType.INPUT_OUTPUT) – creates a Term terminal object in the current design, using the *termName* parameter to name this newly created Term object. If there is already a net in the current design having the same name as this terminal, then this net will be connected to this terminal. Otherwise, a new net will be created, and connected to this terminal, using this *termName* string to name this newly created net. If the *termName* string parameter is empty, or is the name of an existing terminal in the design, then an exception is raised. In addition, the *termType* parameter is used to specify the terminal type for this terminal.

Methods:

destroy() – destroys this Term object, by first destroying any Pin objects owned by this Term object. The underlying OpenAccess database object (oaTerm) is also destroyed.

find(string *name*=) – returns the Term object for a Term having this *name* in the current DloGen design. This is done by searching through all terminals in the current DloGen design. If the *name* parameter is the empty string, then the first terminal in the design is returned. If there are no terminals having this *name* in the current DloGen design, then None is returned.

Note that this method is a static method, so that it can be called without explicitly creating a Term object; it can be directly called using the syntax `Term.find('term1')`.

findCreate(string *name*) – returns the Term object for a Term having this *name* in the current DloGen design. This is done by searching through all terminals in the current DloGen design. If there are no terminals having this *name* in the current DloGen design, then a new Term object is created using the *name* parameter. This method supports BusTerm and BundleTerm creation, as well.

Note that this method is a static method, so that it can be called without explicitly creating a Term object; it can be directly called using the syntax `Term.findCreate('term1')`.

getBit(unsigned int *index*) – for a single-bit terminal, always returns this Term object.

getMustJoinTerms() – returns a uniform list of all Term terminal objects which are in the `must-join` set of this terminal. Note that this list will not contain this terminal as a member of the returned list.

getName() – returns the name for this terminal.

getNet() – returns the net associated with this Term terminal object. If there is no associated net, then an exception is raised.

getNumBits() – returns the number of bits of this Term object. This function always returns 1 for scalar terminals, but it can return 1 or more for bus terminals.

getPins() – returns a uniform list of all of the Pin objects associated with this Term terminal object.

getTermType() – returns the terminal type value assigned for this Term object. This terminal type value will be one of the constant enumerated values defined by the TermType class.

setMustJoin(Term *term*=None) – marks this terminal as requiring connection at an upper level of the design hierarchy. Note that if the value of the *term* parameter is None, then this terminal will not be marked as requiring connection at an upper level of the design hierarchy.

setName(string *name*) – sets the name for this terminal. Note that the name of the associated net for this terminal is also changed to have the same name as this terminal. This is done in order to ensure that any net connected to this terminal will always have the same name as this terminal. If this *name* string parameter is empty, or is the name of an existing terminal or net in the design, then an exception is raised.

setNetOverride(string *assignmentName*, string *netName*) – creates an oaTermConnectDef associating this single-bit terminal with the oaAssignment named *assignmentName*. If the oaAssignment referred to does not exist, this single-bit terminal is associated with the net named *netName*. An exception is thrown if the net named *netName* does not exist.

Feedback

setTermType(TermType *termType*) – sets the terminal type value for this Term object. This *termType* terminal type value should be one of the constant enumerated values defined by the TermType class.

Attributes:

In addition to these methods for the Term class, there are also three attributes (or properties) defined for this Term class, described as follows:

name – the name for this terminal

type – the terminal type for this terminal

AttrType – the display attribute type for this terminal

The **AttrType** attribute is used with the AttrDisplay class, and has the attributes HAS_PINS, NAME and NUM_BITS as well as the **getMembers**() class method defined for the Term.AttrType class. Note that the **name** and **type** attributes will return the same values as returned by the "**getName**()" and "**getTermType**()" methods. In addition, these attributes are settable, so that they can be used to set these values in the same manner as the "**setName**()" and "**setTermType**()" methods.

Examples:

```
# use connectivity created by Santana system
# to illustrate basic connectivity structures
# first create basic Contact object
contact1 = Contact(Layer('diff'), Layer('metal1'), 'A')
# create single terminal and pin for our design
term1 = self.addTerm('A', TermType.INPUT_OUTPUT)
pin1 = self.addPin('A', 'A',
                   contact1.getBBox(Layer('metal1')),
                   Layer('metal1'))
# observe that Net object was automatically created
net1 = term1.getNet()
term1.getName()         # returns 'A'
term1.getTermType()     # returns TermType.INPUT_OUTPUT
term1.getNet()          # returns net1
# check that net names and terminal names are the same
term1.getName()         # returns 'A'
net1.getName()          # returns 'A'
term1.setName('B')      # change terminal name
term1.getName()         # returns 'B'
net1.getName()          # returns 'B', net name changed
term1.getPins()         # returns pin1
```

BusTerm

Description: A bus term represents a logical connection point within a design by one or more bits which are associated with common base name and vector range specification.

Feedback

The BusTerm class is derived from the Term class. Bus terms have vector names, for example, G[1:5:2].

Creation:

BusTerm(string *baseName*, unsigned int *startIndex*, unsigned int *stopIndex*, unsigned int *indexStep*, TermType *termType* = TermType.INPUT_OUTPUT) – creates a bus term object, using the *baseName* parameter to name the base part (that is, without bit indexing) of this newly created BusTerm object. If there is already a net in the current design having the same name as this terminal, then this net will be connected to this terminal. Otherwise, a new net will be created, and connected to this terminal, using this *baseName* string to name this newly created net. If the *baseName* string parameter is empty, or is the name of an existing terminal in the design, then an exception is raised. The *startIndex* parameter is the starting bit index, the *stopIndex* is the stopping bit index, and the *indexStep* parameter is the step of the bit index. In addition, the *termType* parameter is used to specify the terminal type for this terminal.

Methods:

find(string *name* =) – returns the bus term object having this name in the current DloGen design. This is done by searching through all bus term objects in the current DloGen design. If the *name* parameter is the empty string, then the first bus term in the design is returned. If there is no bus term having this name in the current design, then None is returned.

Note that this method is a static method, so that it can be called without explicitly creating a BusTerm object. For example, it can be directly called using the syntax `BusTerm.find('G[1:3]')`.

getBaseName() – returns the base part of the name of this bus term (without bit indexing).

getBit(unsigned int *index*) – returns a bus term bit by its index. For example, for a bus term with name "G[3:0]", and *index* = 0, returns the bus term bit with name "G[3]".

getIndexStep() – returns the step of index of the bus term bits.

getNumBits() – returns the number of bits of this bus term.

getStartIndex() – returns the starting index of the bus term bits.

getStopIndex() – returns the starting index of the bus term bits.

setBaseName(string *baseName*) – sets the base part of this bus term name.

setIndexRange(unsigned int *startIndex*, unsigned int *stopIndex*) – sets the starting and stopping indexes of the bus term bits.

Inherited Methods: This BusTerm class inherits all methods defined for the Term class.

Feedback

BundleTerm

Description: A bundle term represents a logical connection by a multi-bit terminal composed of one or more scalar terminals, bus terminals, or a combination of both, each member terminal in a bundle terminal can repeat. The BundleTerm class is derived from the Term class. Bundle terminal have multi names, for example, 2*A,2*B[5],2*C[1:0].

Creation:

BundleTerm(string *bundleName*, TermType *termType* = TermType.INPUT_OUTPUT) – creates a bundle terminal object, using the *bundleName* parameter. If there is already a net in the current design having the same name as this terminal, then this net is connected to this terminal. Otherwise, a new net is created, and connected to this terminal, using this *bundleName* string to name this newly created net. If the *bundleName* string parameter is empty, or is the name of an existing terminal in the design, then an exception is raised. In addition, the *termType* parameter is used to specify the terminal type for this terminal.

Methods:

find(string *name* =) – returns the bundle term object having this name in the current DloGen design. This is done by searching through all bundle term objects in the current DloGen design. If the *name* parameter is the empty string, then the first bundle term in the design is returned. If there is no bundle term having this name in the current design, then None is returned.

Note that this method is a static method, so that it can be called without explicitly creating a BundleTerm object. For example, it can be directly called using the syntax `BundleTerm.find('A,B[3],G[1:3]')`.

getMember(unsigned int *index*) – returns the specified member of this terminal at the specified bundle member index.

getMembers() – returns a uniform list of the member terminals which are contained in this bundle terminal.

getName() – returns the name of this bundle terminal.

getNumMembers() – returns the number of members in this bundle terminal.

Pin

Description: The Pin class is used to represent the physical connection of terminals on a design instance to nets in the design. Note that a terminal can have more than one Pin object, where multiple Pin objects represent multiple physical connections. However, these multiple physical connections are viewed as a single logical connection.

A Pin object can have one or more associated shapes. These shapes represent non-resistive equivalent connection points. In addition, each of these shapes can be named, so that each of the multiple shapes associated with a Pin object can be uniquely identified and accessed as needed. Also note that multiple Pin objects may be associated with a terminal in the design.

Note that the name of a Pin object must be unique for a design, so that it is convenient to easily access a Pin object in the design. This pin name should be a scalar (non-bus) name.

Creation:

Pin(string *pinName*, string *termName*, Shape *shape*=None) – creates a Pin object for the specified terminal in the current design, using the *pinName* parameter to name this newly created Pin object. If there is already a terminal in the current design having the same name as the *termName* parameter, then this terminal will be used to create this Pin object. Otherwise, a new terminal will be created, using this *termName* string to name this newly created terminal. If there is already a pin in the current design having the same name as the *pinName* string parameter, then an exception is raised. In addition, the *shape* parameter can optionally be used to associate a shape object with this pin.

Methods:

addInstPin(InstPin *instPin*, ShapeFilter *filter*=ShapeFilter()) – adds all shape references from the specified *instPin* instance pin, so that they become shapes associated with this pin. These shapes will be created using the shape references which are found on the layers selected by the *filter* ShapeFilter parameter. These shapes will be associated with this pin, using the current coordinate system for this pin. Note that these promoted shapes can be either Rect, Polygon, Path or PathSeg shapes.

addShape(Shape *shapes*)

addShape(ulist[Shape] *shapes*) – adds the single Shape or each Shape in the list of shapes to this pin. If this shape has already been associated with another pin, then an exception will be raised, since a shape can only be associated with a single pin.

destroy() – destroys this Pin object. The underlying OpenAccess database object (oaPin) is also destroyed.

find(string *name*=) – returns the Pin object for a Pin having this *name* in the current DloGen design. This is done by searching through all pins in the current DloGen design. If the *name* parameter is the empty string, then the first Pin in the design will be returned. If there is no pin having this *name* in the current DloGen design, then None is returned.

Note that this method is a static method, so that it can be called without explicitly creating a Pin object; it can be directly called using the syntax `Pin.find('pin1')`.

getAccessDir() – returns the access direction for this Pin; this is the direction from which this Pin should be accessed from the Instance containing this Pin. The return value is a

Direction, or a list of multiple Directions. Note that a NONE Direction indicates that this Pin has no access direction. Also note that if all four primary directions are set as the access direction for this Pin, then Direction.ANY will be returned by this method.

getAuthorPrefLayer() – returns the Layer object which is the author-specified preferred layer for routing purposes. If there is no author preferred layer defined (using the pre-defined property name), then None will be returned.

getBBox(ShapeFilter *filter*=ShapeFilter()) – returns the bounding box of the geometries on the layers specified by the *filter* ShapeFilter parameter. This bounding box will be the merged bounding box for all of the layers on which this pin has associated geometries.

getLayers() – returns the list of layers on which this pin has associated geometries. If this pin does not have geometries on any layers, then an empty list will be returned.

getName() – returns the name for this Pin object.

getNet() – returns the Net object which is associated with this Pin.

getPrefLayer() – returns the layer which is the author-specified preferred layer for this pin. If there is no author preferred layer defined (using the pre-defined property name), then the top layer on which this pin has geometry will be returned (the same layer as returned by the **getTopLayer**() method). If this pin does not have geometries on any layers, then None will be returned. If all layers for this pin do not have mask numbers, then None will be returned.

getPrefLayerPropName() – returns the name of the OpenAccess property which is used to specify the name string for the author-preferred Layer object which should be used for this Pin object. This property is used to specify which of the layers having geometries for the Pin object should be the preferred layer for routing purposes.

getShapes() – returns a uniform list of the shapes which are currently associated with this pin. These shapes will be the ones which were explicitly associated with this pin, through the use of the **addShape**() method.

getTerm () – returns the terminal associated with this Pin object; if there is no terminal associated with this Pin object, then an exception is raised.

getTopLayer() – returns the top Layer object on which this pin has associated geometries. This top layer has the highest layer number in the Technology file, for the layers on which this pin has associated geometries. If this pin does not have geometries on any layers, then None will be returned. Note that only layers with mask numbers will be considered, since mask numbers are used for layer ordering.

removeShape(Shape *shapes*)

removeShape(ulist[Shape] *shapes*) – removes the single Shape or each Shape in the list of shapes from this pin. If this shape is not associated with this pin, then an exception is raised.

Feedback

setAccessDir(Direction | ulist[Direction] *dir*) – sets the access direction for this Pin; this is the direction from which this Pin should be accessed from the Instance containing this Pin. Note that this access direction can be either a single Direction of a list of Directions. Also note that a NONE Direction for this *dir* parameter indicates that this Pin has no access direction.

setAuthorPrefLayer(Layer *layer*) – sets the Layer object which is the author-specified preferred layer for routing purposes. This preferred layer will be defined using the pre-defined OpenAccess preferred layer property name. Note that this property name can be determined using the **getPrefLayerPropName**() method.

setName(string *name*) – sets the name for this Pin object. If there is already a pin in the current design with the same name as the *name* parameter, an exception is raised.

setTerm (Term *term*) – associates the *term* terminal with this Pin object. Note that multiple Pin objects may be associated with a single terminal.

Attributes:

In addition to these methods for the Pin class, there is also an attribute (or property) defined for this Pin class, described as follows:

name – the name for this pin

Note that this **name** attribute is settable, so that it can be used to set the name for this Pin object. Thus, this attribute can be used in the same way as the "**getName**()" and "**setName**()" methods.

Examples:

```
# use connectivity created by Santana system
# to illustrate basic connectivity structures
# first create basic Contact object
contact1 = Contact(Layer('diff'), Layer('metal1'), 'A')
# create single terminal and pin for our design
term1 = self.addTerm('A', TermType.INPUT_OUTPUT)
pin1 = self.addPin('A', 'A',
                   contact1.getBBox(Layer('metal1')),
                   Layer('metal1'))
# observe that Net object was automatically created
net1 = term1.getNet()
pin1.getName()          # returns 'A'
pin1.getNet()           # returns net1
pin1.getTerm()          # returns term1
# pin names are not synchronized with net and terminal names
 pin1.getName()          # returns 'A'
net1.getName()          # returns 'A'
term1.getName()         # returns 'A'
pin1.setName('B')       # change pin name
pin1.getName()          # returns 'B', new pin name
net1.getName()          # still returns 'A'
```

Feedback

```
term1.getName()         # still returns 'A'
# can access shapes used by this Pin object
pin1.getShapes()        # returns rectangle shape
# can read and set access direction for this Pin object
pin1.getAccessDir()     # returns Direction.NONE
pin1.setAccessDir(Direction.NORTH)
```

InstTerm

Description: The InstTerm class is used to represent the connection between a net and a terminal of the master of an instance in the design. These instance terminals will automatically be created, whenever an instance is created within a design. This instance terminal will usually have a connection to a net in the design. However, if there is no associated net for the InstTerm object, then the instance terminal is disconnected.

Creation:

Note that there is no creation method available for this InstTerm object. That is because these instance terminals are automatically created, whenever an instance is created. That is, InstTerm objects are automatically created upon instantiation.

Methods:

find(string *name*, Instance *inst*) – returns the InstTerm object having this *name* for the Instance *inst* in the current DloGen design. This is done by searching through all InstTerm objects for the given instance *inst* in the current DloGen design. If there is no InstTerm having this *name* for this instance in the current DloGen design, then None will be returned. This method works the same as the Instance.findInstTerm() method.

Note that this method is a static method, so that it can be called without explicitly creating an InstTerm object; it can be directly called using the syntax `InstTerm.find('instTerm1')`.

findInstPin(string *name*=) – returns the InstPin associated with this InstTerm, having the name specified by the *name* parameter. This is done by searching through the set of instance pins for this InstTerm object. Note that if there is no InstPin having this *name* in this set of instance pins, then None will be returned. If the *name* parameter value is not specified, then the first InstPin in the set of instance pins will be returned.

getBit(unsigned int *index*) – returns the InstTerm object that corresponds to the specified index bit of this instance terminal. For a single-bit instance terminal, this function always returns this InstTerm object.

getInst() – returns the Instance object associated with this InstTerm object. If there is no instance associated with this InstTerm, then an exception is raised.

Feedback

getInstPins() – returns a uniform list of all of the InstPin objects for this InstTerm. This is done by obtaining each InstPin object from the instance which belongs to this instance terminal.

getMustJoinInstTerms() – returns a uniform list of all InstTerm objects which must be joined together. This is determined by finding the `must-join` set for the terminal in the instance's master. This is a list of instance terminals which must be joined together by a router being used to connect instances in a design.

getNet() – returns the Net object associated with this InstTerm object. If this InstTerm is not connected to any net, then None is returned.

getNumBits() – returns the number of bits of this InstTerm object.

getTermName() – returns the name for this InstTerm object. This is the same name as the terminal in the instance's master.

getTermType() – returns the terminal type value for this InstTerm object. This is the same type as the terminal in the instance's master. This terminal type value is one of the constant enumerated values defined by the TermType class.

setNet(Net *net*) – adds this InstTerm to the specified Net object. If this InstTerm is already connected to another net, it is first removed from the other net. If the net parameter is None, then this InstTerm is removed from any net to which it is connected; if this InstTerm is not connected to any net, then this method does nothing.

Attributes:

In addition to these methods for the InstTerm class, there are also read-only attributes (or properties) defined for this InstTerm class, described as follows:

termName – the name of this instance terminal

AttrType – the display attribute type for this instance terminal

The **AttrType** attribute is used with the AttrDisplay class, and has the attribute NAME, as well as the **getMembers**() class method defined for the InstTerm.AttrType class.

Note that the **termName** attribute will return the same value as is returned by the "**getTermName**()" method.

Examples:

```
# create instance of basic Python transistor example,
# to obtain automatically generated instance terminals
p = ParamArray()
# define connectivity for this instance
conn = {'D':'d', 'G':'g', 'S':'s', 'B':'b'}
inst1 = Instance('cnPyBasicDlo/PyTransistor', p, conn, 'tran1')
# get instance terminal
instTerm1 = inst1.getInstTerms()[1]
```

Feedback

```
instTerm1.getInst()          # returns transistor instance
instTerm1.getNet()           # returns Net('d')
instTerm1.getTermName()      # returns 'D'
instTerm1.getInstPins()      # returns single instance pin
```

InstPin

Description: The InstPin class is used to represent a mapping of the pin in the instance master into the current coordinate system. Since this InstPin object is simply a mapping, these InstPin objects are read-only, and created whenever an instance is created in the current design. These InstPin objects are provided to simplify calculations in the context of a particular instance. For example, these instance pins can be used to simplify routing or coordinate calculations.

Creation:

Since this InstPin is just a mapping of the pin in the instance master into the current coordinate system, this InstPin object is read-only, which can not be created or destroyed. These instance pins will be created whenever an instance is created in the current design. That is, InstPin objects are automatically created upon instantiation. Note that unlike other classes concerned with connectivity (Net, Term, Pin, InstTerm), there is not any OpenAccess equivalent object for this InstPin class object. This reflects the fact that this InstPin class is provided as a convenience, and is not required to define the basic connectivity information for a design.

Methods:

find(string *name*, Instance *inst*)

find(string *name*, InstTerm *instTerm*) – returns the InstPin object having the name specified by the *name* parameter. The instance pins are obtained from either the *inst* Instance or the *instTerm* InstTerm object. This is done by searching through the set of instance pins from each of these objects. Note that if there is no InstPin having this *name* in this set of instance pins, then None will be returned. This method works the same as the Instance.findInstPin() and InstTerm.findInstPin() methods.

Note that these **find**() methods are static methods, so that they can be directly called.

getAuthorPrefLayer() – returns the author-specified preferred layer for routing purposes for the corresponding pin in the instance submaster. If there is no author preferred layer defined, then None will be returned.

getBBox(ShapeFilter *filter*=ShapeFilter()) – returns the bounding box for the corresponding pin in the instance submaster, for which this pin has geometries on the layers specifed by the *filter* ShapeFilter parameter. This bounding box will be the merged bounding box for all of the layers on which this pin has associated geometries.

getInst() – returns the instance for this InstPin object

getInstTerm() – returns the InstTerm for this InstPin object.

getLayers() – returns the list of layers on which the corresponding pin in the instance submaster has associated geometries. If this pin does not have geometries on any layers, then an empty list will be returned.

getNet() – returns the net for this InstPin object.

getPinAccessDir() – returns the access direction for the corresponding pin in the instance submaster; this is the direction from which this pin should be accessed from the instance containing this pin. The return value is either a Direction, or a list of multiple Directions. Note that a NONE Direction indicates that this corresponding pin has no access direction.

getPinName() – returns the pin name for this InstPin object.

getPrefLayer() – returns the Layer object which is the author-specified preferred layer for the corresponding pin in the instance submaster. If there is no author preferred layer defined (using the pre-defined property name), then the top layer on which the pin has geometry will be returned (the same layer as returned by the **getTopLayer**() method).

If all pin layers have no mask number, then None will be returned.

getShapeRefs(ShapeFilter *filter*=ShapeFilter()) – returns a list of shape references for all shapes which are associated with the corresponding pin in the instance submaster. These shape references are a read-only representation of the shape information for the shapes which are associated with this instance pin. Note that only the RectRef, PolygonRef, PathRef and PathSegRef shape reference types will be returned; all other reference types will be filtered out.

getTermName() – returns the name of the terminal for this InstPin object.

getTermType() – returns the terminal type value for this InstPin object. This is the same type as the terminal in the instance's master. This terminal type value is one of the constant enumerated values defined by the TermType class.

getTopLayer() – returns the top Layer object on which the corresponding pin in the instance submaster has associated geometries. This top Layer object has the highest layer number in the technology file, for the layers on which this pin has associated geometries. Note that only layers with mask numbers will be considered, since mask numbers are used for layer ordering.

Attributes:

In addition to these methods for the InstPin class, there is also a read-only attribute (or property) defined for this InstPin class, described as follows:

pinName – the name for this InstPin object

Note that this **pinName** attribute will return the same value as returned by the "**getPinName**()" method.

Feedback

Examples:

```
# create instance of basic Python transistor example,
# to obtain automatically generated instance pins
p = ParamArray()
# define connectivity for this instance
conn = {'D':'d', 'G':'g', 'S':'s', 'B':'b'}
inst1 = Instance('cnPyBasicDlo/PyTransistor', p, conn, 'tran1')
# get instance pin
instPin1 = inst1.getInstPins()[0]
instPin1.getInst()           # returns transistor instance
instPin1.getNet()            # returns Net('d')
instPin1.getInstTerm()       # returns InstTerm('D')
instPin1.getTermName()       # returns 'D'
instPin1.getPinName()        # returns 'D'
instPin1.getPinAccessDir()   # returns Direction.NONE
instPin1.getLayers()
instPin1.getTopLayer()       # returns Layer('metal4')
instPin1.getPrefLayer()      # returns Layer('metal4')
```

RouteTarget

Description: The RouteTarget class is used by the RoutePath class, and is used to specify the shapes which should be connected by a route path. These shapes can be either rectangle shapes or the shapes associated with a Pin or InstPin object. Note that any Rect, RectRef, Pin or InstPin object will automatically be converted to the appropriate RouteTarget object. Thus, when RouteTarget objects are specified as parameters for any RoutePath class methods, then a Rect, RectRef, Pin or InstPin object can be used instead; this object will automatically be converted to the corresponding RouteTarget object. Alternatively, the RouteTarget object can be explicitly constructed and used. The benefit of this latter approach is that it allows the designer to use the different methods described below for the RouteTarget class, so that additional information about the generated RoutePath can be specified. For example, the point inside the shape which should be used for the RoutePath starting or ending point can be specified for the RouteTarget. This information will be used by the RoutePath class methods to guide the generation of these access points. If these access points are not specified, then the center point for the bounding box of the shape is used.

This RouteTarget represents a target to "route from" or "route to" by the different methods for the RoutePath class. This RouteTarget object basically consists of one or more route boxes, along with the layers on which these route boxes are defined. In the case of a Rect or RectRef, the route box will be constructed using the associated rectangle. In the case of a Pin or InstPin, these route boxes will be constructed using the shapes associated with these pins or instance pins. These route boxes will then be used as the shapes which are connected by the different route paths generated by the different methods for the RoutePath class.

After a RouteTarget is created, and used by a RoutePath routing object, there are a large number of methods which are provided to allow the user to obtain information about the RouteTarget object. This includes information such as the route box (or boxes) that will be used for routing, the layer being used for routing, the access point for starting or stopping a route path, and whether a contact is needed.

Creation:

There are four different sets of parameters which can be used to create this RouteTarget 1) a Rect rectangle, 2) a RectRef rectangle reference, 3) an instance pin InstPin, or 4) a Pin pin. In addition, note that an optional point can be specified as the access point for this route target. This access point specifies where the generated RoutePath routing object should start or stop. If this access point is not specified, then the center of the route boxes for this RouteTarget will be used as the starting or stopping point. Note that the center of these route boxes should lie on a grid point, so that the coordinates for these route boxes should lie on even-numbered grid points. Otherwise, an exception will be generated, when this RouteTarget is used to create a RoutePath.

These four different creation calls have the following parameter signatures:

RouteTarget(Rect *rect*, Point *point*=None)

RouteTarget(RectRef *rectRef*, Point *point*=None)

RouteTarget(InstPin *instPin*, Point *point*=None)

RouteTarget(Pin *pin*, Point *point*=None)

Methods:

getBBox(ShapeFilter *filter*=ShapeFilter()) – returns the bounding box for the route boxes contained in this RouteTarget. If this RouteTarget contains only a single route box, then this route box is returned. If this RouteTarget contains more than one route box, then the merged boxes for all of these route boxes is returned.

getChosenAccessDir() – returns any access direction which was selected for this RouteTarget; if no access direction was selected for this RouteTarget, then the Direction NONE will be returned. The access direction will be set by any RoutePath object, when this RouteTarget is used by a RoutePath object.

getChosenAccessPoint() – returns the access point which is selected by a RoutePath object for this RouteTarget; if no access point has been selected, then None will be returned. The access point can be specified when this RouteTarget is created. It will also be set by any RoutePath object, when this RouteTarget is used by a RoutePath object.

getChosenBox() – returns the box for the chosen shape in this RouteTarget. If this RouteTarget is not used by a RoutePath object, then an inverted box is returned. Otherwise, the bounding box of the shape selected by the RoutePath object is returned.

Feedback

getChosenLayer() – returns layer on which the chosen route box is defined for this RouteTarget. If no shape has been chosen for this RouteTarget, then this method will return None. Note that the RoutePath object will select a shape from this RouteTarget, if it has multiple shapes.

getLayers() – returns the list of layers on which any route boxes are defined for this RouteTarget.

getName() – returns the name generated for this RouteTarget. Note that this name will be generated using the base names "Rect", "RectRef", "Pin" or "InstPin", if this RouteTarget was created from a Rect, RectRef, Pin or InstPin object.

getPrefLayer() – returns the name of any preferred layer. Note that this will be set up when this RouteTarget is constructed, using any preferred layers specified for pins or instance pins, and the layer on which any rectangle was created.

getRoutePathIntersectBox() – returns the box which is the intersection of the route box (or merged bounding boxes) for this RouteTarget and the RoutePath which is using this RouteTarget. If this RouteTarget has not been used by any RoutePath object, then a large inverted box will be returned.

getRoutePathLayer() – returns the layer which will be used by the RoutePath object which is being used with this RouteTarget. If this RouteTarget is not currently being used by any RoutePath object, then None will be returned.

hasLayer(Layer *layer*) – returns True if this RouteTarget has a route box defined on the specified *layer* and returns False, otherwise.

isSingleBox() – returns True if this RouteTarget has only a single route box, and returns False, otherwise.

isValid() – returns True if this RouteTarget is a valid route target and returns False, otherwise. This RouteTarget will be valid, if this RouteTarget has one or more associated route boxes.

needsContact() – returns True if this RouteTarget requires that a contact be generated to properly connect to the RoutePath object, and returns False, otherwise. This will be done by examining the layer used by the selected shapes for this RouteTarget and the route path layer used by the RoutePath object. This method compares the chosen layer with any RoutePath layer. Note that True will be returned when this RouteTarget is not being used by a RoutePath object.

Examples:

```
# create RouteTarget using rectangle creation method;
# note that the access point is also specified
rect1 = Rect(Layer('metal1'), Box(0,0,1,1))
rt1 = RouteTarget(rect1, Point(0,0))
# now get information about this RouteTarget
rt1.isValid()                 # returns True
```

Feedback

```
rt1.isSingleBox()              # returns True
rt1.getName()                  # returns Rect(metal1,Box(0,0,1,1))
rt1.hasLayer(Layer('metal1')) # returns True
rt1.hasLayer(Layer('metal2')) # returns False rt1.getChosenLayer()
   # returns Layer('metal1')
rt1.getChosenAccessDir()       # returns Direction.NONE
rt1.getChosenAccessPoint()     # returns Point(0,0)
rt1.getRoutePathLayer()        # returns None
rt1.needsContact()             # returns True
```

Topology

Description: The Topology class provides the ability to store a string which describes the basic topological configuration for a design object. As an example, consider a parameterized PyCell design for a transistor which uses a parameter to allow the user to optionally generate bars for the generated transistor. In addition, this Topology class can be used to allow the user to specify exactly how the bars for the gate, source and drain regions of the transistor should be laid out. For example, the string `G-DS` would indicate that the gate bar should be laid out on the right side of the transistor, while the drain bar would be laid out on the left side, and that the source bar would be laid out to the left of the drain bar.

Creation:

The Topology class object is created by calling the Topology class with the desired string. For example, the Topology object described in the previous paragraph would be created using the following Python code: `top1 = Topology("G-DS`)."

Methods:

getName() – returns the string value for this Topology object

setName(string *name*) – sets the string value for this Topology object

Examples:

```
top1 = Topology('G-DS')
top1.getName ()           # returns 'G-DS'
top1.setName('S-DG')
top1.getName ()           # returns 'S-DG'
top1.setName('D-SG')
top1.getName ()           # returns 'D-SG'
top1.setName('DS-G')
top1.getName ()           # returns 'DS-G'
#####
```

TECHNOLOGY RELATED CLASSES

This section documents the Python API which is used to access the information that is stored in the technology file which is used by the Santana system. In addition, layers are discussed in this section, since layer objects are an integral part of the technology file and the resulting Python API for technology file objects.

The Santana system uses a technology file to store all of the technology-specific information which should be used in the design of a parameterized cell. This Santana technology file contains the following types of technology-related information: layer names and numbers, purpose names and numbers, user units and user-to-database unit conversion factors, physical design rules, electrical design rules, oxide definitions and MOSFET transistor parameter values. Note that these physical design rules can include single layer minimum spacing, minimum area and minimum width rules, as well as two layer minimum enclosure rules. In general, these physical design rules can include non-layer rules, single layer rules and two-layer rules. These electrical design rules can include single layer sheet resistance, current density and area capacitance rules, as well as two layer edge capacitance and area capacitance rules. These oxide definitions are used to specify the appropriate values for different oxide types, such as `thin` and `thick`.

In turn, these different oxide types are then used to specify various MOSFET transistor model values.

The technology information contained in this Santana technology file then gets put into an OpenAccess Santana technology library, which can then be used to access technology-specific information during the parameterized cell design process. This technology library is generated from the Santana technology file through the use of the `cngenlib` library generation utility program. As part of the Python environment, this OpenAccess Santana technology library gets bound to a Tech technology object, which can then be accessed through various class objects in the Python API. In particular, the Dlo/DloGen, Lib and ParamSpecArray class objects have a `tech` attribute which can be used to access this Tech object. Additionally, there is a static method defined for this Tech class, which allows the Tech class object to be directly created from a Santana technology library.

One of the fundamental objects in the technology files is the Layer object. This Layer object is used to represent the layer in the integrated circuit, and stores all of the basic information about the layer. For example, the Layer object contains the layer name, number and purpose which are stored in the technology file.

Layer

Description: The Layer class is used to store the basic information about a layer. This would include information such as the name of the layer, the purpose of the layer (eg: drawing), the layer number and the purpose number. Note that there are several pre-defined purpose names (as fully described in the `Santana Technology File` reference manual), including the ‘drawing’ and ‘all’ purposes. The ‘drawing’ purpose is used to indicate that the purpose of this Layer object is drawing, while the ‘all’ purpose is used as

a short-hand notation to denote all allowed purposes. All other purpose names which are not pre-defined are user-defined, and should be mapped to a specific purpose number in the technology file. The combination of a layer name and a purpose name in the Layer object creation method denotes a `Layer Purpose Pair`.

Note that design rules can be defined for a particular `Layer Purpose Pair` (LPP), so that the same design rule can have different values for the same layer but different purposes. For example, metal spacing design rules can provide different values for different purposes on the same metal layer.

Creation:

Layer(string *layerName*, string *purposeName*='drawing') – creates a Layer object corresponding to the named layer defined in the technology file. If there is no layer defined in the technology file having the same name as the *layerName* parameter, then an exception is raised. Note that the default layer purpose will be 'drawing' purpose.

Methods:

getAttrs() – returns the following attributes as a string format for this layer:

1. *priority* is the display priority assigned to the layer-purpose pair.
2. *visible* indicates whether the layer-purpose pair is visible in the display device.
3. *selectable* indicates whether objects drawn with the layer-purpose pair are selectable.
4. *contToChgLay* indicates whether the layer-purpose pair contributes to a changed layer.
5. *dragEnable* indicates whether you can drag a shape created with the layer-purpose pair in the layout editor.
6. *validity* indicates whether the layer-purpose pair is valid.

getGridResolution() – returns the manufacturing grid resolution value, in user units. This value will be any layer specific manufacturing grid resolution value which is defined in the technology file. If no layer-specific grid resolution value is defined for this layer in the technology file, then this value will be the default grid resolution value defined for all layers in the technology file.

getLayerAbove()

getLayerAbove(LayerMaterial *material*) – returns the layer immediately above this layer in the mask layer sequence, as determined from the layer mask number information stored in the technology file. If the *material* parameter is specified, then the returned layer will be the first layer above this layer having the specified LayerMaterial value.

If this layer has the highest mask number, then None will be returned. If this layer is not a mask layer, then an exception is raised.

getLayerBelow()

Feedback

getLayerBelow(LayerMaterial *material*) – returns the layer immediately below this layer in the mask layer sequence, as determined from the layer mask number information stored in the technology file. If the *material* parameter is specified, then the returned layer will be the first layer below this layer having the specified LayerMaterial value.

If this layer has the lowest mask number, then None will be returned. If this layer is not a mask layer, then an exception is raised.

getLayerName() – returns the name of this layer.

getLayerNumber() – returns the number of this layer, as defined in the technology file.

getMaterial() – returns the material used for this layer, as defined in the technology file. The material value is one of the following: NWELL, PWELL, NDIFF, PDIFF, NIMPLANT, PIMPLANT, POLY, CUT, METAL, CONTACTLESS_METAL, DIFF, RECOGNITION, or UNKNOWN. (Note that each of these returned material values is prefixed with the LayerMaterial name; for example, LayerMaterial.NWELL).

getPurposeName() – returns the purpose name for this layer.

getPurposeNumber() – returns the purpose number for this layer.

getRoutingDir() – returns the routing direction of this layer, as defined in the technology file. The routing direction value is one of the following: NOT_APPLICABLE, HORIZONTAL, VERTICAL, LEFT_DIAG, RIGHT_DIAG, or NONE. (Note that each of these returned material values is prefixed with the RoutingDir name; for example, RoutingDir.VERTICAL).

isAbove(Layer *layer*) – returns True if this layer is above the *layer* which is passed as a parameter, and will return False otherwise. This will be determined from the layer mask number information stored in the technology file.

isMaskLayer() – returns True if this layer is defined as a mask layer, and will return False otherwise. This will be determined from the layer mask number information stored in the technology file.

Attributes:

In addition to these methods for the Layer class, there are also four attributes (or read-only properties) defined for this Layer class, described as follows:

name – string that is the name for this layer (eg: `metal1`)

number – the number defined for this layer

purposeName – the name for the purpose defined for this layer

purposeNumber – the purpose number defined for this layer

Note that these four attributes will return the same values as the “**getLayerName**()”, “**getLayerNumber**()” , “**getPurposeName**()” and “**getPurposeNumber**()” methods.

Feedback

Examples:

```
layer1 = Layer('metal1')
layer1.getAttrs()           # returns the attributes with the
                            # format "(42 (t) (t) (f) (f) (t) )"
layer1.getLayerNumber()     # returns layer number from tech file
layer1.getPurposeName()     # returns name for drawing purpose
layer1.getPurposeNumber()   # returns number for drawing purpose
layer2 = Layer('metal2')
layer2.isAbove(layer1)      # returns True
layer1.isAbove(layer2)      # returns False
layer1.name                 # returns 'metal1'
layer2.name                 # returns 'metal2'
# define Layer Purpose Pair (LPP) using layer metal1
layer3 = Layer('metal1', 'pin')
layer3.getPurposeName()     # returns 'pin' purpose name
 layer3.getPurposeNumber()   # returns 'pin' purpose number
```

ShapeFilter

Description: The ShapeFilter class is used to define the different layers which should be considered when making bounding box calculations as well as placement calculations. This class allows the designer to specify a layer, or a list of layers, which should be interpreted (or filtered) when performing bounding box and placement calculations. This ShapeFilter object is used by a number of class methods for the Python API. For example, the ShapeFilter object can be used as a parameter for handling bounding box calculations for the **getBBox**() methods for the Shape, Instance, Pin, InstPin, PhysicalComponent, Grouping and DloGen classes, as well as the **getMemberBBox**() method for the InstanceArray class. In addition, the creation method for the ContactRing class makes use of this ShapeFilter object. In addition, various placement related methods, such as the **place**() and **fgPlace**() methods for the PhysicalComponent class, also make use of this ShapeFilter object for placement related calculations.

Creation:

ShapeFilter() – creates an empty ShapeFilter object, which would be used to indicate that all layers should be considered in any bounding box or placement calculations.

ShapeFilter(Layer *layer*) – creates a ShapeFilter object, consisting of only a single layer. This single layer is the only layer which should be considered when the ShapeFilter object is passed to methods which perform bounding box and placement calculations.

ShapeFilter(LayerList *layerList*) – creates a ShapeFilter object, consisting of a list of Layer objects. These layers are the only layers which should be considered when the ShapeFilter is passed to methods which perform bounding box or placement calculations.

ShapeFilter(ShapeFilter *shapeFilter*) – creates a ShapeFilter object, consisting of the Layer objects which are specified by the *shapeFilter* ShapeFilter object. These layers are

Feedback

the only layers which should be considered when the ShapeFilter is passed to methods which perform bounding box or placement calculations.

Methods:

exclude(Layer *layer*) – removes the layer specified by the *layer* parameter from the list of layers which are considered by this ShapeFilter object.

exclude(LayerList *layerList*) – removes the list of layers specified by the *layerList* parameter from the list of layers which are considered by this ShapeFilter object.

excludeTexts() – filters out all Text and AttrDisplay objects.

include(Layer *layer*) – adds the layer specified by the *layer* parameter to the list of layers which are considered by this ShapeFilter object.

include(LayerList *layerList*) – adds the list of layers specified by the *layerList* parameter to the list of layers which are considered by this ShapeFilter object.

isIncluded(Layer *layer*) – returns true if the *layer* parameter is a layer which is in the list of layers considered by the ShapeFilter object, and returns False otherwise.

Examples:

```
# construct filter with all layers available
filter1 = ShapeFilter()
# construct filter only for 'metal1' layer
filter2 = ShapeFilter(Layer('metal1'))
# construct filter to include only first two metal layers
filter3 = ShapeFilter([Layer('metal1'), Layer('metal2')])
# define filter to include all possible layers except diffusion
filter4 = ShapeFilter().exclude(Layer('diff'))
filter4.isIncluded(Layer('diff'))    # returns False
filter4.isIncluded(Layer('metal1'))  # returns True
filter4.isIncluded(Layer('poly1'))   # returns True
```

LayerMaterial

Description: There are several different material types which can be used to classify the different physical layers on a chip. These material types describe the fundamental electrical purpose for the layer. These layer material types are specified by one of the following constant objects, which are defined as attributes for this LayerMaterial class: NWELL, PWELL, NDIFF, PDIFF, NIMPLANT, PIMPLANT, POLY, CUT, METAL, CONTACTLESS_METAL, DIFF, RECOGNITION or UNKNOWN.

These LayerMaterial objects can be used in conjunction with various functions within the Python API Tech classes. This LayerMaterial object is used to specify the type of material for a Layer object, and is typically assigned in the technology file. There is no need to

`construct` any LayerMaterial objects; simply use the LayerMaterial constants such as LayerMaterial.METAL or LayerMaterial.POLY.

Methods:

include(Layer *layer*) – adds the layer specified by the *layer* parameter to the list of layers which are considered by this ShapeFilter object.

include(LayerList *layerList*) – adds the list of layers specified by the *layerList* parameter to the list of layers which are considered by this ShapeFilter object.

Examples:

```
lm1 = LayerMaterial.METAL
lm2 = LayerMaterial.METAL
lm1 == lm2          # returns True
```

PhysicalRule

Description: Due to the complexity of physical design rules, the value for a physical design rule needs to be represented as a higher-level object, rather than simply as a single floating point number. For example, the design rules for minimum extension of one layer over another may not be just a single value. Rather, it may be represented as a pair of values, one for the extension in the horizontal direction and one for the extension in the vertical direction. This is most conveniently handled as a pair of floating point values.

In general, this PhysicalRule object can return values representing a single floating-point number, a pair of floating-point numbers, or a list of of pairs of floating-point numbers. This approach provides the necessary flexibility to represent different types of physical design rules.

These PhysicalRule objects are used in conjunction with the Python API Tech class. This PhysicalRule object is used as the return type for the **getPhysicalRule**() method for the Tech class. Note that there is no need to `construct` any PhysicalRule objects; they will be created and returned automatically by the Tech class when this **getPhysicalRule**() method is used. Note that this PhysicalRule class is derived from the basic Python float floating-point number class. Thus, when a single number is returned in this PhysicalRule object, the result can be used just like a floating-point number.

Methods:

Note that there are no methods defined for this PhysicalRule class.

Inherited Methods: This PhysicalRule class inherits all methods from the basic Python float floating-point number class.

Feedback

Attributes:

There are four read-only attributes (or properties) defined for the PhysicalRule class, described as follows:

comment – the comment of the PhysicalRule object.

id – the id of the PhysicalRule object.

properties – Python dictionary containing any properties defined for this PhysicalRule object; the keys are parameter names, and the values are RuleProperty objects (as described in the next section).

value – the value of the PhysicalRule object; this is either a CoordValue, a DualCoordValue or a DualCoordArrayValue (Note that these types represent a floating-point number, a pair of floating-point numbers, or a list of pairs of floating-point numbers).

Examples:

```
# obtain value for minimum width for metal1
pr1 = self.tech.getPhysicalRule('minWidth', Layer('metal1'))
# return value is PhysicalRule object
pr1.value       # returns 0.18
pr1.properties  # returns { } empty dictionary, no properties
# can also use returned value like floating-point number
pr1 * 2         # returns 0.36
# obtain value for minimum extension of diffusion over contact pr2 =
 self.tech.getPhysicalRule('minExtension',
                                Layer('diff'),
                                Layer('contact'))
# return value is PhysicalRule object
pr2.value       # returns 0.10
pr2.properties  # returns { } empty dictionary, no properties
# now obtain value for minimum extension of metal1 over contact;
# note that there are horizontal and vertical extension values.
pr3 = self.tech.getPhysicalRule('minDualExtension',
                                Layer('metal1'),
                                Layer('contact'))
# return value is PhysicalRule object
pr3.value         # returns (0.02, 0.04)
pr3.value.first   # returns  0.02
pr3.value.second  # returns  0.04
pr3.properties    # returns { } empty dictionary, no properties
```

Feedback

RuleProperty

Description: Another aspect of the complexity of design rules is the need to associate properties and values with design rules in the technology file. For example, design rules used to describe contact layer spacing for vias may need to associate properties such as the number of cuts or the distance between vias. This is handled by means of design rule properties which are associated with a particular design rule in the technology file.

These RuleProperty objects are used in conjunction with the Python API Tech class. This RuleProperty object is used as part of the return value for the **getPhysicalRule**() method for the Tech class. The PhysicalRule class `properties` attribute will provide access to all of the properties which have been defined for a particular design rule in the technology file, using a Python dictionary of RuleProperty objects. Each of these RuleProperty objects has a property name and property value, which can be directly accessed.

Methods:

Note that there are no methods defined for this RuleProperty class.

Attributes:

There are two read-only attributes (or properties) defined for the RuleProperty class, described as follows:

name – the property name for this design rule property

value – the value object for this design rule property

Examples:

```
# first obtain regular spacing rule value for contact layer
spacing1 = self.tech.getPhysicalRule('minSpacing',
                                     Layer('contact'))
# now check for adjacent via spacing for contact layer
rule = self.tech.getPhysicalRule('minAdjacentViaSpacing',
                                 Layer('contact'))
# obtain properties associated with this design rule
spacing2 = rule.value
if rule.properties.has_key('distance'):
    distance = rule.properties['distance']
if rule.properties.has_key('numCuts'):
    numCuts = rule.properties['numCuts']
```

DeviceContext

Description: The DeviceContext class is used to simplify the construction of devices which require the presence of a shape on one or more layers which cover all other shapes in the device. For example, a high-voltage recognition layer can be used for the

construction of high-voltage devices. If a device context is defined in the technology file for high-voltage devices, and then activated through the Python API, the technology query functions for physical design rules will return the values which should be used for creating high-voltage devices. These special values would be different than the ordinary values which would be used for constructing non high-voltage devices. In addition, FG methods such as `fgPlace()` would automatically use these high-voltage device values.

Note that a device context is not directly created through the Python API. Rather, any device context must be defined using the special `deviceContext` syntax in the technology file; the Python API is only used to activate a device context which has already been defined in the technology file. Once a device context has been activated, then subsequent technology query functions automatically return the previously defined values for this device context from the technology file. Thus, the device context is a convenient `short-hand` to access special rules defined for different types of devices.

This device context consists of one or more layers, along with a set of design rule substitutions which define special values for devices constructed on these layers. In addition, a name is given to the device context, so that it can be activated by name through the Python API. These rule substitutions consist of two parts: the rule id for the design rule which is being substituted, and the rule id for the substituting design rule. Note that the syntax for device contexts is fully described in the section of this reference manual which describes the enhancements for the Santana technology file.

Although as many device contexts can be defined in the technology file as desired, only one such device context can be activated through the Python API at a time. Note that initially, no device contexts are activated; device contexts are only activated by the user's Python source code. This device context is activated through the use of the special Python `with` statement, in conjunction with the `DeviceContextManager` class, which is a Python context manager. Note that this `with` statement is only available with Python version 2.5; in addition, this statement requires the use of the special Python `from __future__ import with_statement` in the Python source code file. The device context is activated before this `with` block is executed, and de-activated after this `with` block has completed, even if an exception is raised during the execution of this `with` block.

Note that a device context only exists within the current DloGen design object. If the current design instantiates another PyCell, then it will be necessary to pass the current device context as a parameter to that PyCell. Otherwise, the PyCell being created will not have access to the device context, and will not use any special design rules which have been defined for the device context.

However, note that once a device context has been activated, then it will automatically be used by any of the “smart objects” within the Python API. These “smart objects” include such objects as the Contact, ContactRing or MultiPath objects. That is, whenever any of these objects are created or modified, then they will automatically use the device context which was active when the object was created.

Feedback

Creation:

Note that there is no creation method for this DeviceContext class. Instead, all device contexts are created directly in the technology file through use of the `deviceContext` syntax. The Python API is only used to activate and reference existing device contexts.

Methods:

getLayers() – returns the list of one or more layers which were defined for this device context in the technology file.

getName() – returns the name which was assigned to this device context in the technology file.

getRuleSubstitutions() – returns a Python dictionary containing the different rule substitutions which were defined for this device context in the technology file. Each key in this dictionary will be the rule id for the design rule which is being substituted, while each dictionary value will be rule id for the substituting design rule.

isEmpty() – returns true if this device context is empty, and returns False otherwise.

This device context will be empty, if there are no layers and no rule substitutions defined.

Attributes:

In addition to these methods for the DeviceContext class, there are also four attributes (or read-only properties) defined for this DeviceContext class, described as follows:

layers – list of layers which are defined for this device context

name – name assigned to this device context

ruleSubstitutions – list of rule substitutions which are defined for this device context

emptyContextName – the name of the pre-defined empty device context

Note that the first three of these attributes will return the same values as the "**getLayers**()", "**getName**()" and "**getRuleSubstitutions**()" methods.

Examples:

```
# Assume that a device context named "high_voltage"
# has been defined in the Santana technology file
space1 = self.tech.getPhysicalRule('minSpacing', Layer('gate'))
with DeviceContextManager(self.tech, "high_voltage"):
  # get minimum spacing for high-voltage devices
  space2 = getPhysicalRule('minSpacing', Layer('gate'))
  # use fgPlace() to place existing device
  fgPlace(shape1, EAST, shape2)
space3 = self.tech.getPhysicalRule('minSpacing', Layer('gate'))
# note the different minimum spacing values, since
# high-voltage design rules have different values
space1 == space2    # returns False
```

Feedback

```
space1 == space3    # returns True
space2 == space3    # returns False
```

Ruleset

Description: The **Ruleset** class is used to allow the designer to access and use different sets of physical design rules during the creation of a parameterized cell. A design rule set is simply a named set of design rules in the technology file, which can then be referenced and activated by using the name of this design rule set in the Python code for the parameterized cell. The canonical example of a design rule set is the set of "recommended" (or DFM) design rules for a process technology; these are the design rules which should be used to improve manufacturing yield, but are not required to be used to verify basic design rule correctness for a design. The use of this Ruleset class makes it very easy to activate and use such "recommended" design rules.

Note that a design rule set is not directly created through the Python API. Rather, any set of physical design rules must be defined using the special design rule set name syntax in the Santana technology file; the Python API is only used to activate a design rule set which has already been defined in the technology file. Once a design rule set has been activated, then subsequent technology query functions automatically use the design rules contained in this activated design rule set. For example, the **fgPlace**() `smart place` placement function will use the design rules contained in the activated design rule set to determine placement location values.

This design rule set consists of a set of one or more physical design rules, along with a name defined for this design rule set, so that it can be activated by name through the Python API. These physical design rules specified in a design rule set are exactly the same as any other physical design rule; they are simply defined as belonging to a named design rule set. In addition, the name for this design rule set should be unique within the Santana technology file, so that each design rule set can be uniquely activated by name. Note that the syntax for design rule sets is fully described in the section of this reference manual concerning enhancements for the technology file.

Note that these design rule sets can have a hierarchical structure, so that one design rule set can be based upon the contents of another design rule set. In the case of hierarchical design rule sets, design rule ids are used to identify which version of a design rule should be used. Physical design rules having the same design rule identifier will be replaced with the design rule having the same identifier at the highest level of the hierarchy. Any design rules which do not have a corresponding design rule identifier at the highest level of the hierarchy will be inherited as a new design rule.

Although as many design rule sets can be defined in the technology file as desired, only one such design rule set can be activated through the Python API at a time. Note that the default design rule set is named using the value of the Ruleset.defaultName class attribute; the value of this class attribute is currently set to `default`. This default design rule set is the design rule set which is active when no design rule set has been explicitly

activated. Design rule sets are only activated by the user's Python source code. This device context is activated through the use of the special Python `with` statement, in conjunction with the `RulesetManager` class, which is a Python rule set manager. Note that this `with` statement is only available with Python version 2.5; in addition, this statement requires the use of the special Python `from __future__ import with_statement` in the Python source code file. The design rule set is activated before this `with` block is executed, and de-activated after this `with` block has completed, even if an exception is raised during the execution of this `with` block.

Note that a design rule set only exists within the current DloGen design object. If the current design instantiates another PyCell, then it will be necessary to pass the current design rule set as a parameter to that PyCell. Otherwise, the PyCell being created will not have access to the design rule set, and will not use the different design rules which have been defined for the design rule set.

However, note that once a design rule set has been activated, then it will automatically be used by any of the "smart objects" within the Python API. These `smart objects` include such objects as the Contact, ContactRing or MultiPath objects. That is, whenever any of these objects are created or modified, then they will automatically use the design rule set which was active when the object was created.

Creation:

Note that there is no creation method for this Ruleset class. Instead, all design rule sets are created directly in the technology file through use of the design rule set name syntax. The Python API is only used to activate and reference existing physical design rule sets.

Methods:

getAncestor() – returns the ancestor design rule set for this design rule set. If this design rule set is not a hierarchical rule set, then None will be returned.

getName() – returns the name which was assigned to this design rule set.

Attributes:

In addition to these methods for the Ruleset class, there are also three attributes (or read-only properties) defined for this Ruleset class, described as follows:

name – name assigned to this design rule set

ancestor – ancestor design rule set for this design rule set

defaultName – name used by the default design rule set

Note that the first two of these attributes will return the same values as the “**getName**()” and “**getAncestors**()” methods.

Feedback

Examples:

```
# assume that design rule set named "recommended"
# has been defined in the Santana technology file
space1 = self.tech.getPhysicalRule('minSpacing', Layer('poly1'))
with RulesetManager(self.tech, "recommended"):
  # get minimum spacing for recommended design rules
  space2 = getPhysicalRule('minSpacing', Layer('poly1'))
  # use fgPlace() to place existing device
  fgPlace(shape2, EAST, shape1)#
space3 = self.tech.getPhysicalRule('minSpacing', Layer('poly1'))
## note the different minimum spacing values, since
# recommended design rules have different values
space1 == space2    # returns False
space1 == space3    # returns True
space2 == space3    # returns False
```

The technology class contains all of the technology information which is in the technology file which is currently used by the Santana system. The information in the Santana technology file includes mask layer data, physical design rules, electrical design rules, as well as specific parameterized values for oxides and MOSFET transistors. All of this technology information can then be used in the construction of parameterized cells.

Tech

Description: The Tech class is used to store all of the information contained in the technology file which is used by the Santana system. This technology information includes data about layers and layer relationships, as well as physical design rules and electrical design rules. In addition, the Santana technology file contains specific parameterized information about oxides and MOSFET transistors, which can then be used in the construction of parameterized cells. This technology file information is stored in a Santana technology library, which is implemented as an OpenAccess library. This technology library contains all of the information from the Santana.tech technology file. (Note that this technology library also contains the technology-dependent Contact DLO; this Contact DLO is shipped with the technology library to make it easier for the designer to develop more portable parameterized cell designs).

Note that the physical design rules specified in the Santana technology file can be quite general, in order to represent the complexity of design rules used in newer process technologies. Thus, instead of simply representing the value for a physical design rule as a single floating-point number, it is necessary to use a PhysicalRule object to represent the design rule value.

Also note that these physical design rules can also be conditional design rules, where the value for the design rule depends upon the values of one or more parameters. These conditional rules can be accessed by simply passing the required parameter names and/

or values to the **conditionalRuleExists**(), **getPhysicalRule**() and **physicalRuleExists**() methods for this Tech class.

As a convenience for PyCell development, there are a number of special layers and purposes which are pre-defined for the Santana technology file. Cadence tools such as Virtuoso make use of a number of pre-defined layers and purposes which are defined by default in the Cadence environment. In addition, OpenAccess also provides eleven system-reserved purposes. Instead of requiring all of these layers and purposes to be defined in each Santana technology file, these special system-reserved layers and purposes are automatically pre-defined in the Santana development environment.

(see the `Santana Technology File` reference manual for a complete description of these special pre-defined layers and purposes).

Creation:

Note that there is no special creation method for this Tech object. It is automatically created and associated with the Dlo/DloGen, Lib and ParamSpecArray class objects, when these objects are created. The technology library which is used to create this Tech object is the technology library attached to the OpenAccess library which is being used to store the PyCell design which is being created through the Python API. Note that Tech() will represent the technology object attached to the current DloGen design object. In addition, the `get()` static method for this Tech class can also be used to create a Tech object corresponding to a technology library. Since this `get()` method is a static method, it can be directly called, with no need to first construct the Tech object, as shown in the Examples section for this Tech object.

As a further convenience for PyCell development, this Tech object will automatically be created, even when the Santana technology file does not exist. In the situation where the Santana technology file does not exist, the OpenAccess technology database (tech.db) in the OpenAccess technology library will be used to construct the Tech object `on the fly`. The technology information contained in the `tech.db` file will be used to auto-generate an equivalent Santana technology file. In order to view the contents of this auto-generated Santana technology file, simply set the CNI_DUMP_AUTO_OA_SANTANA_TECH environment variable to `Yes`, and then a text file named `dump.auto.OA.Santana.tech` will be generated in the current working directory. Note that `smart objects`, such as the different Contact classes which make use of specific design rules to construct themselves can still be used in this mode. The required design rules in the auto-generated Tech object will be accessed and used by these `smart objects` when they are created.

Methods:

conditionalRuleExists(string *ruleName*, list *paramNames*)

conditionalRuleExists(string *ruleName*, Layer *layer1*, list *paramNames*)

conditionalRuleExists(string *ruleName*, Layer *layer1*, Layer *layer2*, list *paramNames*)
– returns True if the named conditional physical design rule specified by the *ruleName*

Feedback

parameter exists in the technology file, and returns False otherwise. The *paramNames* parameter is a list of parameter names which are used by the conditional physical design rule. Note that there are conditional rules which do not depend upon any layers, conditional rules which depend upon one layer (one layer rules) and conditional rules which depend upon two layers (two layer rules).

dbu2uu(int *value*, ViewType *viewType*=MASK_LAYOUT) – converts the specified database unit *value* (specified as an integer) to the corresponding user unit value, which is returned as a rounded double value. For example, 1000 database units would typically be converted to a user unit value of 1.0 microns.

dbu2uuArea(int *value*, ViewType *viewType*=MASK_LAYOUT) – converts the specified square database unit *value* (specified as an integer) to the corresponding square user unit value (returned as a double).

The *viewType* parameter in the above unit conversion methods is used to specify the type of view which is being used and can be one of the following values: MASK_LAYOUT, SCHEMATIC, SCHEMATIC_SYMBOL, NETLIST or HIER_DESIGN. Note that the conversion of units may be different for each of these types of views, and the conversion factor may be different for layout than for a schematic view. This conversion factor is defined in the technology file. For example, a MASK_LAYOUT view may have a conversion factor of 1000, where 1 micron as a user unit corresponds to a value of 1000 database units. (Note that these enumerated constant values are defined in the `cni.constants` Python module, which would need to be imported into the Python environment, if not already present).

deviceContextExists (string *name*) – returns True if the named device context specified by the *name* parameter exists in the technology file, and returns False otherwise.

electricalRuleExists(string *ruleName*)

electricalRuleExists(string *ruleName*, Layer *layer1*)

electricalRuleExists(string *ruleName*, Layer *layer1*, Layer *layer2*) – returns True if the named electrical design rule specified by the *ruleName* parameter exists in the technology file, and returns False otherwise. Note that there are electrical rules which do not depend upon any layers, physical rules which depend upon one layer (one layer rules) and physical rules which depend upon two layers (two layer rules).

get(string *techLibName*, *cnTechVersion*=None) – returns the Tech technology object corresponding to the technology library specified by the *techLibName* string parameter. If *cnTechVersion* is specified, then the specified version of the Santana technology information will be returned. If this Tech object does not already exist, it will be created.

Note that this method is defined as a static method, so that it can be called without directly constructing this Tech class object.

getActiveDeviceContext() – returns the currently active device context for this Tech object. If no device context is active, then the pre-defined empty device context is returned. Note that even though multiple device contexts can be defined and referenced for this Tech object, only one device context can be active at one time. See the description of the **DeviceContext** class earlier in this section.

getActiveRuleset() – returns the currently active rule set for this Tech object. If no rule set is active, then the default rule set is returned. Note that even though multiple rule sets can be defined and referenced for this Tech object, only one rule set can be active at one time. See the description of the **Ruleset** class earlier in this section.

getCustomViaDef() – returns a **CustomViaDef** object corresponding to the via definition name.

getCustomViaDefNames() – returns a list of names of all custom via definitions existing in this technology.

getDeviceContexts() – returns list of all device contexts which have been defined for this Tech object. This list is sorted alphabetically by the device context name, and does not include the default empty device context. Note that each of these device contexts needs to be defined in the technology file for this Tech object; see the description of the **DeviceContext** class earlier in this section.

getElectricalRule(string *ruleName*)

getElectricalRule(string *ruleName*, Layer *layer1*)

getElectricalRule(string *ruleName*, Layer *layer1*, Layer *layer2*) – returns the value for the named electrical design rule specified by the *ruleName* parameter, based upon the electrical design rules stored in the technology file. Note that there are electrical rules which do not depend upon any layers, electrical rules which depend upon one layer (one layer rules) and electrical rules which depend upon two layers (two layer rules).

If the named electrical design rule does not exist in the technology file, then a Python run-time exception is raised.

getGridResolution() – returns the manufacturing grid resolution value, in user units. This value will be the default grid resolution value defined for all layers in the technology file. This is the grid resolution value which should be used, when no layer-specific grid resolution value has been defined in the technology file. Note that any layer-specific grid resolution value can be accessed using the Layer class **getGridResolution**() method.

getIntermediateLayers(Layer *layer1*, Layer *layer2*) – returns the lists of route layers and via (or cut) layers, as a tuple of two lists. These are the intermediate layers between *layer1* and *layer2*, which are necessary to make a connection between these two layers. Each of these lists is returned as a uniform list of Layer objects.

getLayer(string *layerName*) – returns the Layer object for the layer having the *layerName* parameter layer name, with default `drawing` purpose. If there is no layer with such name defined in the technology file, an exception is thrown (see **Note**).

getLayer(unsigned int *layerNumber*) – returns the Layer object for the layer having the *layerNumber* parameter layer number, with default `drawing` purpose. If there is no such layer number defined in the technology file, an exception is thrown (see **Note**).

getLayer(string *layerName*, string *purposeName*) – returns the Layer object for the layer having the *layerName* parameter layer name and *purposeName* parameter purpose name. If there is no layer with such name defined in the technology file, or if there is no purpose with such name defined in the technology file, an exception is thrown (see **Note**).

getLayer(unsigned int *layerNumber*, unsigned int *purposeNumber*) – returns the Layer object for the layer having the *layerNumber* parameter layer number and *purposeName* parameter purpose number. If there is no such layer number defined in the technology file, or if there is no such purpose number defined in the technology file, an exception is thrown (see **Note**).

Note: Usually, only layers and purposes defined in the technology file can be used. However, there is one exception. If an instance is created which contains a shape with layer number *X* which is not defined in the technology file, then a temporary layer with name layer_<*X*> is auto-created. If an instance is created which contains shapes with layer purpose number *Y* which is not defined in the technology file, then a temporary purpose with name purpose_<*Y*> is auto-created.

getMaskColorNames() - returns a list of all mask color names defined in this technology.

getMosfetParams(string *type*, string *oxide*, string *parameter*) – returns the value for the named MOSFET parameter. The *type* parameter is a string, any of the type names defined in the MOSFET section of the technology file; sample technology files use `nmos` and `pmos` for these type names. The *oxide* parameter is a string, one of the oxide types defined in the technology file; sample technology files use 'thin' and 'thick' for these oxide types. The *parameter* string specifies the name of the parameter whose value is being queried. For sample technology files, parameters include:

minWidth – minimum gate oxide width value

maxWidth – maximum gate oxide width value

minLength – minimum poly length value

maxLength – maximum poly length value

If any of these parameter values is incorrect, a Python run-time exception is raised.

getOxideParams(string *oxide*, string *parameter*) – returns the value for the named OXIDE parameter. The *oxide* parameter is a string, one of the oxide types defined in the technology file; sample technology files use 'thin' and 'thick' for these oxide types. The

parameter parameter is a string, the name of the parameter whose value is being queried. If either of these parameter values is incorrect, then a Python run-time exception is raised.

getPhysicalRule(string *ruleName*, *params*=None)

getPhysicalRule(string *ruleName*, Layer *layer1*, *params*=None)

getPhysicalRule(string *ruleName*, Layer *layer1*, Layer *layer2*, *params*=None) – returns the value for the named physical design rule specified by the *ruleName* parameter, based upon physical design rules stored in the technology file. The optional *params* parameter is a standard Python dictionary, which specifies the parameter names and values to be used when evaluating a conditional physical design rule. If the *params* parameter is specified, and there is no conditional design rule which satisfies these parameters, then the unconditional design rule of the same name will be used. Note that there are physical rules which do not depend upon any layers, physical rules which depend upon one layer (one layer rules) and physical rules which depend upon two layers (two layer rules).

Note that the return value for this method is a PhysicalRule object, so that the returned value can be either a single floating-point number, a pair of floating-point numbers, or a list of pairs of floating-point numbers. If this returned value is a single floating-point number, then this return value can be handled as an ordinary floating-point number.

If the named physical design rule does not exist in the technology file, then a Python run-time exception is raised.

getRulesets() – returns a uniform list of all rule sets which have been defined for this Tech object. This list is sorted alphabetically by the rule set name. Note that this returned list does include the default rule set (unlike the **getDeviceContexts**() method, which does not include the default empty device context). Also note that each of these rule sets needs to be defined in the technology file for this Tech object; see the description of the **Ruleset** class earlier in this section.

getSantanaLayerNames() – returns a uniform list of the layer names which are defined in the corresponding Santana technology file. These are all of the user-defined layers which can be used when a PyCell design is created. Note that this method will not include any of the pre-defined layer names which are used by the technology file.

(See the `Santana Technology File` reference manual for a full description of these pre-defined layer names)

getSantanaPurposeNames() – returns a uniform list of the purpose names which are defined in the corresponding Santana technology file. These are all of the user-defined purpose names which can be used when a PyCell design is created. Note that this method will not include any pre-defined purpose names which are used by the technology file. (See the `Santana Technology File` reference manual for a full description of these pre-defined purpose names)

getStdViaDef() – returns a **StdViaDef** object corresponding to the via definition name.

Feedback

getStdViaDefNames() – returns a list of names of all standard via definitions existing in this technology.

getTechParams(bool *local*=False) – returns a read-only Python dictionary containing technology parameters and values stored in the corresponding OpenAccess technology database. The keys of this dictionary are the technology property names, while the values of this dictionary are the values of these technology properties, which are stored as integers, floating-point numbers or strings. In the case of application-specific technology properties, these property values will be stored as AppVal objects. Note that these technology parameters are stored in the `techParams` group in the OpenAccess technology database. In case of incremental OpenAccess technology, if *local*=True, the properties are taken only from the top technology database. By default (*local*=False), the properties from all referenced technologies are first added to the resulting dictionary. Note that properties from the top technology database will override properties with the same name from referenced technologies.

getUserUnits(ViewType *viewType*=MASK_LAYOUT) – returns the user units which have been defined for the specified view type.

getValidLayers() – returns the list of all valid LPPs.

getAllLayers() – returns the list of all LPPs.

id() – returns the identification string for the Santana technology file. This identification string is a combination of the name string, and the revision and version numbers.

name() – returns the name string for the Santana technology file.

physicalRuleExists(string *ruleName*, *params*=None)

physicalRuleExists(string *ruleName*, Layer *layer1*, *params*=None)

physicalRuleExists(string *ruleName*, Layer *layer1*, Layer *layer2*, *params*=None) – returns True if the named physical design rule specified by the *ruleName* parameter exists in the technology file, and returns False otherwise. The optional *params* parameter is a standard Python dictionary, which specifies the parameter names and values to be used when evaluating a conditional physical design rule. If the *params* parameter is specified, and there is no conditional design rule which satisfies these parameters, then a check will be made for an unconditional design rule of the same name. Note that there are physical rules which do not depend upon any layers, physical rules which depend upon one layer (one layer rules) and physical rules which depend upon two layers (two layer rules).

revision() – returns the revision number for the Santana technology file.

rulesetExists(string *name*) – returns True if the named rule set specified by the *name* parameter exists in the technology file, and returns False otherwise.

uu2dbu(double *value*, ViewType *viewType*=MASK_LAYOUT) – converts the specified user unit *value* (specified as a double) to the corresponding database unit value, which is

returned as an integer value. For example, a user unit of 1.0 microns would typically be converted to a value of 1000 database units.

uu2dbuArea(double *value*, ViewType *viewType*=MASK_LAYOUT) – converts the specified square user unit *value* (specified as a double) to the corresponding integer square database unit value.

The *viewType* parameter in the above unit conversion methods is used to specify the type of view which is being used and can be one of the following values: MASK_LAYOUT, SCHEMATIC, SCHEMATIC_SYMBOL, NETLIST or HIER_DESIGN. Note that the conversion of units may be different for each of these types of views, and the conversion factor may be different for layout than for a schematic view. This conversion factor is defined in the technology file. For example, a MASK_LAYOUT view may have a conversion factor of 1000, where 1 micron as a user unit corresponds to a value of 1000 database units. (Note that these enumerated constant values are defined in the `cni.constants` Python module, which would need to be imported into the Python environment, if not already present).

version() – returns the version as an unsigned integer for the Santana technology file.

Attributes:

In addition to these methods for the Tech class, there are also five attributes (or read-only properties) defined for this Tech class, described as follows:

activeDeviceContext – currently active device context for this Tech object

deviceContexts – list of all device contexts defined for this Tech object

activeRuleset– currently active rule set for this Tech object

rulesets – list of all rule sets defined for this Tech object

techParams – this property is structured as a Python dictionary, which contains technology parameters and values which are stored in the corresponding OpenAccess technology database. The keys of this dictionary are the technology property names, while the values of this dictionary are the values of these technology properties, which are stored as integers, floating-point numbers or strings. In the case of application-specific technology properties, these property values will be stored as AppVal objects. Note that these technology parameters are stored in the `techParams` group in the OpenAccess technology database. Getting this property is equivalent to calling **getTechParams**(*local*=True) method.

Note that the first four of the above attributes will return the same values as the "**getActiveDeviceContext**()", "**getDeviceContexts**()", "**getActiveRuleset(**)" and "**getRulesets**()" methods.

Feedback

Examples:

```
# get Tech object, by directly reading technology library file
# Tech object also exists on Dlo, ParamSpecArray and Lib objects
tech = Tech.get('cnTech_cni130')
tech.id()       # id ('Ciranova Fictional 0.13um ver.4 rev.0')
tech.name()     # returns name ('Ciranova Fictional 0.13 um')
tech.version()  # returns version number (4)
tech.revision() # returns revision number (0)
tech.getGridResolution()    # returns grid resolution (0.005)
tech.getSantanaLayerNames() # returns layer names used by Santana
# returns route and via layers between 'metal1' and 'metal5'
(routeLayers, cutLayers) = tech.getIntermediateLayers(Layer('metal1'),
 Layer('metal5'))
layer = tech.getLayer('metal1')  # returns 'metal1' layer object
# get various physical design rule values
# first use one layer rules for layer 'metal1'
# get 'metal1' minimum width and spacing values
tech.getPhysicalRule('minWidth', layer)
tech.getPhysicalRule('minWidth', Layer('metal1'))
tech.getPhysicalRule('minSpacing', layer)
tech.getPhysicalRule('minSpacing', Layer('metal1'))
# now illustrate the use of two layer rules;
# get different minimum extension rule values
# this extension design rule returns a single value
 tech.getPhysicalRule('minExtension',
                      Layer('nwell'),
                      Layer('diff'))
# this extension design rule returns a pair of values;
# note that end-of-line is the larger of these values.
pr = tech.getPhysicalRule('minDualExtension',
                          Layer('metal1'),
                          Layer('contact'))
endOfLine = max(pr.value.first, pr.value.second)
# illustrate the use of conditional design rules;
# "wide metal" design rules are used as an example.
# check to see if conditional design rule exists for "wide metal"
tech.conditionalRuleExists('minSpacing',
                          Layer('metal1'),
                          ['width'])
# now obtain spacing values for various metal1 width values
tech.getPhysicalRule('minSpacing',
                     Layer('metal1'),
                     params = {'width':1.0 })
tech.getPhysicalRule('minSpacing',
                     Layer('metal1'),
                     params = {'width':5.0 })
tech.getPhysicalRule('minSpacing',
                     Layer('metal1'),
                     params = {'width':10.0 })
# get various electrical rule values
tech.getElectricalRule('sheetRes', layer)
tech.getElectricalRule('sheetRes', Layer('metal1'))
```

Feedback

```
tech.getElectricalRule('areaCap', Layer('metal1'))
tech.getElectricalRule('areaCap', Layer('metal1'), Layer('metal2'))
tech.getElectricalRule('areaCap', Layer('poly1'), Layer('metal1'))
tech.getElectricalRule('edgeCapacitance', Layer('poly1'),
 Layer('metal1'))
# check to see if physical or electrical rules exist
tech.physicalRuleExists('minSpacing', Layer('metal1'))
tech.physicalRuleExists('minExtension',
                        Layer('nwell'),
                        Layer('diff'))
tech.physicalRuleExists('minDualExtension',
                        Layer('metal1'),
                        Layer('contact'))
tech.electricalRuleExists('sheetRes', Layer('metal1'))
# check to see if conditional physical rule exists
tech.conditionalRuleExists('minSpacing',
                           Layer('metal1'),
                           ['width'])
# check for device contexts
tech.getDeviceContexts()        # returns 'HighVoltage NMOS'
tech.getActiveDeviceContext()   # returns ''
tech.deviceContextExists('HV')  # returns False
# check for rulesets
tech.getRulesets()        # returns 'default', 'recommended'
tech.getActiveRuleset()   # returns 'default'
tech.rulesetExists('DFM') # returns False
# get Oxide voltage supply parameter values
tech.getOxideParams('thin', 'supply')
tech.getOxideParams('thick', 'supply')
# get MOSFET transistor parameter values
tech.getMosfetParams('pmos', 'thin',  'minWidth')
tech.getMosfetParams('nmos', 'thick', 'maxLength')
# Also check for technology parameters and values
# stored in the OpenAccess technology database.
# Note that to run this example, cnexp must be started
# using cni130 technology library, not technology file.
tech.techParams
```

6

CONNECTIVITY REFERENCE CLASSES

There are several reference classes based upon the connectivity class. These Hierarchical Object Reference classes are used to reference connectivity objects in the lower-level hierarchy in a hierarchical layout design.

The Python API provides three major classes for the handling of connectivity information of the lower-level hierarchy of a layout design: the NetRef, PinRef, and TermRef classes. These classes allow the designer to access information, respectively, about Net, Pin, and Term objects in any lower-level hierarchy for the design.

In order to conveniently refer to reference connectivity objects in a design, ShapeRef objects are allowed to get NetRef and PinRef objects.

BlockObjectRef

Description: The BlockObjectRef class is the base class used for all connectivity Hierarchical Object Reference classes. This class allows any connectivity object contained within the current design to be used as a reference object. Note that these reference objects can only be created when they exist within a hierarchical layout design; otherwise, they would just be connectivity objects in the design which could be directly accessed.

Creation:

Note that there is no creation method required for this BlockObjectRef class. These reference objects are created by the **ShapeRef** object, which contains the lower-level hierarchical design. This is handled through the use of the **getNetRef**() and **getPinRef()** methods for the ShapeRef class. Note that when a connectivity object is destroyed, then any associated reference object is also automatically destroyed.

Method:

getProps() – returns a Python dictionary-like object (the PropSet object), which contains any properties which have been defined for this BlockObjectRef object.

Attribute:

props – the properties and values which have been defined for this BlockObjectRef object.

Feedback

NetRef

Description: The NetRef class is derived from the base BlockObjectRef class, and would be used whenever the referenced object is a net.

Methods:

getBit(unsigned int *index*) – for a single-bit net, always returns this NetRef object.

For a bus net, returns a bus net bit by its index. For example, for a bus net with name "G[3:0]", and *index* = 0, returns the bus net bit with name "G[3]".

getName() – returns the name for this NetRef object

getNumBits() – returns the number of bits of this NetRef. This function always returns 1 for scalar nets, but it can return 1 or more for bus nets.

getPinRefs() – returns a list of PinRefs which are connected to any terminal associated with this NetRef object. All PinRef objects associated with this terminal are returned in this list.

getShapeRefs(ShapeFilter *filter*=ShapeFilter()) – returns a uniform list of the ShapeRefs which are currently associated with this NetRef. These shapes are the ones which were explicitly associated with this net, through the use of the **addShape**() method. The list of the ShapeRefs is determined using the *filter* parameter which is specified for each ShapeRef object.

getSignalType() – returns the signal type value assigned to this NetRef object. This signal type value is one of the constant enumerated values defined by the SignalType class.

getTermRef() – returns any terminal that is associated with this NetRef. If there is no terminal associated with this NetRef, then None is returned. Note that unlike OpenAccess, there can be at most one terminal associated with this NetRef object.

getViaRefs() – returns a list of all ViaRefs which are associated with this NetRef. If there are no vias associated with this NetRef, then None is returned.

isGlobal() – returns the Boolean flag value indicating that this NetRef is a global net.

Attribute:

globalNet – the value of the Boolean flag indicating that this is a global net.

TermRef

Description: The TermRef class is derived from the base BlockObjectRef class, and would be used whenever the referenced object is a term.

Feedback

Methods:

getName() – returns the name for this TermRef object

getNetRef() – returns the NetRef associated with this TermRef terminal object. If there is no associated net, then an exception is raised.

getPinRefs() – returns a uniform list of all of the PinRef objects associated with this TermRef terminal object.

getTermType() – returns the terminal type value assigned for this TermRef object. This terminal type value is one of the constant enumerated values defined by the TermType class.

PinRef

Description: The PinRef class is derived from the base BlockObjectRef class, and would be used whenever the referenced object is a pin.

Methods:

getAccessDir() – returns the access direction for this PinRef; this is the direction from which this PinRef should be accessed from the Instance containing this PinRef. The return value is a Direction, or a list of multiple Directions. Note that a NONE Direction indicates that this PinRef has no access direction. Also note that if all four primary directions are set as the access direction for this Pin, then Direction.ANY is returned by this method.

getName() – returns the name for this PinRef object

getNetRef() – returns the NetRef object which is associated with this PinRef.

getShapeRefs(ShapeFilter *filter*=ShapeFilter()) – returns a uniform list of the ShapeRefs which are currently associated with this PinRef. These shapes are the ones which were explicitly associated with this pin, through the use of the **addShape**() method. The list of the ShapeRefs is determined using the *filter* parameter which is specified for each ShapeRef object.

getTermRef () – returns the terminal associated with this PinRef object; if there is no terminal associated with this PinRef object, then an exception is raised.

7

PARAMETERIZED CELL CREATION CLASSES

Dlo

Description: The Dlo class is used to represent the actual design data for a layout design, based upon a parameterized cell. The name `Dlo` is an acronym for `Dynamic Layout Object`, which is a layout object which can be dynamically created, and resized as necessary. These dynamic operations are all performed through the Python API, using different classes and methods which have been provided for this purpose. This Dlo is used to encapsulate the oaDesign class provided by OpenAccess.

Creation:

Dlo(*libName*, *cellName*, *viewName*='layout', *viewType*=None, *mode*='r', *params*=None, HandlingStatus *unsupported*=HandlingStatus.ERROR) – opens and returns the design with the specified *libName* library name and *cellName* cell name. The *mode* parameter should be either 'r' (read), 'w' (write) or 'a' (append); 'w' should be used to create a design, and 'a' should be used to update a design. If *params* is specified, then it should be a valid ParamArray object, and the design must be a parameterized cell. In that case, a sub master for the specified *params* setting is returned in read mode, and the *mode* and *viewType* parameters are ignored. The *viewType* parameter should be either a string for the view type, or a valid ViewType object. The *unsupported* parameter specifies whether OpenAccess design constructs which are not directly supported by Synopsys will be handled as errors (HandlingStatus.ERROR), warnings (HandlingStatus.WARNING) or simply be ignored (HandlingStatus.IGNORE).

Note that Dlo objects should not be directly created for PyCell authoring. Instead, these Dlo objects will be implicitly created whenever the DloGen derived class is used to generate a DloGen design object, in the process of developing a PyCell. However, this creation method can be used to examine designs which are not PyCells.

The Python `with` statement can be used on an opened Dlo object to examine or update its contents. This Dlo() creation method should be used very carefully, and should only be used when the underlying OpenAccess design objects are well understood.

Feedback

Methods:

exists(string *dloName* **)** – returns True if the *dloName* Dlo design object exists, and False otherwise. The *dloName* is a string of the form `<libName>/<cellName>/<viewName>`. If <viewName> is not specified, then the default value `layout` will be used.

Note that this method is defined as a static method, so that it can be called without directly constructing this Dlo (or DloGen) class object.

findComp(string *name*=) – returns the physical component in the current design having this *name* in the current design. This is done by searching through all physical components in the current Dlo design. If there are no physical components having this *name* in the current Dlo design, then `None` is returned.

findCompRef(string *name*) – returns the PhysicalCompRef physical component reference object in the current design having this *name* in the current design. This reference object is created, if it does not already exist. This *name* should be a complete hierarchical name which uses the slash symbol as a hierarchy delimiter; for example, `INV1/ptr1/polyRect`, where `polyRect` is the name of a lower-level shape in the design. Note that this *name* can also refer to a member of an InstanceArray object, using the syntax "baseArrayName[rowIndex, colIndex]". If this *name* does not refer to an actual lower-level physical component in the current design, then `None` is returned. If this *name* contains any syntax errors (such as not being the name of a hierarchical component), then an exception is generated.

findPin(string *name*=) – returns the Pin object within Dlo which has the same name as the *name* parameter. If the *name* parameter is the empty string, then the first pin for this Dlo is returned. If there is no pin for this Dlo having this name, then an exception is raised.

findTerm(string *name*=) – returns the Term object within Dlo which has the same name as the *name* parameter. If the *name* parameter is the empty string, then the first terminal for this Dlo is returned. If there is no terminal for this Dlo having this name, then an exception is raised.

getBBox(ShapeFilter *filter*=ShapeFilter()) – returns the bounding box for this Dlo design object. Any layers specified in the ShapeFilter *filter* parameter are used to perform this bounding box calculation. This bounding box is calculated as the merged bounding box for each of the bounding boxes for the physical components in this Dlo design object defined on the specified layers.

getCellType() – returns one of the following supported cell types for this Dlo design object: BLOCK, BLOCK_BLACK_BOX, BLOCK_RING, CORE, CORE_SPACER, CORE_ANTENNA, CORE_WELL_TAP, CORNER, COVER, COVER_BUMP, NONE, PAD, PAD_AREA_IO, PAD_SPACER, SOFT_MACRO, or VIA.

getComps() – returns a uniform list of all components and Grouping objects which are contained in this Dlo object. Note that this list is a `snapshot` of the components and Grouping objects present when this method is called; the list is not updated as any

components are added to or deleted from the design. Instead, this method should be called to obtain a new `snapshot` of the components in this Dlo object.

getLeafComps() – returns a uniform list of all leaf-level components which are contained in this Dlo object. Note that this list is a `snapshot` of the leaf-level components present when this method is called; the list is not updated as any components are added to or deleted from the design. Instead, this method should be called to obtain a new `snapshot` of the leaf-level components in this Dlo object.

getLib() – returns the library associated with this Dlo design object. If this library is not already open, then it will be opened.

getLpps() – returns a uniform list of layers for this Dlo. If this Dlo has no layers, then an empty list is returned.

getName() – returns a string containing the library name, cell name and view name for this Dlo design object.

getParams(ParamArray *params*=None, bool *all*=True) – returns the ParamArray object which provides the explicit parameters and values which were used when this Dlo object was created. If the Boolean *all* parameter is True, then all parameters (both explicit and default) are returned. If the *params* parameter is specified, then it is set to the same values as are returned by this method.

getPins() – returns a uniform list of all of the Pin objects which are contained in this Dlo design object. Note that this list is a `snapshot` of the pins in this Dlo design object; this list is not updated as any pins are added to or removed from the design.

getProps() – returns a Python dictionary-like object (the PropSet object), which contains any properties which have been defined for this Dlo design object.

getTech() – returns the current Tech technology object associated with this Dlo design object. If there is no Technology object associated with this Dlo object when this method is called, then a default technology object will be associated with this Dlo design object.

getTermOrder() – returns a uniform list of terminal names for the current Dlo design object. The order of these terminal names in this list specifies the order in which terminal names should be specified when an instance of this Dlo design object is created, and the terminal connectivity is being specified by position. If this Dlo design object has more than one terminal and no terminal order has been defined (or it is defined incorrectly), then an exception is raised.

getTerms() – returns a uniform list of all of the Term objects which are contained in this Dlo design object. Note that this list is a `snapshot` of the terminals in this Dlo design object; this list is not updated as any terminals are added to or removed from the design.

getViewType() – returns the view type which is associated with this Dlo design object. This view type will be one of the following values: MASK_LAYOUT, SCHEMATIC,

SCHEMATIC_SYMBOL, NETLIST or HIER_DESIGN. (also see view type description in the documentation for the **Tech** class).

isSubMaster() – returns True if this Dlo design object represents an OpenAccess SubMaster design for a parameterized cell, and False otherwise.

isSuperMaster() – returns True if this Dlo design object represents an OpenAccess SuperMaster design for a parameterized cell, and False otherwise.

save() – saves this Dlo design using the library name, cell name and view name that were used when this Dlo object was first created.

saveAs(string *LibName*, string *cellName*, string *viewName* =) – saves this Dlo design using the *LibName*, *cellName* and *viewName* strings passed as parameters to this method. If the optional *viewName* parameter is not specified (or is an empty string), then the default view name for the Dlo object will be used. (This is currently set to the string 'layout').

setCustomLocation(Location *loc*, Point *point*) – sets a global custom location *loc* with a position specified by the *point* parameter. This location is over the cell object.

setTermOrder(StringList *terms*) – uses a uniform list of terminal names to indicate the order in which terminal names should be specified when an instance of this Dlo design object is created, and the terminal connectivity is being specified by position. Each name in this *terms* uniform list should be the name of a terminal for this Dlo design master object; otherwise, an exception is raised. If all of the terminal names for this Dlo design object are not specified in this *terms* parameter, then an exception is raised.

Attributes:

In addition to these methods for the Dlo class, there are also two attributes (or properties) defined for this Dlo class, described as follows:

props – the properties and values which have been defined for this Dlo design object

tech – the current technology object for this Dlo design object

Note that this **props** attribute value will be the same value as returned by the “**getProps**()” method, and that this **tech** attribute value will be the same value as returned by the “**getTech**()” method.

DloGen

Description: The DloGen class is the base class for all types of DLO generators. Any DLO generator would be derived from this base class. Note that this DloGen class is an abstract base class, where any implementation must be derived from this base class, and the base class by itself is not realized.

Note that any DLO generator class which is derived from this DloGen base class must directly implement the following five virtual methods, which will override the corresponding virtual methods defined for this DloGen class. These five virtual methods, are the methods which actually create the layout design for the parameterized cell.

defineParamSpecs(ParamSpecArray *specs*)

defines the parameters, including default values and constraints, for this PyCell

setupParams(ParamArray *params*)

extracts the value for the parameters specified by the user for this PyCell

genTopology()

generates the netlist topology for this layout design

sizeDevices()

properly sizes the devices (eg: length, width) before generating layout for this design

genLayout() generates the actual physical layout for this layout design

Note that whenever a DLO generator is created, by deriving from this DloGen base class, then the DLO generator will inherit methods from this DloGen class, and also the methods defined for the Dlo class, from which the DloGen class is itself derived.

Virtual Methods:

The following five methods are virtual methods, which must be provided by the class which is derived from this base DloGen class, in order to override the default versions provided by the base DloGen class. The following are concise descriptions of the functionality which will need to be provided by the PyCell author in the derived class.

defineParamSpecs(ParamSpecArray *specs*) – this is a virtual method, which should be provided to override the default version provided by the base DloGen class. This method would involve defining the specification for the parameters of this PyCell DloGen based object. This parameter specification would include parameter names, default values, and definitions of any constraints concerning the allowable values for the parameter. These parameter specifications are used when generating a master for this DloGen object. Note that this method must be defined using the `@classmethod` or `@staticmethod` Python descriptor syntax. It is suggested that the class method descriptor be used, since this approach makes it easier for other classes to be derived from this DloGen derived class.

setupParams(ParamArray *params*) – this is a virtual method, which should be provided to override the default version provided by the base DloGen class. This method would involve reading and processing the parameter values for this PyCell DloGen based object. All process specific decisions would be made in this method.

genTopology() – this is a virtual method, which should be provided to override the default version provided by the base DloGen class. This method would generate the topology for the layout design which will be generated by this PyCell DloGen based object. Note that this method may be empty for a PyCell composed of a single device.

sizeDevices() – this is a virtual method, which should be provided to override the default version provided by the base DloGen class. This method would size the devices (produced by the **genTopology**() method) which make up the layout design which will be generated by this PyCell DloGen based object. Note that this method may be empty for a PyCell composed of a single device.

genLayout() – this is a virtual method, which should be provided to override the default version provided by the base DloGen class. This method would generate the actual physical layout (based upon the sized devices generated by the **sizeDevices**() method) for the layout design which will be generated by this PyCell DloGen object.

Note that in the cases where the contents of any of these required methods may actually be empty (as in the case for a single device PyCell), then the method should still be defined, but with a function body simply consisting of the Python `pass` statement.

Methods:

Notice that this DloGen object has a large number of methods available. This reflects the fact that this is the base class for all PyCell design objects. In addition, several methods are made available as global functions, since they are used throughout the design process.

addPin(string *pinName*, string *termName*, Box *box*, Layer *layer*|ulist[*layers*]) – adds a pin having the name *pinName*, for the terminal having the *termName* name to the design represented by this DloGen object. Note that if the *termName* terminal does not already exist, then it will be automatically created. In addition, this method will also create the geometry for this pin on the layer (or layers) specified by the *layer* parameter, using the rectangle specified by the *box* Box parameter.

addTerm(string | ulist[string] *name*, TermType *termType* = TermType.INPUT_OUTPUT) – adds terminal (or terminals) with the name (or list of names) specified by the *name* parameter to this DloGen design object. The optional *termType* parameter is used to specify the terminal type, which can be one of the values specified by the TermType class: INPUT, OUTPUT, INPUT_OUTPUT, SWITCH, JUMPER, UNUSED, TRISTATE.

currentDloGen() – returns the current DloGen design object. This method would typically only need to be used when the designer is deriving their own classes from other classes besides this DloGen class, and needs to access technology information or various DloGen class methods within their derived class code. Otherwise, within the Python environment of a given DloGen object, the `self` variable is always set to the current DloGen design object.

Note that this method is defined as a static method, so that it can be called without directly constructing this DloGen object; for example, `DloGen.currentDloGen()`.

dbu2uu(int *value*) – converts the specified database unit *value* to the corresponding user unit value, which is returned as a rounded double value. For example, 1000 database units would typically be converted to a user unit value of 1.0 microns. (Note that this conversion factor between database units and user units is specified in the Santana technology file).

dbu2uuArea(int *value*) – converts the specified database unit areal *value* to the corresponding user unit areal value. The return value is a double value.

fgAnd(ulist[PhysicalComponent] *comps1*, ulist[PhysicalComponent] *comps2*, Layer *resultLayer*, ShapeFilter *filter1* = ShapeFilter(), ShapeFilter *filter2* = None, float *grid*=None) – performs a logical AND operation for lists of physical components *comps1* and *comps2*, by selecting those polygon areas which are in both physical components. This calculation is performed using the layers specified in the *filter1* and *filter2* ShapeFilter objects for each list of physical components. The resulting polygon shapes are generated on the *resultLayer* layer. Note that these shape filters will first be used to merge all layers of the two physical components into geometries on the single *resultLayer* layer, and then the Boolean operation will be performed on the merged geometries of the *resultLayer* for both lists of physical components. For example, if the *comps1* physical component contains diffusion and metal layers, and the *comps2* physical component contains poly and metal layers, while the *filter1* shape filter is diffusion and the *filter2* shape filter is poly, then the resulting geometry for the Boolean AND operation will be the MOS gate channel. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point. In addition, these polygon shapes are used to create a Grouping object, which is the return value for this method. If there are no polygon shapes, then this Grouping object is empty.

fgDeriveLayer(ulist[PhysicalComponent] *comps*, string *derivationScript*, Layer *layer*, ShapeFilter *filter*=ShapeFilter(), float *grid*=None) – runs the internal geometry engine to generate derived layer shapes according to the *derivationScript* parameter which contains a sequence of DRC commands to be executed, while the *layer* parameter specifies the layer on which to create the resulting shapes. The *filter* ShapeFilter parameter can be used to specify the exact layers which should be used during this DRC run; by default all layers are considered. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

fgNot(ulist[PhysicalComponent] *comps1*, ulist[PhysicalComponent] *comps2*, Layer *resultLayer*, ShapeFilter *filter1* = ShapeFilter(), ShapeFilter *filter2* = None, float *grid*=None) – performs a logical NOT operation for lists of physical components *comps1* and *comps2*, by selecting those polygon areas contained in the *comps1* physical component which are not contained in the *comps2* physical component. This calculation is performed using the layers specified in the *filter1* and *filter2* ShapeFilter objects for each list of physical components. The resulting polygon shapes are generated on the *resultLayer* layer. Note

that these shape filters will first be used to merge all layers of the two list of physical components into geometries on the single *resultLayer* layer, and then the NOT Boolean operation will be performed on the merged geometries of the *resultLayer* for both list of physical components. In addition, these polygon shapes are used to create a Grouping object, which is the return value for this method. If there are no polygon shapes generated, then this Grouping object will be empty. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

fgOr(ulist[PhysicalComponent] *comps1*, ulist[PhysicalComponent] *comps2*, Layer *resultLayer*, ShapeFilter *filter1* = ShapeFilter(), ShapeFilter *filter2* = None, float *grid*=None) – performs a logical OR operation for lists of physical components *comps1* and *comps*2, by selecting those polygon areas which are in either list of physical components. This calculation is performed using the layers specified in the *filter1* and *filter2* ShapeFilter objects for each list of physical components. The resulting merged polygon shapes are generated on the *resultLayer* layer. Note that these shape filters will first be used to merge all layers of the two lists of physical components into geometries on the single *resultLayer* layer, and then the OR Boolean operation will be performed on the merged geometries of the *resultLayer* for both lists of physical components. In addition, these polygon shapes are used to create a Grouping object, which is the return value for this method. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

fgSize(ulist[PhysicalComponent] *comps*, ShapeFilter *filter*, Coord *sizeValue*, Layer *resultLayer*, float *grid*=None) – expands or shrinks all polygon areas contained in this list of physical components, according to the *sizeValue* parameter value. If this *sizeValue* parameter is positive, then the polygon edges are expanded outward by that amount. If this *sizeValue* parameter is negative, then the polygon edges are shrunk inward by that amount. The resulting polygon shapes are generated on the *resultLayer* layer. In addition, these polygon shapes are used to create a Grouping object, which is the return value for this method. Note that *filter* shape filter is first used to merge all shapes from the *comps* list, and then the SIZE geometrical operation is performed on these merged geometries. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

fgXor(ulist[PhysicalComponent] *comps1*, ulist[PhysicalComponent] *comps2*, Layer *resultLayer*, ShapeFilter *filter1* = ShapeFilter(), ShapeFilter *filter2* = None, float *grid*=None) – performs a logical eXclusive-OR operation for lists of physical components *comps1* and *comps*2, by selecting those polygon areas which are in either list of physical components, but not in both lists of physical components. This operation selects all polygon areas which are common to exactly one list of physical components. This calculation is performed using the layers specified in the *filter1* and *filter2* ShapeFilter objects for each of these physical components. The resulting polygon shapes are generated on the *resultLayer* layer. Note that these shape filters will first be used to merge all layers of the two lists of physical components into geometries on the single *resultLayer* layer, and then the XOR Boolean operation will be performed on the merged geometries of the *resultLayer* for both list of physical components. In addition, these polygon shapes are used to create

a Grouping object, which is the return value for this method. If the *grid* parameter is specified, then the points of the physical components are snapped with grid value to the nearest grid point.

getComp(int *index*) – returns the component contained in this DloGen object which has the specified *index* value in the list of components maintained for this DloGen object. If this *index* value is out of range, then an exception will be generated. Note that these index values are assigned (or updated) as components are added to or removed from this DloGen design object. This index order is exactly the same as the ordering in the list of physical components returned by the **getComps**() method.

getInsts(Box *box*=None, startLevel=0, stopLevel=0) – this method returns a list of all Instance / InstanceArray / VectorInstance and/or InstanceRef / InstanceArrayRef / VectorInstanceRef objects within this DloGen design object. If *box* is specified, only shapes overlapping this box (including touching from outside) will be returned. Optionally, the start and stop levels of cell hierarchy can be specified (the default level is 0, which is also the top level). The resulting list contains only Instance / InstanceArray / VectorInstance and/or InstanceRef / InstanceArrayRef / VectorInstanceRef objects from the levels of hierarchy between startLevel and stopLevel.

getNets() – this method returns a uniform list of all of the Net objects which are contained in this DloGen design object. Note that this list will be a `snapshot` of the nets in this DloGen design object; this list will not be updated as any nets are added to or removed from the design.

getShapes(Layer *layer*=None, Box *box*=None, startLevel=0, stopLevel=0) – this method returns a list of all Shape and/or ShapeRef objects within this DloGen design object. If *layer* is specified, only shapes on this layer will be returned. If *layer* is not specified, all layers are considered. If *box* is specified, only shapes overlapping this box (including touching from outside) will be returned. Optionally, the start and stop levels of cell hierarchy can be specified (the default level is 0, which is also the top level). The resulting list contains only Shape and/or ShapeRef objects from the levels of hierarchy between startLevel and stopLevel.

makeCompName(string *prefix*) – generates a unique name for a physical component within this DloGen design object. The generated name is guaranteed to be unique within the namespace of physical components contained within the current design.

makeGrouping(string *name*=) – creates a Grouping object, which will be composed of all of the physical components currently contained within this DloGen design object. The *name* parameter is used to specify a name for this newly created Grouping object. If this *name* parameter is not specified, then a unique name will be generated. If there are conflicts with existing names, then an exception is raised. Note that the use of this method is more efficient than directly constructing the Grouping object, and then adding all of the individual physical components to it.

makeNetName(string *prefix*) – generates a unique name for a net within this DloGen design object. The generated name is guaranteed to be unique within the namespace of nets contained within the current design.

makePinName(string *prefix*) – generates a unique name for a pin within this DloGen design object. The generated name is guaranteed to be unique within the namespace of pins contained within the current design.

makePinNameSpecNum(string *prefix, unsigned int num*) – generates a unique name with user specified pin number for a pin within this DloGen design object. The generated name is guaranteed to be unique within the namespace of pins contained within the current design.

makeTermName(string *prefix*) – generates a unique name for a terminal within this DloGen design object. The generated name is guaranteed to be unique within the namespace of terminals contained within the current design.

Note that the above five methods only guarantee uniqueness of the generated names within the specified namespaces for the current design. If it is required that generated names be unique across multiple DloGen design objects, then the Unique class should be used instead of these methods.

mirrorX(Coord *yCoord*=0) – mirrors all physical components in this DloGen object about the X coordinate axis. The optional *yCoord* parameter can be used to specify a displacement for the location of the X coordinate axis.

mirrorY(Coord *xCoord*=0) – mirrors all physical components in this DloGen object about the Y coordinate axis. The optional *xCoord* parameter can be used to specify a displacement for the location of the Y coordinate axis.

moveBy(Coord *dx*, Coord *dy*) – moves all physical components in this DloGen design object by distance *dx* in the x direction and distance *dy* in the y direction.

moveTo(Point *destination*, Location *handle*=Location.CENTER_CENTER, ShapeFilter *filter*=ShapeFilter()) – moves all physical components in this DloGen design object, such that the *destination* point becomes the handle point for the bounding box for this physical component at the given *handle* location. This bounding box is determined using the *filter* ShapeFilter parameter. If this physical component does not have any geometry on the layers specified by the *filter* ShapeFilter parameter, then an exception is raised.

moveTowards(Direction *dir*, Coord *d*) – moves all physical components in this DloGen design object in the given direction *dir* by the specified distance *d*.

rotate90(Point *origin*=None) – rotates all physical components in this DloGen object by 90 degrees in a counter-clockwise direction. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

rotate180(Point *origin*=None) – rotates all physical components in this DloGen object by 180 degrees in a counter-clockwise direction. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

rotate270(Point *origin*=None) – rotates all physical components in this DloGen object by 270 degrees in a counter-clockwise direction. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

setOrigin(Point *origin*) – moves all physical components in this DloGen object, so that the specified point *origin* becomes the new origin for the current DloGen design object.

transform(Transform *trans*) – applies the transform *trans* passed as a parameter to all of the physical components in this DloGen design object.

uu2dbu(double *value*) – converts the specified user unit *value* to the corresponding database unit value, which is returned as an integer value. For example, a user unit of 1.0 microns would typically be converted to a value of 1000 database units. (Note that this conversion factor between user units and database units is specified in the Santana technology file).

uu2dbuArea(double *value*) – converts the specified user unit areal *value* to the corresponding database unit areal value. The return value is an integer capable of storing 8-byte values. Note that the conversion factor is the square of the user units to database units conversion factor.

_setEnableInstRecursion(bool *val*) – if *val* is True, then an Instance contained within this DloGen design object is allowed to instantiate a submaster containing this Instance. If *val* is False, then this recursive instantiation will not be allowed for any Instance contained within this DloGen design object. Note that by default, such instance recursion will not be allowed within this DloGen design object.

_isEnabledInstRecursion( ) – returns the current state of the Boolean flag used to control recursive instance instantiation within this DloGen design object. If the return value of this method is True, then instance recursion will be allowed; otherwise, instance recursion will not be allowed for this DloGen design object.

withNewDlo(object *callback*, string *lname*, string *cname*, string *vname*=None) – this method is largely used for batch regression testing of PyCells. The *callback* parameter is a reference to an existing Python function (or method) which is used to generate the DLO. This *callback* will be bound to the current DloGen object which is being used to populate the newly created DLO. The remaining *lname*, *cname* and *vname* parameters are used to specify the library name, cell name and view name of the newly created DLO. If the *lname* (library name) or *cname* (cell name) parameters are not specified, then an exception is raised. However, if the *vname* (view name) parameter is not specified, then the default value of `layout` will be used for the view name.

Inherited Methods: This DloGen class inherits all methods defined for the Dlo class.

Attributes:

In addition to these methods for this DloGen class, there is also an attribute (or property) defined for this DloGen class, described as follows:

AttrType – the display attribute type for this DloGen object

This **AttrType** attribute is used with the AttrDisplay class, and has the attributes CELL_NAME, CELL_TYPE, LAST_SAVED_TIME, LIB_NAME and VIEW_NAME as well as the **getMembers**() class method defined for the DloGen.AttrType class.

Examples:

```
# This is a DloGen example for a simplified Contact structure.
# It simply constructs rectangles on the different interconnect
# and via layers, and ignores cuts on any of these layers.
# There is also no metal overlap for this Contact object.
# Additional layers are also ignored for this simple example.
class simpleContact(DloGen):
    # define the parameters for this PyCell
    # The 'layer1' and 'layer2' parameters are the two layers
    # which are being connected by this Contact object.
    # Width and height parameters are used for all rectangles.
    # Note that default values are assigned for all parameters
    @classmethod
    def defineParamSpecs(cls, specs):
        specs('layer1', specs.tech.getLayer('metal1'))
        specs('layer2', specs.tech.getLayer('metal2'))
        specs('width', 2.0)
        specs('height', 1.0)
    # use variables to store parameter values entered by the user
    def setupParams(self, params):
        self.layer1 = params['layer1']
        self.layer2 = params['layer2']
        self.width = params['width']
        self.height = params['height']
    # generate the layout for this Contact PyCell.
    # This is done by creating rectangles for each of
    # the layers between the two interconnect layers.
    def genLayout(self):
        x = self.width / 2.0
        y = self.height / 2.0
        # get the intermediate layers between layer1 and layer2;
        # this information is obtained from the technology file.
        (routeLayers, viaLayers) =
        self.tech.getIntermediateLayers(self.layer1, self.layer2)
        # construct the contact rectangles on the route layers
        for layer in routeLayers:
            Rect(layer, Box(-x, -y, x, y))
        # construct the contact rectangles on the via layers
        for layer in viaLayers:
            Rect(layer, Box(-x, -y, x, y))
    def genTopology(self):
```

Feedback

```
        pass
    def sizeDevices(self):
        pass
```

VersionedDloGen

The VersionedDloGen class is used to provide different versions of the Python source code for the same PyCell. The PyCell developer can use this class to create PyCells which use different versions of the Python source code to generate different versions of the layout for the PyCell. The PyCell user can then select a particular version of the Python source code by means of a special code versioning parameter for the PyCell.

Recall that the Python source code for any PyCell is always contained in a Python class which is derived from the DloGen base class. However, in order to make use of this source code versioning capability for PyCells, the PyCell author should instead define their PyCell class as being derived from the base VersionedDloGen class. This new VersionedDloGen class is derived from the base DloGen class, and contains additional internal methods and attributes which are used for Python source code versioning purposes. For example, this VersionedDloGen class uses an internal attribute to store the name of the PyCell parameter which should be used to select a specific source code version. This code version value will then be used to select the requested version of the Python source code which should be used to generate the layout for the PyCell.

More specifically, the PyCell developer should first derive their PyCell class from the VersionedDloGen class; this PyCell class will be the default class which will be invoked whenever no code versioning parameter value is specified when the PyCell is created by the PyCell user. Then for each different code version for this PyCell, the PyCell developer should create a new PyCell class, which is derived from this first PyCell class. The naming convention which is used is that each of these derived version classes should use the class name "<baseClassName>_<version>". For example, suppose a Python package contains a basic MOSFET PyCell named "Mos", which also has two different source code versions. These classes should be defined as follows:

```
class Mos(VersionedDloGen):
  @classmethod
  def defineParamSpecs(cls, specs):
      specs('cnCodeVersion', '')
      specs('length', 2.0)
      specs('width', 1.0)
      ...
    def setupParams(self, params):
      ...
    def genLayout(self):
        # Python code for generating default layout
        ...
class Mos_v1(Mos):
  def genLayout(self):
    # Python code for version "v1" of generated layout
```

```
    ...
class Mos_v2(Mos):
  def genLayout(self):
    # Python code for version "v2" of generated layout
    ...
```

Note that only the genLayout() method should need to be provided in each of these derived version classes; only the generated layout will be different for each of the versioned PyCells. Once these versioned PyCells have been defined, then they can be invoked as follows:

```
inst0 = Instance('Mos', ParamArray())
inst1 = Instance('Mos', ParamArray(cnCodeVersion = 'v1'))
inst2 = Instance('Mos', ParamArray(cnCodeVersion = 'v2'))
```

Note that the first instance will invoke the layout code for the "Mos" class, since no cnCodeVersion parameter value is specified, so that the default version will be invoked. For the second and third instances, the `v1` and `v2` versions for the generated layout will automatically be invoked.

Methods:

There are currently no methods for this VersionedDloGen class which need to be used by the PyCell developer. It is simply sufficient to derive their versioned PyCell class from VersionedDlogen and use the built-in naming convention for versioned class names.

Lib

Description: The Lib class is used to represent the OpenAccess library which is used to read and write PyCell design objects. Note that when the Python top-level editing session is started, that all libraries which are specified in the default `lib.defs` library definition file are automatically opened. This `lib.defs` file is used by OpenAccess to store the logical names and file paths for the libraries used in a design project. The benefit of using this `lib.defs` file is that it allows all design libraries to automatically be opened at once; otherwise, the Lib **open**() method would need to be used to open each library.

Methods:

attachTechLib(string *techLibName*) – attaches the Santana technology library specified by the *techLibName* parameter to this library. If this library already has an associated technology library, then the existing attachment is replaced by this new attachment.

Note that this method needs to be used carefully, as otherwise the library may become corrupted or otherwise unusable. This method is only provided for binding a Santana technology library to this library.

close() – closes this library, so that the contents of this library become inaccessible, and any uncommitted changes are lost. Note that using this method to close this library also destroys the Lib class object.

create(string *name*, string *path*) – creates the library specified by the *name* string parameter, using the string *path* parameter as the file path for the location of this library. If this *path* parameter is not specified or is the empty string, then any matching library entry defined in the default `lib.defs` file will be used. If this library can not be created for any reason, then an exception is raised. In particular, if the *path* parameter specifies a non-empty directory, a non-existent directory, a non-writeable directory, or a file name, then an exception is raised. Note that this *path* parameter should be set to the directory path for the library, not the parent directory which contains this library directory.

Note that this method is defined as a static method, so that it can be called without directly constructing this Lib class object.

definePcell(object *cls*, string *cellName*, string *viewName* = 'layout',

string *viewType* = None, string *cellType* = None) – defines a parameterized cell in this library, by using the DloGen-derived class object specified by the *cls* parameter to create the parameterized cell. The *cellName* parameter must be specified, or an exception is raised. If the *viewName* or *viewType* parameters are not specified, then the default values (`layout` and `MASK_LAYOUT`) will be used.

This method is typically used inside the "_ _init_ _.py" Python package file which is used to define all of the Python parameterized cells in a library. This method would be called once for each parameterized cell in the library, in order to define the cell for the library. (Note that examples of this "_ _init_ _.py" file are contained in the Python source code files which are shipped for the Python API and PyCell Studio™ Tutorials).

getName() – returns the name of the library as a string.

getPath() – returns the path for the library as a string.

getProps(*cellName*=None, *viewName*=None, *readonly*=True) – returns a Python dictionary-like object, which contains any properties which have been defined for this library. If the *cellName* parameter is specified, then only properties for the cell will be returned; if the *viewName* parameter is also specified in addition to the cell name, then only properties for the cell view will be returned. If the *readonly* parameter is True, then the associated DM data file will be opened in read-only mode. If the *readonly* parameter is False, then the DM data file will be opened in update mode. If this *readonly* parameter is None, then the associated DM data file will first be opened in update mode, and if that fails, then it will be opened in read-only mode.

getTech(*cnTechVersion*=None) – returns the current Santana Tech technology object which is associated with this library. If *cnTechVersion* is specified, then the specified version of the Santana technology information will be returned. If there is no Santana technology database object associated with this library, then an exception is raised.

getTechFilePath() – returns the file path for the current Santana technology file which is attached to this library.

getTechLibName() – returns the name of the current Santana technology library which is attached to this library.

installTechFile(string *techFilePath*, bool *withCoreDlos*=True,

bool *forceBinary*=False, bool *createTechDB*=True, string *displayFilePath*=None) – populates this library with the technology information which is contained in the Santana technology file specified by the *techFilePath* parameter. The technology information in the specified technology file is copied into this library. If this library already has an associated Santana technology library, then the existing technology information in this library will be replaced by the new technology information. If *withCoreDlos* is set, then this library will also contain the `core` PyCells which are defined for a Santana technology library. These `core` PyCells are the Contact, AbutContact, ArrayInstContact and the ViaCut PyCells. If this parameter is not set, then only the technology information will be updated. If *forceBinary* is set, then the ASCII technology file specified by the *techFilePath* parameter will be used to generate a binary technology file for this library. If *generateTechDB* is set, then the OpenAccess technology database (tech.db) will be generated. Note that this tech.db file may already be generated by other tools, so that it should not be overwritten by this method. If the *displayFilePath* parameter is specified, then the LPP display assignments from the specified display file will be written into the resulting OpenAccess technology database tech.db (as application specific extensions). Note that this method needs to be used carefully, as otherwise the library may become corrupted or otherwise unusable. This method is only provided for binding a Santana technology library to this library.

loadPcells(string *cpkgID*) – loads the parameterized cells in the package specified by the *cpkgID* parameter. These parameterized cells will be loaded into this library, so that they will be available to be used from this library. If the specified code package can not be loaded for any reason, then an exception is raised.

Note that this method should be used carefully, as otherwise the library of parameterized cells may become corrupted or otherwise unusable. Note that this method is only provided for managing libraries of parameterized cells, not for authoring Python parameterized cells.

open(string *name*, string *path*='') – opens the library specified by the *name* string passed as a parameter. The *path* string parameter is an optional parameter used to specify the file path for the location of this library. If this library has already been defined in the default `lib.defs` file, then this *path* parameter is not needed, as the matching library entry in the `lib.defs` file will automatically be used. If this library is already open, then this method will simply return the existing Lib object. If this library can not be opened for any reason, then an exception is raised.

Note that this method is defined as a static method, so that it can be called without directly constructing this Lib class object.

updatePcell(string *cellName*, string *viewName* = 'layout') – updates an existing parameterized cell in this library, by using its current code and technology bindings.

The specified parameterized cell is updated, by simply re-defining the parameterized cell. Note that the *cellName* parameter must be specified, or an exception is raised. If the *viewName* parameter is not specified, then the default value `layout` will be used.

This method is useful when the parameterized cell determines default parameter values using data stored in external files or data structures, as this method can then be used to automatically update default parameter values for the parameterized cell.

Attributes:

In addition to these methods for this Lib class, there are also two attributes (or properties) defined for this Lib class, described as follows:

props – the properties and values which have been defined for this library

tech – the current technology object for this library

Note that this **tech** attribute value will be the same value as returned by the "**getTech**()" method. Also note that if there is no Technology object associated with this library when the **tech** attribute is accessed, then an exception is raised.

Examples:

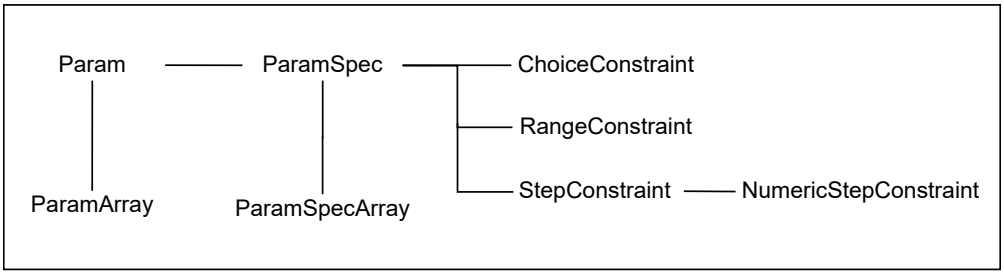
```
lib1 = Lib.create('demo', '/home/user/work/demo')
lib2 = Lib.open('demo2')  # open library from "lib.defs" file
lib1.getName()      # returns "demo"
lib1.getPath()      # returns "/home/user/work/demo"
lib2.getName()      # returns "demo2"
lib2.getPath()      # returns name like "/home/user/work/demo2"
```

PARAMETER CLASSES

An important part of the Python API is concerned with the methods which are provided to handle the different parameter values for the parameterized cell that is being generated. There are two major aspects to handling parameters. The author of the parameterized cell needs to define the different parameters for the parameterized cell, along with their default values. In addition, there should be a means to validate the values for these parameters. The user of the parameterized cell needs to be able to specify the desired values for each of these parameters, and receive feedback when the value for a parameter is not valid. The Python API provides convenient functions for the parameterized cell author to define the parameters, their default values, as well as a means to validate the parameter values entered by the user of the parameterized cell.

The different Python objects for the handling of parameters in the system are the following: Param, ParamSpec, ParamArray, ParamSpecArray, ChoiceConstraint, RangeConstraint, StepConstraint and NumericStepConstraint. The Param object is the basic parameter object, consisting of the parameter name, value and type. The ParamSpec object is the parameter specification object, consisting of the parameter, along with a specification for the default value and the validation method which should be used to validate values for this parameter. These validation methods make use of the ChoiceConstraint, RangeConstraint, StepConstraint and NumericStepConstraint objects to perform parameter value checking. These parameter and parameter specification objects are combined into the ParamArray and ParamSpecArray array objects, so that the parameters and parameter values can be handled as a single object. The parameterized cell author would use the ParamSpecArray object in order to define the parameters for their parameterized cell, along with the default values and verification routines. The user of the parameterized cell would use the ParamArray object to set the values of the parameters for the parameterized cell that is being instantiated.

The relationship between these different parameter objects is diagrammed as follows:



Notice that the Param and ParamSpec classes are only being described for purposes of conceptual understanding. These classes are not explicitly used in the Python API. Instead, the ParamArray and ParamSpecArray classes are the classes which are explicitly used to construct parameters and parameter specifications. In addition to these classes for handling parameters for parameterized cells, the ViaParam class is also provided to specify parameter values for the StdVia class (described in the section on Via classes).

These parameter objects are now described as follows:

Param

Description: The Param class is used to store the name and value for a parameter, along with the type of the parameter. The type for the parameter can be one of several different types: integer, float, double, Boolean, string, Point, Box, Layer, Direction, Topology and lists of each of these different types. The cFloat class can be used to create a C/C++ single precision floating-point parameter type, in addition to the standard double type.

In order to provide maximum interoperability between different EDA tools, it is recommended that only OpenAccess native types be used; these native types include integer, float, Boolean and string types.

Note that Param class objects are not directly created through the Python API. Instead, they will be implicitly created whenever parameters are added to a ParamArray object.

ParamArray

Description: The ParamArray class is used to store the different parameters and their values for a parameterized cell. This class can be thought of as a Python dictionary, where the keys are the parameter names and the items are the parameter values. In fact, the standard Python dictionary syntax can be used to add parameters and values to an existing ParamArray class object.

Creation:

This ParamArray can be constructed using either of the following constructor methods:

ParamArray()

ParamArray(**argv*) – in this case where lists are used, these lists are of the form “name, type, value”.

ParamArray(***kwds*) – in this case where keywords are used, these keywords are of the form: name = value.

Methods:

add(*name*, *value*)

add(*name* = *value*) – adds a parameter and its value to this ParamArray. If there is already a parameter having the same name in this ParamArray, then an exception is raised. Note that this method provides additional checking, which is not provided by the Python dictionary syntax; only use this method if such additional checking is desired.

get(string *name*) – returns the value for the parameter having the same name as the *name* parameter. If there is no parameter of this name in this ParamArray, then an exception is raised. Note that this method can also be called with a tuple of parameter names, in which case it will return the tuple of corresponding parameter values.

has_key(string *name*) – returns True if there is a parameter having the same name as the *name* parameter in this ParamArray, and returns False otherwise.

iteritems() – returns an iterator for all of the parameter and value pairs defined in this ParamArray; each call to this iterator would return one parameter and its associated value. This method is typically used in conjunction with the Python `for` statement.

iterkeys() – returns an iterator for all of the parameters in this ParamArray. This iterator method would typically be used in conjunction with the Python `for` statement.

itervalues() – returns an iterator for all of the values in this ParamArray. This iterator method would typically be used in conjunction with the Python `for` statement.

remove(string *name*) – removes the named parameter from this ParamArray. If the named parameter does not exist in this ParamArray, then an exception is raised.

reset() – resets this ParamArray, by removing all parameters from the array.

set(*name*, *value*)

set(*name* = *value*) – sets the value for the named parameter to passed value. If the parameter does not already exist in this ParamArray, then it will first be added to this ParamArray.

setFromSpecs(ParamSpecArray *specs*) – builds the parameters for this ParamArray, by copying the parameters which have been specified in the passed *specs* ParamSpecArray.

update(ParamArray *params*) – updates this ParamArray, by adding all of the parameters contained in the *params* ParamArray parameter. If the parameter already exists in this ParamArray, then the value of this parameter will be updated. If this parameter does not already exist, then this new parameter and its value will be added to this ParamArray.

Examples:

```
p = ParamArray()
# add width and length parameters to this ParamArray object
p['width'] = 1.0
p['width'] = 2.0
p.has_key('width')     # returns True
p['length'] = 1.5
p.has_key('length')    # returns True
# now get values for width and length parameters
width = p['width']
length = p['length']
(width, length) = p.get('width', 'length')
p.reset()
# can create ParamArray object and add parameters in one step
p = ParamArray(width = 1.0, length = 1.5)
```

ParamSpec

Description: The ParamSpec class is used to store the parameter, along with a specification which consists of a default value, an optional documentation string, and an optional constraint, which is used to validate the parameter value.

Note that ParamSpec class objects are not directly created through the Python API. Instead, they will be implicitly created whenever parameter specifications are added to a ParamSpecArray object.

ChoiceConstraint

Description: The ChoiceConstraint class is used to store the constraint which specifies the allowable range of values for a given parameter. The complete list of allowable values

for the parameter is used to define the choice constraint. In addition, the user can also specify an optional `failure action` which should be invoked, when the value for the parameter is not within this range of allowable values. This action can be one of three values: REJECT, ACCEPT, USE_DEFAULT. If this `failure action` parameter is not specified, then it will be REJECT by default. (Note that these enumerated constant values are defined in the `cni.constants` Python module, which would need to be imported into the Python environment, if it is not already present). This choice constraint can be specified as follows:

ChoiceConstraint(*valueList*, *action*=REJECT)

Examples:

```
ChoiceConstraint(['nmos', 'pmos'], REJECT)
ChoiceConstraint(['thin', 'thick'], USE_DEFAULT)
ChoiceConstraint([Layer('metal1'), Layer('metal2'),
                  Layer('metal3'), Layer('metal4')])
#################################################
```

RangeConstraint

Description: The RangeConstraint class is used to store the constraint which specifies the allowable range of values for a given parameter. The minimum and maximum values for the parameter are used to define this object, along with any desired resolution value. If the *resolution* value is not specified, then the *resolution* value will be determined from the database unit to user unit conversion factor for the current viewType (see the documentation for the **Tech** class). In order to avoid possible problems with floating-point arithmetic, all internal calculations will be performed with integer arithmetic, using the specified *resolution* value. In addition, the user can also specify an optional `failure action` that should be invoked, when the value for the parameter is not within this range of allowable values. This action can be one of three values: REJECT, ACCEPT, USE_DEFAULT. If this `failure action` parameter is not specified, then it will be REJECT by default. This range constraint can be specified as follows:

RangeConstraint(*low, high, resolution=None, action*=REJECT)

If the range of allowable values is only described in terms of a minimum or maximum value (ie- a `one sided` range constraint), then the special Python value `None` should be used for the unused minimum or maximum value. For example, if the range of allowable values is any number 10 or larger, then the minimum value would be 10 and the unused maximum value would be specified as `None`.

Examples:

```
RangeConstraint(1, 100, REJECT)
RangeConstraint(0.5, 1.25, USE_DEFAULT)
RangeConstraint(1.5, 100.0)
# minimum value is 10, no maximum
```

Feedback

```
RangeConstraint(10, None, REJECT)
# maximum value is 100, no minimum
RangeConstraint(None, 100, USE_DEFAULT)
```

StepConstraint

Description: The StepConstraint class is used to store the constraint which specifies the allowable values for a given parameter. This constraint is used to specify that all allowable values are an integer multiple of the specified step size. This is defined by specifying the step size, the starting value, stopping value and desired resolution value. As an example, this constraint can be used to indicate that all allowable values lie on grid point values. This Step constraint can be specified as follows:

StepConstraint(*step, start*=0*, limit*=None, *resolution*=None*, action*=REJECT)

Note that the *start* and *stop* values are not required to be integer multiples of the *step* parameter step size value. If the *resolution* value is not specified, then the *resolution* value will be determined from the database unit to user unit conversion factor for the current viewType (see documentation for the **Tech** class). In order to avoid possible problems with floating-point arithmetic, all internal calculations will be performed with integer arithmetic, using any specified *resolution* value. In addition, the user can also specify an optional `failure action` which should be invoked, when the value for the parameter is not one of the allowable values. This action can be one of three values: REJECT, ACCEPT, USE_DEFAULT. If this `failure action` parameter is not specified, then it will be REJECT by default.

Examples:

```
StepConstraint(1, 0, 10, action=REJECT)
StepConstraint(2, 0, 100, action=USE_DEFAULT)
StepConstraint(10, 1, 1000)
# ensure that all values lie on grid points
gridSize = self.tech.getGridResolution()
StepConstraint(gridSize, 0, 10)
# ensure that values lie on even multiple grid points;
# these will be 0, 2*gridSize, 4*gridSize, 6*gridSize etc.
StepConstraint(2*gridSize, 0, 10)
# ensure that values lie on odd multiple grid points;
# these will be gridSize, 3*gridSize, 5*gridSize, 7*gridSize etc.
StepConstraint(2*gridSize, gridSize, 10)
```

NumericStepConstraint

Description: The NumericStepConstraint class is used to store the constraint which specifies the allowable values for a parameter which uses string values to represent integer or floating-point values. Note that the **Numeric** class is used to represent floating-point or integer values in terms of scaled number strings. This constraint class is derived

from the **StepConstraint** class, and specifies that all allowable values are a multiple of the indicated step size, represented as a scaled string value. This is defined by specifying the step size, the starting value, stopping value, desired resolution value and scaling factor. As an example, this constraint can be used to indicate that all allowable values lie on grid point values. This NumericStep constraint can be specified as follows:

NumericStepConstraint(*step, start*=0*, limit*=None, *resolution*=None*,*

scaleFactor='u'*, action*=REJECT)

Note that the *start* and *stop* values are not required to be integer multiples of the *step* parameter step size value. If the *resolution* value is not specified, then the *resolution* value will be determined from the database unit to user unit conversion factor for the current viewType (see documentation for the **Tech** class). In order to avoid possible problems with floating-point arithmetic, all internal calculations will be performed with integer arithmetic, using any specified *resolution* value. In addition, the user can also specify an optional `failure action` which should be invoked, when the value for the parameter is not one of the allowable values. This action can be one of three values: REJECT, ACCEPT, USE_DEFAULT. If this `failure action` parameter is not specified, then it will be REJECT by default.

Examples:

```
NumericStepConstraint(1, '0', '10', action=REJECT)
NumericStepConstraint(2, '0', '100', action=USE_DEFAULT)
NumericStepConstraint(10, '1', '1000')
# ensure that all values lie on grid points
gridSize = self.tech.getGridResolution()
stepSize = Numeric(str(gridSize) + 'u')
NumericStepConstraint(stepSize, '0', '10')
# ensure that values lie on even multiple grid points
stepSize = Numeric(str(2*gridSize) + 'u')
NumericStepConstraint(stepSize, '0', '10')
# ensure that values lie on odd multiple grid points
start = Numeric(str(gridSize) + 'u')
NumericStepConstraint(stepSize, start, '10')
```

ParamSpecArray

Description: The ParamSpecArray class is used to store the different parameter specifications for the parameters defined for a parameterized cell. This class can be thought of as a Python dictionary where the keys are the parameter names and the items are the parameter default values, documentation strings and constraints.

Note that a Tech technology class object can be associated with the ParamSpecArray when the ParamSpecArray object is constructed. This allows the author to access technology specific information when defining default parameters for a parameterized cell. Thus, the parameterized cell can be developed in a technology-independent manner.

Feedback

Creation:

This ParamSpecArray can be constructed using any of the following constructor methods:

ParamSpecArray()

ParamSpecArray(Tech *tech*=None)

ParamSpecArray(Tech *tech*=None, ***kwds*) – in this case where keywords are used,

these keywords are of the form: name = (defaultValue, [docstr, [constraint]]), where the docstr documentation string and the constraint constraint definition are optional.

Note that in many cases, it will not be necessary to construct this ParamSpecArray object, as the ParamSpecArray will be passed as a parameter to the DloGen class `defineParamSpecs()` method.

Methods:

add(*name*, *value*, *docString* = None, *constraint* = None) – adds a parameter and its specification to this ParamSpecArray of parameter specifications. This includes an optional documentation string describing the parameter and its function, as well as an

optional constraint. This optional constraint would be either a ChoiceConstraint, RangeConstraint or StepConstraint. If there is already a parameter having the same name in this ParamSpecArray, then an exception is raised.

has_key(string *name*) – returns True if there is a parameter of this name in this ParamSpecArray, and returns False otherwise.

iterkeys() – returns an iterator for all of the parameters in this ParamSpecArray; each iteration would return one parameter. This iterator method would typically be used in conjunction with the Python `for` statement.

iteritems() – returns an iterator for all of the parameter and specification pairs defined in this ParamSpecArray; each iteration would return one parameter and its associated value, documentation string, and constraint. This iterator method would typically be used in conjunction with the Python `for` statement.

itervalues() – returns an iterator for all of the values defined in this ParamSpecArray; each iteration would return the value, documentation string and constraint for an associated parameter. This iterator method would typically be used in conjunction with the Python `for` statement.

remove(string *name*) – removes the named parameter and its specification from this ParamSpecArray of parameter definitions. If the named parameter does not exist in this ParamSpecArray, then an exception is raised.

verify(ParamArray *params*) – verifies the parameter values contained within this ParamArray, using the constraints stored in this ParamSpecArray. An exception will be

raised, if the parameter value is invalid, and the applicable constraint was defined using the REJECT failure action value. Otherwise, no exception is raised. Note that this method is only meant to be used for checking the parameter constraints; it assumes that the parameter types have already been checked.

Attributes:

In addition to these methods for the ParamSpecArray class, there is also an attribute (or property) defined for this ParamSpecArray class, described as follows:

tech – the technology object for this ParamSpecArray

Note that by default, this would be the current Tech technology object being used when this ParamSpecArray object was created. If there is no Tech object associated with this ParamSpecArray when the **tech** attribute is accessed, then an exception is raised.

Examples:

```
# note construction is usually not necessary, as object is passed
# as a parameter to the DloGen class "defineParamSpecs()" method
specs = ParamSpecArray()
specs.add('width', 1.0, 'transistor width',
          RangeConstraint(1.0, 100.0, USE_DEFAULT))
specs.add('tranType', 'pmos', 'MOSFET transistor type',
          ChoiceConstraint(['pmos', 'nmos']))
specs.has_key('width')        # returns True
specs.has_key('tranType')     # returns True
# can also specify parameter values based upon technology values
length = specs.tech.getMosfetParams('pmos', 'thin', 'minLength')
specs.add('length', length, 'transistor length',
          RangeConstraint(length, length*100, USE_DEFAULT))
```

The following examples show how the parameterized cell author would use the ParamSpecArray class to define parameters for a parameterized cell, and how the user of this parameterized cell would use the ParamArray class to set the values for these parameters for the parameterized cell.

Example 1 (defining parameter specifications):

In the case of a hypothetical transistor parameterized cell, the PyCell author would provide parameters for the user to be able to specify values for the transistor width and length, as well as the type of the transistor (pmos or nmos) and the type of oxide (thin or thick) to be used. This is done using the ParamSpecArray class, along with the ChoiceConstraint and RangeConstraint objects as follows:

```
specs = ParamSpecArray()
# get minimum width and length values from technology file
# and then define range constraints in terms of these values
width = specs.tech.getMosfetParams('pmos', 'thin', 'minWidth')
specs.add('width', width,  'device width',
           RangeConstraint(width, width*100, USE_DEFAULT))
```

Feedback

```
length = specs.tech.getMosfetParams('pmos', 'thin', 'minLength')
specs.add('length', length, 'device length',
          RangeConstraint(length, length*100, USE_DEFAULT))
# also provide choice constraints for transistor and oxide types
specs.add('tranType', 'pmos',  'MOSFET Type',
          ChoiceConstraint(['pmos', 'nmos']))
specs.add('oxide', 'thin', 'oxide (thin or thick)',
          ChoiceConstraint(['thin', 'thick']))
 ##########################################################################
##########################################################################
################################################
```

In the case of the transistor type parameter named `tranType`, the user must specify one of the two legal values given to the ChoiceConstraint object (`pmos` or `nmos`). However, in the case of the `length` and `width` parameters, if the user provides a value that is outside of the valid range given by the RangeConstraint object (between the minimum width and 100 times the minimum width), then the default value for that parameter will be used.

Example 2 (specifying parameter values):

In this case of a hypothetical transistor parameterized cell, the user would provide the parameter values for the transistor length and width, as well as the transistor type and oxide type. This is done using the ParamArray class as follows:

```
p = ParamArray()
# can use Python dictionary style syntax
p['length'] = 1.0
p['width'] =  0.5
p['tranType'] = 'pmos'
p['oxide'] = 'thick'
# or can use the "set()" method syntax
p.set('length', 1.0)
p.set('width', 0.5)
p.set('tranType', 'pmos')
p.set('oxide', 'thick')
# or can set parameter values in single step
p = ParamArray(length=1.0,width=.5,tranType='pmos',oxide='thick')
```

This ParamArray will then be used as an argument to the Python function which actually creates this hypothetical transistor. Note that if any parameter values are not specified in this ParamArray, then the default values for these missing parameters will be used when this hypothetical transistor is created.

ViaParam

Description: The ViaParam class is used to specify the parameter values for the StdVia class object, as described in the section on Via classes. There are a number of fixed parameter names which are provided by the ViaParam class, so that all parameter values for the standard via can be set by the ViaParam class. These parameter values can

easily be accessed, through a Python dictionary-like syntax. The keys for this ViaParam class are the eleven pre-defined parameter names, while the values are the values for each of these pre-defined parameter names. These parameter values can be accessed using either the `self['name']` or `self.name` attribute syntax. The pre-defined parameter names for this ViaParam class are the following:

cutLayer – layer which should be used as the cut layer for the standard via

cutSize – fixed values for size of the via cut used by the standard via

cutSpace – spacing values between via cuts for the standard via

implant1Ext – enclosure values for implant layer over bottom layer (layer1)

implant2Ext – enclosure values for implant layer over top layer (layer2)

layer1Ext – enclosure values for bottom layer (layer1) over the cut layer for standard via

layer1Offset – x and y offset values for the bottom layer (layer1) enclosure rectangle

layer2Ext – enclosure values for top layer (layer2) over the cut layer for standard via

layer2Offset – x and y offset values for the top layer (layer2) enclosure rectangle

numHVCuts – number of horizontal and vertical cuts for the standard via

originOffset – the offset in x and y coordinates from the origin for the standard via

Creation:

ViaParam(ViaParam *params* = None, ulist[Coord] *layer1Ext*=None, ulist[Coord] *layer2Ext*=None, ulist[Coord] *implant1Ext*=None, ulist[Coord] *implant2Ext*=None, ulist[Coord] *layer1Offset*=None, ulist[Coord] *layer2Offset*=None, ulist[Coord] *originOffset*=None, ulist[Coord] *cutSpace*=None, ulist[Coord] *cutSize*=None,

Layer *cutLayer*=None, ulist[int] *NumHVCuts*=None) – creates a ViaParam object, using the specified parameter values. Note that any parameter values specified as ulist[Coord] in this list of parameters can also be a tuple of values. For example, the enclosure parameters can either be a list of one or two coordinate values, or a tuple of one or two Coord values. If the value of the *params* parameter is specified, then this ViaParam object obtains its values from this *params* ViaParam object.

Methods:

hasDefault(string *paramName*) – returns True if the specified *paramName* parameter is set to its default value, and returns False otherwise.

items( ) – returns list of all items defined for this ViaParam object; these would be the parameter names and values for all pre-defined parameters for this ViaParam object.

Feedback

iteritems( ) – returns an iterator for all of the items defined in this ViaParam object; each iteration would return one parameter value and its associated value. This iterator method would typically be used in conjunction with the Python `for` statement.

keys( ) – returns list of all keys defined for this ViaParam object; these would be the names of all of the pre-defined parameters for this ViaParam object.

iterkeys() – returns an iterator for all of the keys defined in this ViaParam object; each iteration would return one parameter name. This iterator method would typically be used in conjunction with the Python `for` statement.

setDefault(string *paramName*) – sets the value for the specified *paramName* parameter to its default value.

Attributes:

In addition to these methods for the ViaParam class, there is also attributes (or properties) defined for the ViaParam class, described as follows:

cutLayer – cut layer for the standard via

cutSize – via cut sizes used by the standard via

cutSpace – spacing values between via cuts for the standard via

implant1Ext – enclosure values for implant layer over bottom layer (layer1)

implant2Ext – enclosure values for implant layer over top layer (layer2)

layer1Ext – enclosure values for bottom layer (layer1) over the cut layer for standard via

layer1Offset – x and y offset values for the bottom layer (layer1) enclosure rectangle

layer2Ext – enclosure values for top layer (layer2) over the cut layer for standard via

layer2Offset – x and y offset values for the top layer (layer2) enclosure rectangle

numHVCuts – number of horizontal and vertical cuts for the standard via

originOffset – offset in x and y coordinates from the origin for the standard via

Examples:

```
# first create ViaParam by specifying all values
vp1 = ViaParam(layer1Ext = [0.1, 0.2],
               layer2Ext = [0.1, 0.2],
               implant1Ext = [0.2, 0.3],
               implant2Ext = [0.2, 0.3],
               layer1Offset = [0.1, 0.1],
               layer2Offset = [0.1, 0.1],
               originOffset = [1, 2],
               cutSpace = [0.10, 0.12],
               cutSize = [0.10, 0.10],
```

```
                cutLayer = Layer('via1'),
                numHVCuts = [2, 2])
# Now create ViaParam, using ViaParam from StdVia
# which is defined and stored in the technology file
# assume that standard via "contact" exists in technology file
via1 = StdVia("contact")
defaultViaParams = via1.getParams()
vp2 = ViaParam(params = defaultViaParams)
# can also override any of these parameter values
vp3 = ViaParam(params = defaultViaParams,
                cutSize = 0.10,
                cutSpace = 0.12)
```

8

UTILITY CLASSES

There are a number of basic utility classes which have been provided in the Python API to facilitate various design creation tasks. For example, there are special utility class objects to generate unique names within a design, manage log files produced by the Santana system, and manage design libraries.

Property Classes

It is often very useful to be able to define properties which can be associated with a design object. These user-defined properties can be defined by making use of the Prop and PropSet classes to define one or more properties, which can then be directly associated with a particular object in the design. These properties can then be accessed by name, using the `getProps()` methods (or `props` attribute) which is available for the Dlo/DloGen, PhysicalComponent and Lib classes (or any classes derived from these classes).

This PropSet class allows the designer to define any number of different properties on these design objects. As such, a property is just a simple name and value pair, where the name is used to identify the property, and the value is the value for the property. Note that these properties can contain many different types of values. For example, there are properties which can contain Boolean (True/False) values, integer values, floating point values, as well as string values. Range properties can be used to specify legal ranges for numbers, as well as allowed string values. In addition, hierarchical and application specific properties can also be defined.

Prop

Description: The Prop class is just used internally to store the properties which can be defined using the PropSet class. Note that Prop class objects are not directly created through the Python API. Instead, they will be implicitly created whenever properties are added to the PropSet class object.

PropSet

Description: The PropSet class is used to store all of the user-defined properties which have been defined for a given design object. These properties can be defined and associated with the Dlo, PhysicalComponent and Lib classes, as well as any classes

which are derived from these classes. This PropSet class is designed to be very much like the standard built-in Python dictionary object, so that it is as easy as possible to add new properties to an existing PropSet object.

The types of properties which are currently provided by this PropSet class are the following: Boolean, integer, float, double, string, range, hierarchical and application-specific. Any one of these types of properties can be added to the PropSet, by simply supplying the property name and value. For example, the Python statement `props['resistance'] = 4e-6` adds a double floating-point property named 'resistance', with the value of 4e-6 to the PropSet object named `props`. Similar examples can be constructed for the case of Boolean, integer and string properties.

In the case of range properties, legal ranges of numerical values can be specified using the **RangeVal** class. This **RangeVal** class object is constructed using three numbers: the value assigned to this property, the lower bound value and the upper bound value. In a similar fashion, legal ranges of strings can be specified using the **EnumVal** class. This **EnumVal** class object is constructed using two values: the string value assigned to this property, and the list of allowed string values. Note that the **RangeVal** class is derived from the Python int or float types, while **EnumVal** is derived from the Python str type. Thus, **RangeVal** objects can be used just like integers or floating-point numbers, while **EnumVal** objects can be used just like strings. Note that if a value is assigned to one of these objects which is outside the range of legal values, then an exception is raised.

In the case of hierarchical properties, the properties have no direct values, but rather have other properties attached to them as their values. Since these other properties can also be hierarchical properties, it is then possible to construct tree-like property structures. These hierarchical property structures are very much like a hierarchical Python dictionary, where the value for a dictionary key is another dictionary. For example, a hierarchical property named `constraints` could be defined to have different types of constraint properties; each such constraint property is itself a set of floating-point properties.

In the case of application-specific properties, the PropSet class provides a great deal of flexibility. The value of an application specific property can be described using the **AppVal** class object. This **AppVal** class object is constructed using a pair of strings, where the first string is used as the application-type identifier, and the second string is used as the value's data `bytes`. In the case of a Python object, the built-in application type `PyObject` can be used to store any Python object type which can be serialized. Thus, such standard Python objects as lists and dictionaries can be stored as property values. This approach allows very general application-specific properties to be defined.

The `ILList` application type can also be used to store SKILL lists as property values.

Note that the short-hand `pyAppVal(obj)` can be used for `AppVal('PyObject', obj)`, while the short-hand `skAppVal(obj)` can be used for `AppVal('ILList', obj)`.

Feedback

Creation:

It is only necessary to explicitly construct a PropSet object when defining hierachical properties. In all other cases, the PropSet object for the Dlo design objects, Lib objects and PhysicalComponent objects are automatically created when these different objects are created. The existing PropSet object associated with these design objects is simply accessed using the `getProps()` method, or the `props` attribute for the design object.

In the case of hierarchical properties, a PropSet object is created as follows:

PropSet() – creates a new PropSet object, which is not bound to any design object, which should only be used when creating hierarchical properties. (Note that most methods defined for this PropSet class will raise an exception, if the PropSet object is not already bound to a design object).

Methods:

clear() – clears this PropSet, by removing all property definitions from this PropSet object.

empty() – returns True if this PropSet is empty (ie- contains no property definitions), and returns False otherwise.

get(string *name*, object *defaultVal* = None) – returns the value for the property with the *name* passed as a parameter. If there is no property having this name in this PropSet, then the default value specified in the defaultVal parameter will be used as the value for this property.

getProp(string *name*) – returns the value for the property with the *name* passed as a parameter. If there is no property with this name in this PropSet, then an exception is raised.

has_key(string *name*) – returns True, if this PropSet contains a property definition with the same name as the *name* parameter.

Note that the following methods operate the same as the similarly named methods for the Python built-in dictionary.

iterkeys() – returns an iterator for the property names (keys) contained in this PropSet. This iterator would be used in conjunction with the Python `for` statement.

itervalues() – returns an iterator for the property values (values) contained in this PropSet. This iterator would be used in conjunction with the Python `for` statement.

iteritems() – returns an iterator for the property name and value pairs (items) contained in this PropSet. This iterator would be used in conjunction with the Python `for` statement.

keys() – returns a list of the property names (keys) contained in this PropSet.

values() – returns a list of the property values (values) contained in this PropSet.

Feedback

items() – returns a list of the property name and value pairs (items) contained in this PropSet.

update(PropSet other) – updates this PropSet, by adding all of the property names (keys) and property values (values) contained in the PropSet specified by the *other* parameter. Note that this operation may overwrite existing property names (keys) and property values (values) in this PropSet.

Examples:

```
# Boolean, string, integer and floating point property examples
rect1 = Rect(Layer('metal1'), Box(0.0, 0.0, 10.0, 10.0))
rect1.props['hasAbutMethod'] = True
rect1.props['numSides'] = 4
rect1.props['abutMethod'] = 'auto'
# can use either float or double for floating point values
rect1.props['resistance1'] = cFloat(4e-6)
rect1.props['resistance2'] = 4e-6
# can also use range properties for numbers or strings
rect1.props['resistance'] = RangeVal(4e-6, 2e-6, 8e-6)
rect1.props['abutType'] = EnumValue('auto',[ 'auto', 'none'])
rect1.props['resistance'].value    # returns 4e-6
rect1.props['abutType'].value      # returns 'auto'
# hierarchical property example, using two levels of hierarchy
rect1.props['constraint'] = PropSet()
rect1.props['constraint']['min'] = 0.5
rect1.props['constraint']['max'] = 1.5
rect1.props['constraint']['min']   # returns 0.5
rect1.props['constraint']['max']   # returns 1.5
# application-specific property example, using Python list object
rect1.props['order1'] = AppVal('PyObject', ['left', 'right'])
rect1.props['order1']   # returns AppVal(['left', 'right'])
# application-specific property example, using 'ILList' as
# an application-specific identifier, to store Skill list
rect1.props['order2'] = AppVal('ILList', '("left", "right")'))
rect1.props['order2']   # returns AppVal('("left", "right")')
```

Python Uniform List Class

Although the Santana system is based upon the standard Python language, there is one area where it was useful to extend the basic Python language. This is the case where a Python list should only contain items all having the same type. This “uniform list” extension is the provision of a Python list type object, which only can accept elements all having the same type.

As an example, the `getSantanaLayers()` method for the Tech object will always return a uniform list of the Layer objects which are used by the Santana system. All elements in this list will be Layer objects. Another example occurs when setting up default values for parameters. It may very well be the case that a parameter should be used to specify Layer

objects to be used by the given PyCell. However, the default value for this parameter should be an empty list of Layer objects. Since the element type for an empty Python list can not be specified, this uniform list extension is used instead. The default value can be specified as an empty uniform list, which should only contain Layer object elements.

---

ulist

Description: The uniform list object is used to define a Python list, which can only contain elements having a pre-defined type. For example, lists containing only integer values or floating-point values can be defined, as well as lists containing only layout design objects of a certain type, such as Layer objects.

Methods:

This uniform list object has the same methods which currently exist for ordinary Python lists. However, these methods have been extended to check that the elements being operated upon have the expected type. If the element does not have the expected type, then an exception is raised. For example, the usual `append()` operation can be used to append an element to a list. However, in the case of this uniform list, the type of the element being appended to the list will be checked to see that it has the expected type. If the element being appended does not have the expected type, then an exception is raised.

In addition to the standard list operations briefly described below, there is also an additional property named `itemType`. This property provides the expected base type for

all elements in the uniform list. This base type can be any of the pre-defined types which are provided by the Python language (such as int, large, float, complex, list, etc), or any of the types for the layout design objects defined in the Python API (such as Layer, Contact, etc).

A uniform list can be directly created by using either of the forms `ulist[type]()` or `ulist[type](list)`. For example, to contain a uniform list which should contain only integer values, "`ulist[int]()` or `ulist[int]([1,2,3,4,5])` can be used. The first form creates an empty uniform list which should only contain integer values, while the second form creates a uniform list containing the integer values 1 through 5.

In addition, to create uniform lists, a `helper` object named `ul` has been provided in the Python API. For example, to create a uniform list which should only contain integer values can be done using either of the forms `ul(int)` or `ul[1,2,3,4,5]`. The first form creates an empty uniform list which is meant to only contain integer values, while the second form creates a uniform list containing the integer values 1 through 5. In order to get access to this `ul` object, it needs to be imported from the `cni.utils` module.

append(*element*) – adds a new element to the end of this uniform list.

Feedback

copy() – returns a copy of this uniform list

count(*element*) – returns the number of times that the given element appears in this uniform list. If the element is not in this list, the value of zero will be returned.

extend(*list*) – extends this uniform list by adding the elements in the passed list to the end of this uniform list

index(*element*) – returns the position of this element in this uniform list

insert(*index*, *element*) – inserts the element at the index position in this uniform list

pop(*index*) – returns the element at position index from this uniform list, and also removes it from the list

remove(*element*) – removes the given element from this uniform list

reverse() – reverses the order of the elements in this uniform list, modifying the list in place

sort() – sorts the elements in this uniform list, modifying the list in place

Examples:

```
u1 = ulist[Layer]()
u1.append(Layer('metal1'))
u1.append(Layer('metal2'))
u1.append(Layer('metal3'))
u1.index(Layer('metal2'))     # returns 1
u1.pop(1)                     # returns Layer('metal2')
```

PointList

Description: The PointList object is used to define and operate on lists of Point objects. Since this object is a list that should only contain Point objects, it is derived from the uniform list ulist class object described above. These PointList objects can then be used in the various DLO shape creation methods. For example, a PointList list object could be used to specify the points which would define a Polygon shape, a Path shape or a MultiPath object.

Creation:

PointList(ulist[Point] = None) – creates a PointList object, using the uniform list of Point objects to define this PointList object.

Methods:

In addition to the standard methods that are already defined for the uniform list object, and the standard Python list, this PointList class also has several methods which can be used to analyze and operate on lists of points.

Feedback

addOffsets(Coord *begin*=0, Coord *end*=0) – modifies this PointList, by adding the *begin* offset value to the first point in this PointList and adding the *end* offset value to the last point in this PointList. If the offset value is positive, then the point will be extended; if the offset value is negative, then the point will be retracted. If there are not at least two points in this PointList, then an exception is raised.

compress(bool *isClosed*=True) – compresses this PointList, by removing any extra (coincident and/or collinear) points from this PointList. The optional *isClosed* parameter is used to indicate whether this set of points is meant to represent a closed shape or not. If all points are collinear, then the first and last points will be the result of compressing this PointList. If the first and last points are coincident, then only the first point is returned.

compress_oa(bool *isClosed*=True) – Same function as compress() but is implemented by compress() of OpenAccess library. In some cases, PointList processed by compress() cannot be used to create polygon as it complains that colinear points exist. If this happens, use compress_oa() instead. Note that compress() is retained for backward compatibility consideration.

containsPoint(Point *point*, bool *incEdges*=True) – returns True, if the specified *point* is contained inside the area of this PointList, and returns False otherwise. If *incEdges* is True, then any *point* which lies on the edge of this PointList will be considered to be contained by this PointList. If *incEdges* is False, then any point which lies on the edge of this PointList will be considered not to be contained by this PointList.

copy() – returns a copy of this PointList list; all points are shared between the two lists.

deepcopy() – returns a deep copy of this PointList list; points are not shared between the two lists.

genJustifyPoints(Direction *justify*, Coord *sep*) – returns a justified point list which is constructed from this PointList. The *justify* parameter specifies the direction of the justified point list relative to this PointList, and the *sep* parameter specifies the separation between the justified point list and this PointList. If there are not at least two points in this PointList, or if there are any collinear or coincident points in this PointList, then an exception is raised. In addition, an exception will be raised if the *justify* direction is not one of the directions defined by the X-axis (EAST, WEST or EAST_WEST).

getArea() – returns the area enclosed by this PointList, when it is considered to be a closed shape. Note that this returned area will be positive when the points in this PointList are ordered counter-clockwise, and will be negative when the points are ordered clockwise. If this PointList has no rotation, then the area value returned by this method will be zero. Thus, this method can be used to determine the counter-clockwise or clockwise ordering of the points in a PointList. Note that if the PointList is self-intersecting, then the enclosed area can not be reliably used to determine the clockwise or counter-clockwise ordering for the points in the PointList.

getBBox() – returns bounding box Box object which contains all points in this PointList.

Feedback

hasExtraPoints(bool *isClosed*=True) – checks the list of points for any subset of points which are either coincident or collinear, and returns True if there are such points, and False otherwise. The optional *isClosed* parameter is used to indicate whether this set of points is meant to represent a closed shape or not.

isOrthogonal(bool *isClosed*=True) – returns True, if all the points in this PointList are orthogonal, and returns False, otherwise. The optional *isClosed* parameter is used to indicate whether this set of points is meant to represent a closed shape or not. Note that this method is useful when creating subpaths for a MultiPath object.

isSelfIntersecting(bool *isClosed*=True) – returns True, if this PointList is self-intersecting, and False otherwise. This PointList is considered to be self-intersecting, if any segment in the PointList intersects any other segment. Note that any parallel or end-point intersections are not considered to be valid self-intersections by this method. If *isClosed* is True, then this PointList will be treated as a closed shape; otherwise, it will be treated as an open Path shape.

mirrorX(Coord *yCoord*=0) – mirrors all points in this PointList, in the X direction, using the optional *yCoord* parameter as the origin to center this mirroring operation.

mirrorY(Coord *xCoord*=0) – mirrors all points in this PointList, in the Y direction, using the optional *xCoord* parameter as the origin to center this mirroring operation.

moveBy(Coord *dx*, Coord *dy*) – moves all points by the x and y coordinate values passed using the *dx* and *dy* parameter values.

moveTowards(Direction *dir*, Coord *distance*) – moves all points in this PointList in the given direction *dir* by the specified *distance*. Note that this *distance* value is specified in user units. This method is provided for consistency with the PhysicalComponent `moveTowards()` method.

moveTo(Point *destination*, Location *handle*=Location.CENTER_CENTER) – moves all points in this PointList, so that the *destination* Point becomes the handle point for the bounding box at the handle location for this PointList. This method is provided for consistency with the PhysicalComponent `moveTo()` method.

onEdge(Point *point*, bool *isClosed*=True) – returns True, if the specified *point* is on the edge of this PointList, and returns False otherwise. This is determined by checking each segment in this PointList, to determine whether the specified *point* is contained within that segment. If *isClosed* is True, then this PointList will be treated as a closed shape; otherwise, it will be treated as an open Path shape.

rotate90(Point *origin*=None) – rotates all points in this PointList counter-clockwise 90 degrees around the origin. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

Feedback

rotate180(Point *origin*=None) – rotates all points in this PointList counter-clockwise 180 degrees around the origin. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

rotate270(Point *origin*=None) – rotates all points in this PointList counter-clockwise 270 degrees around the origin. If the optional *origin* parameter is not provided, then the (0,0) point will be used as the origin for this rotation operation.

transform(Transform *trans*) – applies the transform *trans* passed as a parameter value to all of the points in this PointList.

Inherited Methods: This PointList class inherits all methods defined for the Python uniform list class.

Examples:

```
p1 = PointList([Point(0,0), Point(1.0, 1.0), Point(2.0, 2.0)])
p1.hasExtraPoints()      # returns True
p1.compress()            # removes extra points
p1.hasExtraPoints()      # returns False
p1                       # PointList([Point(0,0), Point(2,2)])
p1.isOrthogonal()        # returns False
p1.isSelfIntersecting()  # returns False
# since p1 is not self-intersecting, the area test
# can be used to check for any rotation of points
p1.getArea()             # returns 0, no rotation

p2 = PointList([Point(0,0), Point(1.0, 0), Point(1.0, 10), Point(2.0,
 1.0)])
p2.hasExtraPoints()      # returns False
p2.mirrorX()
p2.mirrorX()             # p2 back to original points
p2.rotate90()
p2.rotate270()           # p2 back to original points
p2.getBBox()             # returns Box(0,0,2,10)
p2.isSelfIntersecting()  # returns True
p2.isSelfIntersecting(isClosed=False)  # returns False
# p2 is self-intersecting as a closed shape, so area test
# can not then be reliably used to check point rotation
p2.getArea()             # returns -4.5
# construct justified point list from this PointList
p2.genJustifyPoints(justify=WEST, sep=1.0)
# add offset to starting point for this PointList
p2.addOffsets(begin=1.0)
```

Unique

Description: The Unique class is used to generate a unique name for a name within the name space of a given design object. The name generated by this object is guaranteed to be unique within a single design, provided that this naming object is used to construct all

names for the objects within a design. For example, this object can be used to generate unique names for design instances within a design, such as a name for a contact, contact ring or other named design object.

Note that this utility class method does not check the DloGen name space for any potential conflicts; that is why uniqueness of names is only guaranteed when this naming object is used to contruct names for design objects. Note that the DloGen class methods makeCompName(), makeNetName(), makePinName(), makePinNameSpecNum() and makeTermName() can be used to ensure that any generated names will be unique.

This unique name object has only a single method, the `Name()` method. There is no separate constructor for this object, and it is only necessary to call this `Name()` method to generate the unique name.

Methods:

Name(string *baseName*) – returns a unique name, which is based upon the *baseName* string passed as a parameter. If no parameter is passed to this method, then the default base name string used is `CNI_`.

Examples:

```
name1 = Unique.Name()          # returns 'CNI_1'
name2 = Unique.Name('tran')    # returns 'tran2'
name3 = Unique.Name('TRAN')    # returns 'TRAN3'
# creates a contact structure, giving it a
# unique name based upon "contact" base name
contact1 = Contact(Layer('metal1'), Layer('metal2'),
 Unique.Name('contact'))
```

NameMapper

Description: The NameMapper class is used to define a name mapping to be used to map instance names and net names, when the **clone**() method is used to clone design instances. When a design object is cloned, the cloned (or copied) object needs to have a different instance name; in addition, the connectivity of the cloned object may be different from the original design object. This NameMapper class is provided to allow the designer to easily specify how instance and net names should be generated for the cloned design object. By default, if the name mapper is not used with the **clone**() method, then auto-generated names will be used for the new instance names, and net connectivity of the original objects will be used for the net connectivity of the cloned objects. Through the use of this NameMapper class, the designer can essentially specify any desired name mapping algorithm, so that the cloning operation can be as complete as possible.

This name mapper object has only a single method, the “**map**()” method. This method performs the name mapping operation which was specified for this name mapper object.

Feedback

Creation:

NameMapper(object obj) – creates a NameMapper object, using the object which is passed as a parameter. This object can be any of the following: 1) a prefix-suffix string having the format `subPrefix/addPrefix:subSuffix/addSuffix`, 2) a Python dictionary of names and mapped names, 3) a general Python callback function, which takes the name as a parameter and returns the mapped name.

In the case of the prefix-suffix string, the `sub` strings specify the strings which should be removed from the name prefix or suffix, while the `add` strings specify the strings which should be added to the name prefix or suffix. If the `:` character is not used, then the string only applies to the name suffixes. In the case of the dictionary, note that if the name is not found in the dictionary, then this name will be returned as the mapped name. That is, no dictionary key error will be generated if the name is not in the dictionary. Note that the capability to specify a general Python callback function allows any type of complex name mapping operation to be performed.

Methods:

map(string *name*) – returns the mapped name for the *name* string passed as a parameter. This mapped name is mapped using the mapping specification which was specified when this name mapper object was created.

Examples:

```
nameMapper1 = NameMapper('p/n:Pos/Neg')
nameMapper1('pInvMos')          # returns 'nInvNeg'
dict = {'inP':'inN', 'outP':'outN'}
nameMapper2 = NameMapper(dict)
nameMapper2('outP')             # returns 'outN'
def convertName(name):
  return(name + "_p")
nameMapper3 = NameMapper(convertName)
nameMapper3('IN')               # returns 'IN_p'
```

Log

Description: The Log class is used to write information to the various log files which can be generated during a design session. These various log files contain information such as errors, warnings, general information, debug data and test data. For example, the author can generate debug information to the debug log file as a design is being created; such debug information could contain information about each design object as it is created through the Python API. In addition, if warning or error conditions occur during the generation of the design, then the appropriate warning or error messages can be generated to the warning or error log files.

The user can specify the log files which should be used when running the Santana system, by means of the following environment variables:

CNI _LOG_ANY

CNI_LOG_ERROR

CNI_LOG_WARNING

CNI_LOG_INFO

CNI_LOG_MESSAGE

CNI_LOG_DEBUG

CNI_LOG_TEST

CNI_LOG_PLACE

CNI_LOG_CONNECT

CNI_LOG_DRC

CNI_LOG_DEFAULT

These environment variables should be set to the full path name for the file that will contain the specified type of logging information. (The file name `/dev/null` can be used to redirect and ignore the logging information). Note that if the file name for the CNI_LOG_ANY environment variable is provided, then this file will contain a copy of all logging information which is generated (in addition to that generated for each individual log file). If the CNI_LOG_DEFAULT environment variable is not set, then no default logging messages will be generated. Also note that the log files for the CNI_LOG_PLACE and CNI_LOG_CONNECT environment variables are not accessible through this Python interface. That is because these are special system log files that are used to generate information about the placement and routing calls that are made by the Santana system during the generation of a design.

Note that there is no need to create this Log class object, as it is a static object, which is already created in the Python environment. In addition, all of the methods for this Log object are static. Thus, the methods for this Log object can be invoked by just using the name of this Log object; for example, Log.error(), or Log.warning(). This Log object has built-in knowledge about the file and directory structures which are used for the log file system. In particular, if the user has specified a file name for any of these log files through the use of environment variables, and if the specified file does not exist when the appropriate Log object method is called, then the log output file will automatically be created.

Methods:

Note that all of these methods will automatically provide a new line/carriage return as needed for the given output log file. Thus, it is not necessary to specify the `\n` character as part of the message string being passed to each of these methods.

Feedback

debug(string *msg*) – writes the *msg* message string to the debug log file. The debug log file contains debugging information about the design, which would be useful when the user is attempting to debug problems in the generation of the design.

error(string *msg*) – writes the *msg* message string to the error log file. The error log file contains information about error conditions which were encountered. This can be either fatal or non-fatal error conditions.

info(string *msg*) – writes the *msg* message string to the info log file. The info log file contains general information about the design, which the user should be informed about, but is not important enough to be generated in the warning or error log files.

test(string *msg*) – writes the *msg* message string to the test log file. The test log file contains testing information about the design. This test log file is generally only used when the design generation is part of an automated regression or testing suite. The test information could contain testing data as CPU run times, expected output results, etc.

warning(string *msg*) – writes the *msg* message string to the warning log file. The warning log file contains information about the design which the user should be aware of, but would generally not be so serious as to generate an error message in the error log file.

Examples:

```
Log.error("Output file %s does not exist" % fileName)
Log.warning("Not all parameters have been specified for design")
Log.info("%d design instances have been generated for design" %
 numInstances)
Log.test("All regression tests passed")
Log.debug("In Python Generator - creating contacts")
Log.debug("In Python Generator - creating bars")
Log.debug("In Python Generator - creating pins")
```

SnapType

Description: There are several different snapping types (or rounding conventions) which can be used to `snap` a point to a pre-defined rectangular grid point. This is typically done to ensure that a coordinate lies on a manufacturing grid point. There are five basic snapping types which are provided by this SnapType class: CEIL, FLOOR, ROUND, TRUNC and ROUND_CEIL. Each of these different enumerated values corresponds to a pre-defined rounding convention. The CEIL snap type rounds the fractional part of a grid unit to the next higher grid point towards positive infinity. The FLOOR snap type rounds the fractional part of a grid point to the next lower grid point, towards negative infinity. The ROUND snap type rounds the fractional part of a grid point to the nearest grid point, where any half-way values are rounded away from the zero grid point. The TRUNC snap type rounds the fractional part of a grid point to the next grid point towards the zero grid point. The ROUND_CEIL snap type rounds any fractional part to the nearest grid point,

rounding halfway cases towards positive infinity. Note that the first four snap types directly correspond to the four C language library rounding routines of the same name.

These SnapType objects are using in conjunction with the snapping methods which are provided for the Grid class, as well as the PhysicalComponent classes. These different snap types are used to specify the type of rounding which should be used when a coordinate is snapped to a grid point.

Methods:

Note that there is a snapping method for this SnapType class, as well as four static methods, which can be used to perform the four different types of snapping operations. The five enumerated values SnapType.CEIL, SnapType.FLOOR, SnapType.ROUND, SnapType.TRUNC and SnapType.ROUND_CEIL are effectively short-hand notations for the five methods SnapType.**ceil**(), SnapType.**floor**(), SnapType.**round**(), SnapType.**trunc**() and SnapType.**round_ceil**().

snap(*self*, Coord *size*, Coord *val*) – performs a snapping operation on the *val* coordinate value, using the *self* snapping type and the *size* grid size value, to snap this *val* coordinate to the proper grid point.

ceil(Coord *size*, Coord *val*) – snaps the *val* coordinate value to the proper grid point for the *size* grid size, using the CEIL snap type.

floor(Coord *size*, Coord *val*) – snaps the *val* coordinate value to the proper grid point for the *size* grid size, using the FLOOR snap type.

round(Coord *size*, Coord *val*) – snaps the *val* coordinate value to the proper grid point for the *size* grid size, using the ROUND snap type.

round_ceil(Coord *size*, Coord *val*) – snaps the *val* coordinate value to the proper grid point for the *size* grid size, using the ROUND_CEIL snap type. Note that this is the snapping approach used by the system to convert user units to database units.

trunc(Coord *size*, Coord *val*) – snaps the *val* coordinate value to the proper grid point for the *size* grid size, using the TRUNC snap type.

Examples:

```
SnapType.ceil(0.05, 2.23)          # returns  2.25
SnapType.floor(0.05, 2.23)         # returns  2.20
SnapType.round(0.05, 2.23)         # returns  2.25
SnapType.trunc(0.05, 2.23)         # returns  2.20
SnapType.round_ceil(0.05, 2.23)    # returns  2.25
SnapType.ceil(0.05, -2.23)         # returns -2.20
SnapType.floor(0.05, -2.23)        # returns -2.25
SnapType.round(0.05, -2.23)        # returns -2.25
SnapType.trunc(0.05, -2.23)        # returns -2.20
SnapType.round_ceil(0.05, -2.23)   # returns -2.25
snType = SnapType.CEIL
```

Feedback

```
snType.snap(0.05, 2.23)            # returns  2.25
snType.snap(0.05, -2.23)           # returns -2.20
```

Grid

Description: The Grid class is an abstract representation of the manufacturing grid which is used to manufacture integrated circuits. This Grid class can represent an actual or hypothetical manufacturing grid. The Grid object has a basic resolution known as the grid size value. This grid size value is specified, whenever a Grid object is created. Note that this user defined Grid class is more general than the manufacturing grid, in that it can have any X and Y resolution values, whereas an actual manufacturing mask grid only has a single resolution value, which is used for both X and Y coordinate values. It is often required to `snap` a given coordinate value to lie on a grid point. This is provided by the three different `snap()` methods for this Grid class. In addition, there are also `snap()` methods for the PhysicalComponent class. These `snap` methods are used to explicitly align layout objects to grid points; note that this `snapping` operation will not be automatically handled by other classes and methods in the Python API.

This user-defined Grid object also provides the ability to define different default snapping types, which are used to specify how a coordinate value gets `snapped` to a grid point. These are the five different snapping types defined by the SnapType class (SnapType.CEIL, SnapType.FLOOR, SnapType.ROUND, SnapType.TRUNC, SnapType.ROUND_CEIL). The default snapping type for a Grid object can be defined when the Grid object is created. In addition, all of the different `snap()` methods (for both the Grid class and the PhysicalComponent class) allow the snapping type to be specified. If the snapping type is not specified in these `snap()` methods, then the default snapping type specified when the Grid object was created will be used instead.

It is suggested that a consistent snapping type be used, in order to avoid possible cumulative truncation errors which may occur, and produce DRC spacing violations. When generating layout from left to right or top to bottom, a default snap type of SnapType.CEIL should generally work well to prevent such DRC spacing violations. Another general principle for avoiding such DRC errors when moving components and using snapping, is to use the SnapType.CEIL when the component is above the reference component, and use the SnapType.FLOOR when the component is below.

Creation:

Grid(Coord *xSize*, Coord *ySize*=None, SnapType *snapType*=SnapType.CEIL) – creates a Grid object, based upon the specified *xSize* X-coordinate resolution value. If the Y-coordinate *ySize* resolution value is not specified, then the *xSize* resolution value will be used by default. The *snapType* parameter value is used to specify the default snap type which should be used whenever the snap type is not specified in any of the different `snap()` methods. The *xSize* or *ySize* parameter value should be greater than zero; otherwise, an exception is raised. Note that these *xSize* and *ySize* coordinate values are

implicitly converted to database units before performing calculations; this conversion is performed using the current Tech object for the design.

Methods:

getSize() – returns the size or resolution value for this Grid object. This method will return the X-coordinate resolution value, without checking that the X and Y coordinate resolution values are the same; this checking is not being done for performance reasons.

getSnapType() – returns the snap type which has been defined for this Grid object. This will be one of the five enumerated type values: SnapType.CEIL, SnapType.FLOOR, SnapType.ROUND, SnapType.TRUNC or SnapType.FLOOR_CEIL.

getXSize() – returns the X-coordinate size or resolution value for this Grid object.

getYSize() – returns the Y-coordinate size or resolution value for this Grid object.

setSize(Coord *size*) – sets the size or resolution value for this Grid object; both the X-coordinate and Y-coordinate resolution values will be set to the *size* parameter value.

setXSize(Coord *size*) – sets the X-coordinate size or resolution value for this Grid object.

setYSize(Coord *size*) – sets the Y-coordinate size or resolution value for this Grid object.

setSnapType(SnapType *snapType*) – sets the snap type for this Grid object. This should be one of the five enumerated type values: SnapType.CEIL, SnapType.FLOOR, SnapType.ROUND, SnapType.TRUNC or SnapType.FLOOR_CEIL.

snap(Coord *val*, SnapType *snapType*=None, unsigned int *mult*=1) – returns a coordinate value, which is the `snapped to` grid coordinate value. The *val* parameter is the coordinate value to be `snapped`, while the *snapType* parameter is the type of snapping which should be used. The *snapType* parameter can be one of five values: SnapType.CEIL, SnapType.FLOOR, SnapType.ROUND, SnapType.TRUNC or SnapType.FLOOR_CEIL. If this *snapType* parameter is not specified, then the default snap type specified when this Grid object was created will be used as a default. The *mult* parameter specifies the multiple to be used for the X-coordinate grid value.

snapX(Coord *val*, SnapType *snapType*=None, unsigned int *mult*=1) – returns a coordinate value, which is the `snapped to` X-coordinate grid value. The *val* parameter is the coordinate value to be `snapped`, while the *snapType* parameter is the type of snapping which should be used. The *snapType* parameter can be one of five values: SnapType.CEIL, SnapType.FLOOR, SnapType.ROUND, SnapType.TRUNC SnapType.FLOOR_CEIL. If this *snapType* parameter is not specified, then the default snap type will be used as a default. The *mult* parameter specifies the multiple to be used for the X-coordinate grid value.

snapY(Coord *val*, SnapType *snapType*=None, unsigned int *mult*=1) – returns a coordinate value, which is the `snapped to` Y-coordinate grid value. The *val* parameter is the coordinate value to be `snapped`, while the *snapType* parameter is the type

of snapping which should be used. The *snapType* parameter can be one of five values: SnapType.CEIL, SnapType.FLOOR, SnapType.ROUND, SnapType.TRUNC SnapType.FLOOR_CEIL. If this *snapType* parameter is not specified, then the default snap type will be used as a default. The *mult* parameter specifies the multiple to be used for the X-coordinate grid value.

toCoord(double *val*, SnapType *snapType*=None, double *step*=0) – converts a floating-point value into the closest coordinate value, as determined by the *snapType* rounding convention and the *step* quantization value. The *val* parameter is the floating-point value to be converted into a coordinate value, while the *snapType* parameter is the type of snapping which should be used. The *snapType* parameter can be one of five values: SnapType.CEIL, SnapType.FLOOR, SnapType.ROUND, SnapType.TRUNC SnapType.FLOOR_CEIL. If this *snapType* parameter is not specified, then the default snap type will be used as a default. The *step* parameter specifies the precision for the val floating-point value; if this *step* value is non-zero, then the *val* floating-point value is `quantized` into mutiples of this *step* value. Note that the resulting coordinate value which is retuned by this method may not be on a grid point; it may be necessary to snap this coordinate value to a grid point.

Examples:

```
# Construct a Grid object using manufacturing grid value;
# this example uses the default CNI130 technology file.
grid1 = Grid(self.tech.getGridResolution())
grid1.getSnapType()                # returns SnapType.CEIL
grid1.getSize()                    # returns 0.005
grid1.getXSize()                   # returns 0.005
grid1.getYSize()                   # returns 0.005
# try various snapping operations for this Grid object
grid1.snap(0.999, SnapType.CEIL)   # returns  1.0
grid1.snap(0.999, SnapType.FLOOR)  # returns  0.995
grid1.snap(0.999, SnapType.ROUND)  # returns  1.0
grid1.snap(0.999, SnapType.TRUNC)  # returns  0.995
grid1.snap(-0.999, SnapType.CEIL)  # returns -0.995
grid1.snap(-0.999, SnapType.FLOOR) # returns -1.0
grid1.snap(-0.999, SnapType.ROUND) # returns -1.0
grid1.snap(-0.999, SnapType.TRUNC) # returns -0.995
# snap to 2X grid value, by using the optional mult parameter
grid1.snap(0.999,  mult = 2)       # returns 1.0
grid1.snap(-0.999, mult = 2)       # returns -0.99
```

Numeric

Description: The Numeric class is used to create a floating point number from a string representation, such as `10ns`. This string representation is composed of two parts: 1) a number part and 2) a scale factor part. Thus, this Numeric class can be used to represent a floating point number as a floating point number along with a scaling factor. Since this

Feedback

Numeric class is derived from the base Python float class, it can be used just like a regular floating point number in any numerical computation.

The number part of this Numeric class string representation can be any valid Python integer or floating point number; this Python floating point number can be represented using standard scientific notation, such as `1.23e-4`. The scaling factor part of this Numeric class string representation must be one of the following pre-defined scaling factor string values:

Character	Name	Multiplier
Y	Yotta	1e24
Z	Zetta	1e21
E	Exa	1e18
P	Peta	1e15
T	Tera	1e12
G	Giga	1e09
M	Mega	1e06
K or k	Kilo	1e03
‘’	No Scale Factor	1.0
%	percent	1e-2
c	centi	1e-2
m	milli	1e-3
u	micro	1e-6
n	nano	1e-9
p	pico	1e-12
f	femto	1e-15
a	atto	1e-18
z	zepto	1e-21
y	yocto	1e-24

Note that any characters after the first character in the scaling factor are simply ignored. Thus, the scaling factor `mVolt` is the same as `m`. This capability can be used to create more descriptive scaling factors.

Creation:

Numeric(int | float | string) – creates a Numeric object, based upon the specified number or string. The string must be a string of the form <number><scaleFactor>, where the <scaleFactor> is one of the pre-defined scaling factors in the above table of scaling factor strings. That is, this string representation must be composed of a number part and a scaling factor part, where the scaling factor is a pre-defined scaling factor string.

Methods:

scaleFormat(string *scaleFactor*=None) – returns the floating point number formatted using the specified *scaleFactor* scaling value. If this *scaleFactor* parameter is not specified, then the floating point number is returned using the scale factor which was used when the Numeric class object was created.

Attributes:

In addition to this single method for the Numeric class, there is also an attribute (or property) defined for this Numeric class, described as follows:

scaleFactor – the default (original) scale factor

scale_factors – list of all available scaling factors, along with their values

The **scaleFactor** is the scale factor which was used when this Numeric class object was defined. For example, if a numeric value was defined as `Numeric('10ns')`, then the default scale factor would be `n`, corresponding to the `nano` 1e-9 scaling factor. The **scale_factors** list contains all of the information from the above table of scaling factors.

Examples:

```
# first create Numeric class objects
a = Numeric(1)
b = Numeric(2.3)
c = Numeric('1.5n')
# create Numeric for 2.34 millivolts; note that all characters
# after the first are simply ignored, only "2.34m" is used.
x = Numeric('2.34mVolts')
print x                   # returns Numeric('2.34m')
# check that Numeric object is equivalent to floating point value
y = 2.34e-3
x == y                    # returns True
# can also use Numeric objects in computations
z = 2*x
print z                   # returns 0.00468
# obtain scale factor for Numeric object
x.scaleFactor             # returns 'm'
```

Feedback

```
# obtain number part of Numeric object
x/Numeric(x.scaleFactor)  # returns 2.34
# format number using different scale factors
x.scaleFormat()           # returns '2.34m', default scale factor
x.scaleFormat('')         # returns '0.00234', no scale factor
x.scaleFormat('u')        # returns '2340u', 'u' scale factor
```

cFloat

Description: The cFloat class is used to create a single precision floating point number. The resulting single precision floating point number has the same precision as a C/C++ `float` type floating-point number. This cFloat class can then be used to `wrap` a Python floating-point number as a C/C++ single precision floating point number. This capability is useful when incorporating lower-level parameterized cells which were originally developed in SKILL and have parameters which require single precision floating point number values. Thus, this cFloat class can be used when defining parameters for a PyCell, or when defining properties which are required to have single precision floating point values. Since this cFloat class is derived from the base Python float class, it can be used just like a regular floating point number in any numerical computation.

Creation:

cFloat(float) – creates a cFloat object, based upon the specified floating point number.

Methods:

There are no methods currently defined for this cFloat class

Examples:

```
# create cFloat class objects
a = cFloat(2.3)
b = cFloat(2.2/0.5)
# can also use cFloat objects in computations
z = a+b
print z                   # returns 6.7
```

AttrDict

Description: The AttrDict class is used to create a Python dictionary in which the elements of the dictionary can directly be accessed through the use of attribute syntax. That is, instead of using the usual Python dictionary access syntax `dict[key]` to access a stored dictionary value, the attribute syntax `dict.key` can be used instead. This dot attribute syntax is generally easier to use than the standard bracket operator syntax used with Python dictionaries. Thus, this AttrDict class is provided as a convenience for the PyCell developer.

Feedback

Creation:

AttrDict([arg]) – creates an AttrDict dictionary object, based upon the specified arguments, which specify the keys and values for this attribute dictionary. These arguments can be specified either using positional arguments or as a set of keyword arguments. For example, “AttrDict(a=1, b=2, c=3)” and “AttrDict([[‘a’,1], [‘b’,2], [‘c’,3]])” both create exactly the same attribute dictionary, where keys are ‘a’, ‘b’ and ‘c’, and the corresponding values are 1, 2 and 3.

Methods:

clear() – removes all items from this attribute dictionary.

get(*k*, *d*=None) – if *k* is a key in this attribute dictionary, then returns the corresponding value; otherwise, it returns the specified default value *d*. Note that this method will not raise an exception, if the key *k* is not in the attribute dictionary.

has_key(*k*) – returns True if *k* is a key in this attribute dictionary, and False otherwise.

items() – returns all of the key/value pairs defined for this attribute dictionary.

iteritems() – returns an iterator for all of the key/value pairs defined in this attribute dictionary; each call to this iterator would return one key and its associated value. This method is typically used in conjunction with the Python `for` statement.

iterkeys() – returns an iterator for all of the keys in this attribute dictionary. This iterator method would typically be used in conjunction with the Python `for` statement.

itervalues() – returns an iterator for all of the values in this attribute dictionary. This iterator method would typically be used in conjunction with the Python `for` statement.

keys() – returns all of the keys defined for this attribute dictionary.

pop(*k*, [*default*]) – if the specified key *k* is in this attribute dictionary, it is removed from this attribute dictionary, and its corresponding value is returned. Otherwise, if the specified key *k* is not in this attribute dictionary, then an exception is raised.

popitem() – removes and returns an arbitrary key/value pair from this attribute dictionary. This method can be used to destructively iterate over an attribute dictionary, and remove its items one by one. Note that if this attribute dictionary is empty, then an exception is raised.

setdefault(*k*, [*default*]) – if the specified key *k* is in this attribute dictionary, then return its value. If the key *k* is not in this attribute dictionary, then insert key *k* into this attribute dictionary. If a value is specified for *default*, then it is used as the value for key *k*; otherwise, the value for key *k* is set to None.

update(*E*) – updates this attribute dictionary with all of the key/value pairs from the specified dictionary *E*. Note that this method will overwrite existing keys, if the same key is found in both the dictionary *E* and this attribute dictionary.

Feedback

values() – returns all of the values defined for this attribute dictionary.

Examples:

```
# create attribute dictionary
ad = AttrDict(a=1, b=2, c=3, d=4, e=5)
# access elements using attribute syntax
ad.a                    # returns value 1
ad.b                    # returns value 2
ad.c                    # returns value 3
# can also access elements using standard dictionary syntax
ad['a']                 # returns value 1
ad['b']                 # returns value 2
ad['c']                 # returns value 3
ad.has_key('a')         # returns True
ad.has_key('g')         # returns False
ad.get('a')             # returns 1
ad.get('a', 2)          # returns 1, as key is in dictionary
ad.get('g', 2)          # returns 2, as key not in dictionary
ad.setdefault('f', 6)   # returns 6 (adds f:6 to dictionary)
ad.setdefault('g', 7)   # returns 7 (adds g:7 to dictionary)
ad.setdefault('a', 7)   # returns 1 (since key in dictionary)
ad.keys()               # returns all keys in dictionary
ad.values()             # returns all values in dictionary
ad.items()              # returns all key/value pairs
ad.pop('g', 2)          # removes g:7 from dictionary, returns 7
ad.keys()               # returns keys, key g removed
ad.values()             # returns values, value 7 removed
ad.items()              # returns items, item g:7 removed
# can also access all keys, values or items in dictionary,
# using the Python iterators for keys, values or items.
for key in ad.iterkeys():
    print "key = ", key
for value in ad.itervalues():
    print "value = ", value
for item in ad.iteritems():
    print "item = ", item
ad.clear()              # removes all items from dictionary
ad.keys()               # returns empty list
ad.values()             # returns empty list
ad.items()              # returns empty list
```

OrderedDict

Description: The OrderedDict class is used to create a Python dictionary in which the elements of the dictionary are kept ordered according to the order in which the dictionary keys were specified. Consequently, the methods for the OrderedDict class return keys, values or items in an ordered fashion, unlike the standard Python dictionary class. However, the OrderedDict class does provide methods which are similar to those for the built-in standard Python dictionary class. Note that this OrderedDict class is a third-party

Python Open Source implementation, which is not developed by Synopsys, but provided as part of the overall Python API product.

Creation:

OrderedDict(*init_val*=(), *strict*=False) – creates an OrderedDict dictionary object, based upon the specified *init_val* initial value ordered list of key and value pairs. The optional *strict* parameter can be used to specify that when slice assignment is performed for this OrderedDict object, then the keys for the OrderedDict object being assigned from must not contain any keys which are already being used by this OrderedDict object. If *strict* is set to True, then this check for unique keys is performed; otherwise, it is not performed.

Methods:

clear() – removes all items from this ordered dictionary.

copy() – returns a copy of this ordered dictionary, as another OrderedDict object.

index(*key*) – returns the position of the specified *key* in this ordered dictionary.

insert(*index, key, value*) – sets the specified *value* for the specified *key* at the specified position *index* in this ordered dictionary.

items() – returns a list of tuples, which represent all of the key/value pairs defined for this ordered dictionary.

iteritems() – returns an iterator for all of the key/value pairs which are defined in this ordered dictionary; each call to this iterator would return one key and its associated value. This method is typically used in conjunction with the Python `for` statement.

iterkeys() – returns an iterator for all of the keys in this ordered dictionary. This iterator method would typically be used in conjunction with the Python `for` statement.

itervalues() – returns an iterator for all of the values in this ordered dictionary. This iterator method would typically be used in conjunction with the Python `for` statement.

keys() – returns a list containing all of the keys defined for this ordered dictionary.

pop(*key*, **args*) – if the specified *key* is in this ordered dictionary, it is removed from this ordered dictionary, and its corresponding value is returned. Otherwise, if the specified *key* is not in this ordered dictionary, then an exception is raised.

popitem(*i* = -1) – removes and returns the key/value pair specified by the index *i* from this ordered dictionary. This method can be used to destructively iterate over an ordered dictionary, and remove its items one by one. Note that if this ordered dictionary is empty, or if the index *i* is not a valid position index value, then an exception is raised. By default, the position index *i* is set to -1, which refers to the last element in this ordered dictionary. Note that the standard Python dictionary **popitem**() method removes a random key/value pair, whereas this method removes the specified key/value pair.

Feedback

rename(*old_key*, *new_key*) – renames the key for a given value, without modifying the sequence order for this ordered dictionary. If the specified *new_key* key already exists in this ordered dictionary, then an exception is raised.

reverse() – reverses the sequence order for the items of this ordered dictionary.

setdefault(*key*, *defval*=None) – if the specified *key* is in this ordered dictionary, then return its value. If the specified *key* is not in this ordered dictionary, then insert *key* into this ordered dictionary. If a value is specified for *defval*, then it is used as the value for *key*; otherwise, the value for *key* is set to None.

setitems(*items*) – uses the specified *items* parameter to set all of the items for this ordered dictionary. This *items* parameter is a list of tuples, which represent all of the key/value pairs for this ordered dictionary. Note that this list of tuples is in the same format as is returned by the **items**() method.

setkeys(*keys*) – uses the specified *keys* parameter to replace all of the keys for this ordered dictionary. This *keys* parameter is a list of new keys, which should contain the same keys as already exist for this ordered dictionary. If the specified *keys* parameter contains any keys which are not already existing keys for this ordered dictionary, then an exception is raised. Note that the list of keys specified by this *keys* parameter does not need to be in the same order as the existing set of keys for this ordered dictionary; the ordering of this list of keys will determine the new ordering for this ordered dictionary.

setvalues(*values*) – uses the specified *values* parameter to replace all of the values for this ordered dictionary. This *values* parameter is a list of new values, which should be the same length as this ordered dictionary. If the specified *values* parameter is not the same length as this ordered dictionary, then an exception is raised.

sort(**args*, ***kwargs*) – sorts the key order for this ordered dictionary. Note that this method takes exactly the same arguments as the standard Python list **sort**() method.

update(*from_od*) – updates this ordered dictionary with all of the key/value pairs from the specified *from_od* ordered dictionary. In addition, this required *from_od* parameter can also specify a sequence of (key, value) pairs. If the *from_od* parameter is not an ordered dictionary or a sequence of (key, value) pairs, then an exception is raised.

values() – returns a list containing all of the values defined for this ordered dictionary.

Examples:

```
# first import OrderedDict ordered dictionary class
from odict import OrderedDict
# create ordered dictionary
od = OrderedDict(((1,'c'), (3,'b'), (2,'a')))
# can obtain all keys, values or items in ordered dictionary
od.keys()                # returns all keys in dictionary
od.values()              # returns all values in dictionary
od.items()               # returns all items in dictionary
# now add additional item to ordered dictionary
```

Feedback

```
od.setdefault(4,'d')      # returns 'd' (adds 4:'d' to dictionary)
od.setdefault(2,'e')      # returns 'a' (since key in dictionary)
od.keys()                 # returns all keys in dictionary
od.values()               # returns all values in dictionary
od.items()                # returns all key/value pairs
od.pop(4)                 # removes 4:'d', returns 'd'

od.keys()                 # returns keys, key 4 removed
od.values()               # returns values, value 'd' removed
od.items()                # returns items, item 4:'d' removed
# can also access all keys, values or items in dictionary,
# using the Python iterators for keys, values or items.
for key in od.iterkeys():
    print "key = ", key
for value in od.itervalues():
    print "value = ", value
for item in od.iteritems():
    print "item = ", item
od.clear()                # removes all items from dictionary
od.keys()                 # returns empty list
od.values()               # returns empty list
od.items()                # returns empty list
```

ParamDictSpec

Description: The ParamDictSpec class is used to specify parameters and the specifications for these parameters. This is done by creating an ordered Python dictionary in which the dictionary keys are the parameter names, and the dictionary values are the specifications for these parameters. This parameter specification consists of four elements: a Boolean flag indicating whether the parameter is required or not, the type specification for the parameter, an optional documentation string for the parameter, and an optional constraint specification for the parameter. This ParamDictSpec class is derived from the base OrderedDict class, so that the parameter names will be stored in the order in which they were added to this ParamDictSpec dictionary. Thus, the ordering provided by the ParamDictSpec class is provided as a developer convenience.

Note that the ParamDictSpec class is typically used in conjunction with the Helix `XGenInlineTemplate` constraint, which is fully described in the Helix reference manual (in the `HELIX CONSTRAINT FILE` chapter in the `Helix User Guide` reference manual).

Creation:

Since the ParamDictSpec is directly derived from the base OrderedDict class, the creation method for the ParamDict class is exactly the same as for the OrderedDict class.

Methods:

optional(*key, datatype, doc*=None, *constraint*=None) – adds an optional parameter and the associated parameter specification to this ParamDictSpec object. The parameter

name is specified by the *key* parameter, while the parameter specification is specified by the required *datatype* type specification, the optional *doc* documentation string, and the optional *constraint* parameter constraint. If the specified *key* parameter name already exists, then an exception is raised.

required(*key, datatype, doc*=None, *constraint*=None) – adds a required parameter and the associated parameter specification to this ParamDictSpec object. The parameter name is specified by the *key* parameter, while the parameter specification is specified by the required *datatype* type specification, the optional *doc* documentation string, and the optional *constraint* parameter constraint. If the specified *key* parameter name already exists, then an exception is raised.

match(*data*) – checks to see if the parameter and value specified by the *data* parameter matches the specification for the parameter in this ParaDictSpec object. The parameter and value for the *data* parameter is specified using standard Python dictionary format.

If the parameter value matches the specification, then the input *data* dictionary is returned, possibly updated with additional default parameter values. If the parameter values does not match the specification, then an exception is raised.

Examples:

```
# first import ParamDictSpec dictionary specification class
from cni.utils import ParamDictSpec
# create parameter dictionary specification
pds = ParamDictSpec()
# add required parameters to dictionary specification
pds.required('AbutDevices', bool, "")
pds.required('NwellNetName', str, "")
pds.required('Pmos', list, "")
pds.required('Nmos', list, "")
# also add optional parameter to dictionary specification
pds.optional('Alignment', str, "")
# now check to see if parameter values match specification
data = {}
data['AbutDevices'] = True
data['NwellNetName'] = 'VSS'
# check for required parameters
pds.match(data)    # exception, missing required parameters
data['Pmos'] = ['MP1', 'MP2']
pds.match(data)    # exception, still missing required parameters
data['Nmos'] = ['MN1', 'MN2']
pds.match(data)    # matches, all required parameters specified
data['Alignment'] = "top"
pds.match(data)    # matches, including optional parameters
data['AbutDevices'] = 7
pds.match(data)    # exception, wrong parameter type
```

Feedback

CDF

Description: The CDF class is used to read CDF data which is already stored as a property on a cell in a library. This CDF data should be stored as a string property using the ILList format, and the name of this property should be `cdfData`. This CDF data will be returned using an AttrDict attribute dictionary, so that the CDF data can then be accessed using standard Python attribute syntax. Note that this CDF class is only meant to be used for read-only access of the CDF data; any attempted updates will have no effect on the CDF data which is stored on the cell.

Creation:

Note that there is no creation method for this CDF class, as it is assumed that the CDF data is already stored as a property on the specified cell in the library. The required CDF object will automatically be created and used whenever the **forCell**() method is invoked.

Methods:

forCell(string *libName*, string *cellName*) – returns any stored CDF data for the specified cell in the library, using an AttrDict attribute dictionary. This allows all CDF data to be accessed using Python attribute syntax. If no CDF data has been stored on the specified cell, then an exception is raised.

Examples:

```
# first need to import CDF module
from cni.integ.cdfUtil import CDF
# assume that CDF data is already stored on cell in library
cdf = CDF.forCell('IPL_cni130', 'Nmos')
# now access CDF data using attribute syntax
cdf.keys()                  # returns all top-level keys
cdf.fieldWidth              # returns value for fieldWidth
cdf.parameters.w.defValue   # returns default value for param 'w'
```

DPL

Description: The DPL class is used to convert a SKILL Disembodied Property List (DPL) object into a Python dictionary object. This DPL data will be returned using an AttrDict attribute dictionary, so that the DPL data can then be accessed using standard Python attribute syntax. This conversion process makes it much easier to work with SKILL data which is always being stored using disembodied property lists. For example, some of the SKILL data which is stored in CDF format is stored using DPL objects.

Creation:

DPL([arg]) – creates an AttrDict attribute dictionary object, which is based upon the specified list of arguments, which should represent a SKILL disembodied property list.

More specifically, this disembodied property list should contain an arbitrary first element (usually nil in SKILL), which is then followed by the property names and values.

Methods:

Note that there are no additional methods defined for this DPL class, as once the DPL object is created, then the resulting attribute dictionary can be used to directly access the elements which are stored in the Disembodied Property List.

Examples:

```
# first need to import DPL module
from cni.integ.cdfUtil import DPL
# create DPL attribute dictionary from Disembodied Property List
dpl = DPL([False, 'x', 1234, 'y', 'Spice', 'z', 3.14])
# can now access DPL data using attribute syntax
dpl['x']                    # returns 1234
dpl.x                       # returns 1234
dpl.y                       # returns 'Spice'
dpl.z                       # returns 3.14
```

Fill

Description: The Fill class is used to fill a physical component with patterns of either rectangles or instance array objects. The filling methods in this Fill class are `smart` methods which make use of the Santana `Geometry Engine` to perform the polygon filling operations, so that they will make use of the relevant design rules in the associated technology file. Note that this is in contrast to the fill methods which are provided as part of the Rect class, which do not make use of this Santana `Geometry Engine`.

Creation:

Note that all methods for this Fill class are static methods. Thus, there is no constructor method for this Fill class, and methods are invoked by directly invoking the class object name. That is, this Fill object will already exist in the Python environment and methods can be directly called as needed.

Methods:

fillPhysCompWithInstArrays(PhysicalComponent *comp*, Layer *encLayer*, Layer *layer*, InstanceArray *instArray*, Coord *layerExt*=None, Point *origin*=None, Grouping *group*=None) – fills the specified *comp* physical component with an instance array which is constructed on the specified *layer* and enclosure layer *encLayer*. The *instArray* parameter specifies the instance array which should be used to construct the fill pattern. The *layerExt* layer extension value allows a single value or a list value. If *layerExt* is a single value, the specified enclosure value would be same in both X and Y directions. If *layerExt* is a list value, the specified enclosure value would be enclosure-X and enclosure-Y separately. If the *layerExt* layer extension value is not specified, then the minimum extension design rule

value for the specified *encLayer* will be used. The *origin* parameter is used to specify an origin point for the set of fill patterns; if this origin point is not specified, then the fill patterns will be centered to the *comp* physical component. The *group* parameter can be used to store the fill patterns which are generated by this method; if this grouping parameter is not specified, then no grouping will be used.

fillPhysCompWithLargeArraysOfInstArrays(PhysicalComponent *comp*, Layer *encLayer*, Layer *layer*, Coord *arraySpaceX*, Coord *arraySpaceY*, InstanceArray *instArray*, Coord *layerExt*=None, Point *origin*=None, Grouping *group*=None) – fills the specified *comp* physical component with a large array of instance arrays which are constructed on the specified *layer* and enclosure layer *encLayer*. The *arraySpaceX* and *arraySpaceY* parameter values specify the spacing between instance arrays in the array. The *instArray* parameter specifies the instance array which should be used to construct the array of instance arrays for the fill pattern. The *layerExt* layer extension value allows a single value or a list value. If *layerExt* is a single value, the specified enclosure value would be same in both X and Y directions. If *layerExt* is a list value, the specified enclosure value would be enclosure-X and enclosure-Y separately. If the *layerExt* layer extension value is not specified, then the minimum extension design rule value for the specified *encLayer* will be used. The *origin* parameter is used to specify an origin point for the set of fill patterns; if this origin point is not specified, then the fill patterns will be centered to the *comp* physical component. The *group* parameter can be used to store the fill patterns which are generated by this method; if this grouping parameter is not specified, then no grouping will be used.

fillPhysCompWithLargeArraysOfRects(PhysicalComponent *comp*, Layer *encLayer*, Layer *layer*, Coord *arraySpaceX*, Coord *arraySpaceY*, unsigned *numRows*, unsigned *numCols*, Coord *width*=None, Coord *height*=None, Coord *spaceX*=None, Coord *spaceY*=None, Coord *layerExt*=None, Point *origin*=None, Grouping *group*=None) – fills the specified *comp* physical component with a large array of rectangles which are constructed on the specified *layer* and enclosure layer *encLayer*. The *numRows* and *numCols* parameter values specify the dimensions of this array of rectangles, while the *arraySpaceX* and *arraySpaceY* parameter values specify the spacing between rectangles in the array. If the *width* or *height* parameter is not specified, then the minimum width design rule value for the specified *layer* will be used. Similarly, if the *spaceX* or *spaceY* parameter is not specified, then the minimum spacing design rule value for the specified *layer* will be used. The *layerExt* layer extension value allows a single value or a list value. If *layerExt* is a single value, the specified enclosure value would be same in both X and Y directions. If *layerExt* is a list value, the specified enclosure value would be enclosure-X and enclosure-Y separately. If the *layerExt* layer extension value is not specified, then the minimum extension design rule value for the specified *encLayer* will be used. The *origin* parameter is used to specify an origin point for the set of fill patterns; if this origin point is not specified, then the fill patterns will be centered to the *comp* physical component. The *group* parameter can be used to store the fill patterns which are generated by this method; if this grouping parameter is not specified, then no grouping will be used.

Feedback

fillPhysCompWithRects(PhysicalComponent *comp*, Layer *encLayer*, Layer *layer*, Coord *width*=None, Coord *height*=None, Coord *spaceX*=None, Coord *spaceY*=None, Coord *layerExt*=None, Point *origin*=None, Grouping *group*=None, GapStyle *gapStyle*=GapStyle.MIN_CENTER) – fills the specified *comp* physical component with equally sized rectangles which are constructed on the specified *layer* and enclosure layer *encLayer*. If the *width* or *height* parameter is not specified, then the minimum width design rule value for the specified *layer* will be used. Similarly, if the *spaceX* or *spaceY* parameter is not specified, then the minimum spacing design rule value for the specified *layer* will be used. The *layerExt* layer extension value allows a single value or a list value. If *layerExt* is a single value, the specified enclosure value would be same in both X and Y directions. If *layerExt* is a list value, the specified enclosure value would be enclosure-X and enclosure-Y separately. If the *layerExt* layer extension value is not specified, then the minimum extension design rule value for the specified *encLayer* will be used. The *origin* parameter is used to specify an origin point for the set of fill patterns; if this origin point is not specified, then the fill patterns will be centered to the *comp* physical component. The *group* parameter can be used to store the fill patterns which are generated by this method; if this grouping parameter is not specified, then no grouping will be used. The *gapStyle* parameter can be used to control how the spacing between rectangles should be distributed (as one of MINIMUM, DISTRIBUTE, PART_DISTRIBUTE, or MIN_CENTER); the default value for *gapStyle* is GapStyle.MIN_CENTER.

Examples:

```
# fill rectangle with field of sub-rectangles
r1 = Rect(Layer('diff'), Box(0, 0, 1.0, 1.0))
# first fill using DRC minimum sizes and spacing
g1 = Grouping()
Fill.fillPhysCompWithRects(r1, Layer('diff'), Layer('contact'),
                           group = g1)
# now fill using pre-specified sizes and spacing
g1.destroy()
Fill.fillPhysCompWithRects(r1, Layer('diff'), Layer('contact'),
                           width = 0.18, height = 0.18,
                           spaceX = 0.2, spaceY = 0.2)
```

CrossOver

Description: The CrossOver class is used to compute the minimum 45 degree gridded

X cross-over of two parallel paths. This `crossing-over` layout construction is often used in the generation of layout for more complicated PyCells such as spiral inductors, differential pairs or any cells which require special routing capabilities. Note that the widths of these two parallel paths may be different, and these different widths are automatically handled by this class. Once this 45 degree X cross-over has been computed, then this **CrossOver** class provides methods to generate the polygons which cross over and connect each of these two parallel paths. These 45 degree X cross-overs will be computed

based upon the widths of the two parallel paths, as well as the spacing between these parallel paths.

Creation:

CrossOver(Coord *width1*, Coord *space*, Coord *width2*=None, Grid *grid*=None, Coord *space45*=None, Coord *width45*=None) – creates a CrossOver object, based upon the minimum 45 degree X cross-over between two parallel paths. The *width1* value is the width of the first path, while the *width2* value is the width of the second path; if *width2* is not specified, then it will be set to the value of the first path (*width1*) by default.

The *space* value specifies the spacing between the first path and the second path. The optional *grid* value specifies the Grid object which should be used for snapping the diagonal values *width45* and *space45* to grid; note that the values for *width1*, *width2* and *space* will not be adjusted. The optional *space45* value specifies the diagonal spacing to be used; if not specified, this value will be the grid-snapped value of *space*, scaled by the square root of 2. The optional *width45* value specifies the diagonal width to be used; if not specified, this value will be the grid-snapped average value of *width1* and *width2*, scaled by the square root of 2.

Note that diagonal spacing is measure diagonally on the 45 degree or 135 degree axes.

In this case, any distances measured on these axes are automatically scaled by the square root of 2, in order to maintain exact numerical accuracy.

Methods:

genPath1Polygon(Direction *dir*, Direction *justify*, Point *innerPoint*, Layer *layer*) – generates the polygon corresponding to the cross-over points which are returned by the **getPath1Points**() method. The points in this polygon are ordered from the inner edge points to the outer edge points.

genPath2Polygon(Direction *dir*, Direction *justify*, Point *innerPoint*, Layer *layer*) – generates the polygon corresponding to the cross-over points which are returned by the **getPath2Points**() method. The points in this polygon are ordered from the inner edge points to the outer edge points

getDistance() – returns minimum distance required to create the 45 degree X cross-over obtained by connecting the parallel paths on one side of the cross-over to the other side.

getPath1Extends() – returns the list of two cross-over extensions for the first path, from the starting side of the cross-over. Note that the inner edge of the first path is the edge of the path which is closest to the second path.

getPath1Points(Direction *dir*, Direction *justify*, Point *innerPoint*) – returns the list of two point lists for the cross-over points for the first path; the first point list is the list of points for the inner edge points, while the second is the list of points for the outer edge. The *dir* parameter specifies the heading direction of the two paths; if this *dir* direction is not one of the four primary directions(NORTH, SOUTH, EAST, WEST), then an exception is raised.

The *justify* parameter specifies the justification of the first path relative to the centerline between the two paths with respect to the heading direction. For example, EAST means that the first path is to the right of the centerline and the second path, while WEST means that it is on the left side. The *innerPoint* parameter specifies the point of the first path inner edge at the start of the cross-over.

getPath2Extends() – returns the list of two cross-over extensions for the second path, from the starting side of the cross-over. Note that the inner edge of the second path is the edge of the path which is closest to the first path.

getPath2Points(Direction *dir*, Direction *justify*, Point *innerPoint*) – returns the list of two point lists for the cross-over points for the second path; the first point list is the list of points for the inner edge points, while the second is the list of points for the outer edge. The *dir* parameter specifies the heading direction of the two paths; if this *dir* direction is not one of the four primary directions(NORTH, SOUTH, EAST, WEST), then an exception is raised. The *justify* parameter specifies the justification of the second path relative to the centerline between the two paths with respect to the heading direction. For example, EAST means that the second path is to the right of the centerline and the first path, while WEST means that it is on the left side. The *innerPoint* parameter specifies the point of the second path inner edge at the start of the cross-over.

getSpace() – returns the spacing value between the first path and the second path

getWidth1() – returns the width of the first path

getWidth2() – returns the width of the second path

getWidth45() – returns the actual diagonal width value, which is scaled by the square root of 2. Note that the actual diagonal width value used by the layout construction may be different from the specified *width45* value specified in the **CrossOver**() creation method, due to rounding operations performed for symmetry and gridding constraints.

set(Coord *width1*, Coord *space*, Coord *width2*=None, Grid *grid*=None, Coord *space45*=None, Coord *width45*=None) – changes the dimensions of this **CrossOver** object, using the specified parameter values. Note that these parameter values are exactly the same as the parameter values used by the **CrossOver** creation method.

Attributes:

In addition to the methods for the CrossOver class, there are also four attributes (or properties) defined for this CrossOver class, described as follows:

distance – minimum distance required to create cross-over

path1Extends – list of cross-over extensions for first path

path2Extends – list of cross-over extensions for second path

width45 – actual diagonal width value

Feedback

Note that these attribute values will be the same values as returned by the corresponding methods for this **CrossOver** class. The **distance** attribute value will be the same value as returned by the "**getDistance**()" method, while the **path1Extends** attribute value will be the same value as returned by the "**getPath1Extends**()" method. The **path2Extends** attribute value will be the same value as returned by the "**getPath2Extends**()" method, while the **width45** attribute value will be the same as the "**getWidth45**()" method value.

Examples:

```
# first create two parallel paths, with same width
path1 = Path(Layer('metal1'), 0.5,
             [Point(-5.0, -0.25), Point(0, -0.25)])
path2 = Path(Layer('metal2'), 0.5,
             [Point(-5.0, 0.5), Point(0, 0.5)])
# create 45 degree CrossOver object
co = CrossOver(0.5, 0.25)
# generate the cross-over polygons
poly1 = co.genPath1Polygon(EAST, EAST, Point(0,0),
                           Layer('metal1'))
poly2 = co.genPath2Polygon(EAST, WEST, Point(0,0.25),
                           Layer('metal2'))
# also determine cross-over distance
dist = co.getDistance()        # returns 1.18
# get path extension values
co.getPath1Extends()           # returns [0.11, 0.32]
co.getPath2Extends()           # returns [0.11, 0.32]
# Get all points for path1 and path2 cross-over;
# note that will be same points as for polygons.
poly1.getPoints()
co.getPath1Points(EAST, EAST, Point(0,0))
poly2.getPoints()
co.getPath2Points(EAST, WEST, Point(0, 0.25))
# Now extend thse two parallel paths from cross-over point;
# note that the cross-over distance is used for this purpose.
path3 = Path(Layer('metal1'), 0.5,
             [Point(dist, 0.5), Point(5, 0.5)])
path4 = Path(Layer('metal2'), 0.5,
             [Point(dist, -0.25), Point(5, -0.25)])
```

Marker

Description: The Marker class is used to indicate design violations and the objects which are causing these design violations in the OpenAccess database. The location of the design violation can be represented using a list of points which is assigned to the marker. In addition, a message string which describes the design violation can also be attached to the marker. The objects which caused the design violation can also be associated to the marker. This Marker class provides the additional capability of assigning a name for the tool which reported the design violation, along with assigning a severity level for the design violation. The Marker class also provides the capability to specify whether a marker is

visible, when the OpenAccess design database is displayed using an OpenAccess based layout viewing tool. Note that when an exception occurs during the creation of a PyCell instance, then a Marker object will automatically be created for this design violation. This is in addition to the instance bounding box rectangle and the text string describing the error condition which are also created.

Creation:

Marker(ulist[Point] *points*, string *msg*=`Error marker`, string *shortMsg*=`Error marker`, string *tool*="*PyCell*", bool *isVisible*=True, bool *isClosed*=True,

MarkerSeverity *severity*=MarkerSeverity.FATAL_ERROR, MarkerDeleteWhen deleteWhen=MarkerDeleteWhen.NEVER) – creates a Marker object, based upon the specified parameter values. The *points* parameter specifies the list of points which describe the edges of the marker and describe the location of the design violation. The *msg* parameter is a string which desribes the detailed error condition for the marker, while the *shortMsg* string specifies an abbreviated error message string.

The *tool* parameter specifies the name of the design tool which reported the error condition for this marker. The *isVisible* Boolean parameter specifies whether this marker should be made visible when the OpenAccess database is viewed using a layout viewing tool. The *isClosed* Boolean parameter specifies whether the list of points for this marker represents a closed path. The *severity* parameter specifies the severity of the design error violation for this marker; this severity level should one of the following pre-defined values: ANNOTATION, INFO, ACKNOWLEDGED_WARNING, WARNING, SIGNED_OFF_ERROR, ERROR, SIGNED_OFF_CRITICAL_ERROR, CRITICAL_ERROR or FATAL_ERROR. The *deleteWhen* parameter specifies which condition this marker is getting deleted and its value should be one of the following pre-defined values: NEVER, FIRST, LAST or MODIFY

Methods:

addObject(PhysicalComponent | BlockObject | ulist[PhysicalComponent] | ulist[BlockObject] *obj*) – adds the specified physical component(s) or BlockObject (connectivity object) causing a design violation to the list of objects for this marker.

Note that the BlockObject class is the base class from which all connectivity classes (such as Net, Pin and Term) are derived. If the specified object is already associated with this marker, then an exception is raised.

clone(NameMapper *nameMap*=NameMapper(), NameMapper *netMap*=NameMapper()) – returns a cloned copy of this Marker object. Note that the *nameMap* and *netMap* parameters will be ignored, as this marker only refers to the physical components which are associated with this marker.

getBBox(ShapeFilter *filter*=ShapeFilter()) – returns the bounding box for this marker.

Note that this marker does not have layers associated with it, so that any layers specified in the *filter* ShapeFilter object will necessarily be ignored.

getMsg() – returns message string describing the design violation for this marker.

getDeleteWhen()() – returns the condition of the marker which is getting deleted and these are the pre-defined values: NEVER, FIRST, LAST or MODIFY..

getObjects() – returns the list of objects causing design violations which are associated with this marker.

getPoints() – returns list of points which specifies the area or location of the design error for this marker.

getSeverity() – returns severity level of the design violation for this marker; this is one of the pre-defined values: ANNOTATION, INFO, ACKNOWLEDGED_WARNING, WARNING, SIGNED_OFF_ERROR, ERROR, SIGNED_OFF_CRITICAL_ERROR, CRITICAL_ERROR or FATAL_ERROR.

getShortMsg() – returns the short message string which is associated with this marker.

getTool() – returns name of the tool reporting the design violation for this marker.

isClosed() – returns True if the list of points specifying the location of the design error for this marker is closed; returns False, otherwise.

isVisible() – returns True if this marker should be visible when it is displayed in a layout viewing tool; returns False, otherwise.

removeObject(PhysicalComponent | BlockObject | ulist[PhysicalComponent] | ulist[BlockObject] *obj*) – removes the specified physical component or BlockObject (connectivity object) causing a design violation from the list of objects for this marker. Note that the BlockObject class is the base class from which all connectivity classes (such as Net, Pin and Term) are derived. If the specified object is not associated with this marker, then an exception is raised.

setDeleteWhen(MarkerDeleteWhen deleteWhen) – sets the condition for the marker which is getting deleted; these are the following pre-defined values: NEVER, FIRST, LAST or MODIFY.

sets which condition this marker is going to be deleted; this should be one of the following pre-defined values: NEVER, FIRST, LAST or MODIFY.

setPoints(ulist[Point]) – sets the list of points which specifies the area or location of the design error for this marker.

setSeverity(MarkerSeverity) – sets the severity level of the design violation for this marker; this should be one of the following pre-defined values: ANNOTATION, INFO, ACKNOWLEDGED_WARNING, WARNING, SIGNED_OFF_ERROR, ERROR, SIGNED_OFF_CRITICAL_ERROR, CRITICAL_ERROR or FATAL_ERROR.

Feedback

setShortMsg(string *msg*) – sets short message string to be associated with this marker.

setTool(string *msg*) – sets name of the tool reporting the design violation for this marker.

Examples:

```
# first create Nmos instance
p = ParamArray()
inst1 = Instance('baseKit_cni130/Nmos', p)
# now create Marker to indicate design error
pts = inst1.getBBox().getPoints()
m1 = Marker(pts, msg='possible error', tool='PyCell_Checker',
            severity=MarkerSeverity.WARNING)
m1.addObject(inst1)
m1.setSeverity(MarkerSeverity.WARNING)
# obtain information about this Marker
m1.getMsg()             # returns 'possible error'
m1.getObjects()         # returns Nmos instance ([Instance 'I_0'])
m1.getPoints()          # returns list of points for Marker
m1.getSeverity()        # returns MarkerSeverity.WARNING
m1.getTool()            # returns 'PyCell_Checker'
```

DrcSummary

Description: The DrcSummary class is used to obtain information about the results of running the design rule checking program, which is based upon the Santana “Geometry Engine”. This DRC checking program can either be run interactively from the viewing tool, or directly through the Python API. The “**fgDrc**()” global function is used to run this design rule checking program.

When this design rule checking program is run through the Python API, it will make use of the design rules contained in the technology file which is attached to the DloGen design object which is specified for this “**fgDrc**()” function. Thus, the same environment is used to either run this DRC program interactively from the viewer tool or from the Python API. Note that the DrcSettings class is only provided to support future extensions of the “**fgDrc**()” function. Right now, it is not necessary to use this parameter, as the default parameter value will create an empty DrcSettings parameter, which will then be used when calling this “**fgDrc**()” method.

Methods:

checkedRules() – returns the number of design rules which were checked during the run of the design rule checking program. Note that this method returns the total number of checked rules, after the reduction of the design rules in the rule deck has taken place. For example, if there are only shapes defined on the ‘metal1’ layer in the current design, then this method will only return the number of design rules which are exclusively defined for the ‘metal1’ layer, and will ignore all other design rules.

Feedback

errorNumber(string *ruleName*) – returns the number of design rule checking errors generated for a specific design rule. The design rule is determined by the *ruleName* parameter. Note that if this particular design rule is not found in the DRC summary information, then zero will be returned by default.

getRuleList() – returns a list of all design rules which failed during the running of the design rule checking program. Each of these failed design rules is the rule name string for the design rule. Each of these design rule strings can then be used with the "**errorNumber**()" method, to determine the number of design rule checking errors which were generated for each of these failing design rules.

totalErrorNumber() – returns the total numbers of design rule errors found on the current design. This includes the total number of errors for all design rules which were checked during the running if the design rule checking program.

Examples:

```
# run design rule checking program on current design;
# it is being assumed that there are DRC errors in this design
ds = fgDrc(self.currentDloGen())
# now examine the output of running DRC program
ds.totalErrorNumber()    # returns total number of errors
ds.checkedRules()        # returns number of DRC rules checked
# get the number of DRC errors for each failed design rule
errorList = ds.getRuleList()
for rule in errorList:
  print "%d errors for rule %s" % (ds.errorNumber(rule), rule)
```

AppDef

Description: The AppDef class provides a way to add extension values to existing database objects that are used to create extensions on Shape, Instance, Via, Net, Term, Pin, and InstTerm objects. This AppDef class has no separate constructor for an object, and it is only necessary to call the get(), set(), and destroy() methods to access the extension values on existing database objects.

Methods:

destroy(*object*, string *name*) – destroys AppDef data with the *name* for the specified *object*.

get(*object*, string *name*) – returns the value of the AppDef with the *name* from the specified *object*. If there is no AppDef with this *name* on the *object* , then an exception is raised.

Feedback

set(*object*, string *name*, string *valueType*, *value*) – sets AppDef data according to the given *valueType* and *value* for the specified *object*. The *valueType* supports the following extension types:

- - boolean: Stores a boolean value on database objects.
- - int: Stores an integer value on database objects.
- - float: Stores a floating point number on database objects.
- - double: Stores a double precision floating point number on database objects.
- - string: Stores a string value on database objects.

Examples:

```
rect = Rect( Layer('metal3'), Box(0, 0, 1.0, 1.0))
# set a string value on rect object with the name RectStr
AppDef.set(rect, "RectStr", 'string', "PyCellStudio")
# set a float value on rect object with the name RectFloat
AppDef.set(rect, "RectFloat", 'float', 3.14)
# get the value from rect object by the name RectStr
s = AppDef.get(rect, "RectStr"))
# get the value from rect object by the name RectFloat
f = AppDef.get(rect, "RectFloat"))
```

CCAttr

Description: The CCAttr class provides a way to change color, stipple, line style of shapes, which are visible only by Custom Compiler. If the user doesn't use this class to change shape's outlook, the default is determined by Customer Compiler. This class is defined by the 'cci' module and doesn't work for PyCells whose view type is MASK_LAYOUT.

Methods:

setColor(*shape*, string *color*) – sets color of shape. Values of *color* (for example, 'gold', 'green', 'purple') are defined by Customer Compiler. When this value is set, the other two attributes (stipple, line style), if not set previously, will be set to 'solid' and 'solid' respectively.

setStipple(*shape*, string *stipple*) – sets stipple of shape. Values of *stipple*(for example, 'triangle', 'solid', 'stipple3') are defined by Customer Compiler. When this value is set, the other two attributes (color, line style), if not set previously, will be set to 'white' and 'solid' respectively.

setLineStyle(*shape*, string *lineStyle*) – sets line style of shape. Values of *lineStyle* (for example, 'mediumDashed', 'dots', 'hidden') are defined by Customer Compiler. When

this value is set, the other two attributes (color, stipple), if not set previously, will be set to ‘white’ and ‘solid’ respectively.

getColor(*shape*) – returns value of shape’s color. If color value is not set previously, then empty string (‘’) is returned.

getStipple(*shape*) – returns value of *shape*’s stipple. If stipple value is not set previously, then empty string (‘’) is returned.

getLineStyle(*shape*) – returns value of *shape*’s line style. If line style value is not set previously, then empty string (‘’) is returned.

Examples:

```
from cci import CcAttr
rect = Rect( Layer('metal3'), Box(0, 0, 1.0, 1.0) )
# set color of rect
CcAttr.setColor(rect, 'gold')
# set stipple of rect
CcAttr.setStipple(rect, 'triangle')
# set line style of rect
CcAttr.setLineStyle(rect, 'mediumDashed')
```

9

GLOBAL FUNCTIONS

There is a global function which is provided in the Python API to make it easier to create parameterized cell layout designs. This global function is used to perform DRC design rule checking.

Note that there is an extensive set of methods provided for the PhysicalComponent class. Many of these methods are also provided as global functions, since they are so often used in the creation of parameterized cell designs. These global functions are defined by the contents of the 'cni.aliases' module, which should be imported into the Python environment to make use of these global functions.

fgDrc(Dlo *dlo*, DrcSettings *drcSettings*=None)

Description: This function runs the design rule checking program (based upon the `Geometry Engine`) on the specified *dlo* design object. The returned value is the special *drcSummary* utility class object which can be used to query the results of running design rule checking; methods are provided for this class object to determine the number of errors, the number of design rules checked, etc. Note that the *drcSettings* default parameter is only provided for future enhancements in setting up the environment to run this design rule checking program. Right now, it is not necessary to use this parameter, as the default parameter value will create an empty DrcSettings parameter, which will then be used when calling this "**fgDrc**()" method. This function uses all of the design rules specified in the Santana technology file attached to the specified *dlo* design object, as is done when running the DRC program through the graphical viewing tool. Thus, the same environment is used to either run this DRC program interactively from the viewer tool or from the Python API.

Parameter List: This function takes two parameters. The *dlo* parameter is the Dlo design object on which the design rule checking program should be run. The drcSettings parameter is an optional parameter, and is only provided to support future extensions of this "**fgDrc**()" function. Right now, it is not necessary to use this parameter, as the default parameter value will create an empty DrcSettings parameter, which will then be used when calling the "**fgDrc**()" function.

Return Values: This function returns a single value, the DrcSummary class object, which contains all of the information about the results of running the design rule checking program. The different methods for this DrcSummary class can then be used to obtain and analyze the results of running the design rule checking program, as shown in the **Example** section below.

Examples:

```
# run design rule checking program on current design;
# it is being assumed that there are DRC errors in this design
ds = fgDrc(self.currentDloGen())
# now examine the output of running DRC program
ds.totalErrorNumber()    # returns total number of errors
ds.checkedRules()        # returns number of DRC rules checked
# get the number of DRC errors for each failed design rule
errorList = ds.getRuleList()
for rule in errorList:
  print "%d errors for rule %s" % (ds.errorNumber(rule), rule)
```

10

PYTHON PROGRAMMING ENVIRONMENT

Static Methods

There are a few methods in the Python API which have been defined to be static methods for certain objects. In the case where a class method is defined to be a static method, the method can be called without explicitly constructing an instance of that class object. This approach allows certain class methods to effectively be called as stand-alone functions, without the need to create the class object. This is done to allow these methods to be more generally used than they otherwise might be used.

In order to invoke one of these static methods, it is only necessary to call the static method using the class name, instead of the instance name for an object of that class. For example, in order to invoke the static method `areColinearPoints` from the Point class, the class name `Point` would be used, as follows: `Point.areColinearPoints(p1, p2, p3)`. Note that this syntax can only be used for methods which have been pre-defined to be static methods; if the method is not a pre-defined static method, then an exception is raised.

The following is a list of the objects and methods for which static methods have been defined. In addition, reasons for using each static method are provided.

Object Methods Remarks

Log test(), error(), writing messages to different pre-defined log files

warning(), info()

Unique Name() unique name generation

Grouping find() find named grouping in design

Instance find() find named instance in design

InstanceArray find() find named instance array in design

VectorInstance find() find named vector instance in design

Net find() find named net in design

Net findCreate() find named net in design; create if not found

BusNet find() find named bus net in design

BundleNet find() find named bundle net in design

Pin find() find named pin in design

Term find() find named terminal in design

Term findCreate() find named terminal in design; create if not found

BusTerm find() find named bus terminal in design

BundleTerm find() find named bundle terminal in design

InstPin find() find named instance pin in design

InstTerm find() find named instance terminal in design

Shape find() find named shape in design

Rect fillBBoxWithRects() fill bounding box with rectangles

Rect fillDiagBoxWithRects() fill diagonal box with rectangles

Rect fillDiagBoxWithDiagRects() fill diagonal box with diagonal rectangles

Fill fillPhysCompWithInstArrays() fill physical comp with instance arrays

Fill fillPhysCompWithLargeArraysOfInstArrays()

Fill fillPhysCompWithLargeArraysOfRects ()

Fill fillPhysCompWithRects() fill physical comp with rectangles

DloGen currentDloGen() current design object for class derivation

DloGen withNewDlo() used for batch regression testing

Lib create(), open() can create new library

RoutePath StraightLine() can create various types of routes

RoutePath StraightLineToBar()

RoutePath CShape()

RoutePath LShape()

RoutePath ZShape()

RoutePath FlightLine()

RoutePath checkLayerWidth()

RoutePath Connect()

Feedback

RoutePath findAdjacentInterconnectLayer()

Tech get() obtain Tech object from technology library name

Ellipse genPolygonPoints() generate list of points to approximate Ellipse

Point areColinearPoints() can check any three points for co-linearity

Point invalid() create invalid point for use with other classes

Direction getMembers() list of directions contained in Direction class

Examples:

```
Unique.Name('test')       # returns unique name
Lib.create('demo          # creates a new library
Instance.find('Contact7') # find instance 'Contact7' in design
p1 = Point(100,100)
p2 = Point(200,200)
p3 = Point(300,300)
Point.areColinearPoints(p1, p2, p3)  # returns True
```

Python Namespaces

The class objects and their associated methods which have been described in this document for the Python API are organized into a small number of different Python modules. The reason for doing this is to keep the associated name spaces well organized, and also provide an overall structure for this Python API. The following Python modules which can be imported into your Python environment can be described as follows:

cni.dlo – contains all class objects for the Python API which are to be used for authoring parameterized cells.

cni.constants – contains constant definitions for important symbolic constant objects used in the Python API, so that these symbolic constant objects can be used globally. Note that the cni.utils.importConstants(class) method can be used to selectively import constants.

cni.aliases – contains a number of useful aliases for methods defined for the DloGen and Tech class objects, as well as some useful global convenience functions for creating design objects. This module is generally meant to be used when the `PyCell Explorer` tool is used for doing interactive design exploration. Otherwise, it is best to use the fully qualified class object names for these various methods and convenience functions.

cni.geo – contains a number of useful methods for the PhysicalComponent class, concerned with geometric placement and alignment. Many of these methods are based upon the "Geometry Engine".

cni.utils – contains class for attribute dictionaries.

cni.integ.common – contains a number of useful methods for PyCell development, including testing. Also includes PyCell support for developing stretch handles and auto-abutment features when PyCells are used in conjunction with EDA layout editing tools.

cni.integ.cdfUtil – contains attribute dictionary class, as well as utilities for accessing CDF data stored on a cell in a library, and converting SKILL disembodied property lists into attribute dictionaries.

cni.po – contains several useful methods for point list geometric operations.

These different Python modules have the following contents:

cni.dlo – contains the fundamental class objects for the Python API, including the following objects: 'AbutContact', 'Arc', 'ArcRef', 'ArrayInstContact', 'AttrDisplay', 'AttrDisplayRef', 'Bar', 'Box', 'BusNet', 'BusTerm', 'ChoiceConstraint', 'CompoundComponent', 'Contact', 'ContactRing', 'CoordValue', 'DeviceContext', 'DeviceContextManager', 'Direction', 'Dlo', 'DloGen', 'Donut', 'DonutRef', 'Dot', 'DotRef', 'Ellipse', 'EllipseRef', 'FailAction', 'Font', 'GapStyle', 'Grid', 'Grouping', 'GroupingRef', 'IPhysicalComponent', 'Instance', 'InstanceRef', 'InstanceArray', 'InstanceArrayRef', 'InstanceArrayMemberRef', 'InstPin', 'InstTerm', 'Layer', 'LayerMaterial', 'Lib', 'Line', 'LineRef', 'Location', 'Log', 'MultiPath', 'NameMapper', 'Net', 'Numeric', 'Orientation', 'ParamArray', 'ParamConstraint', 'ParamSpecArray', 'Path', 'PathRef', 'PathSeg', 'PathSegRef', 'PathStyle', 'PhysicalComponent', 'Pin', 'Point', 'PointList', 'Polygon', 'PolygonRef', 'PropSet', 'Range', 'RangeConstraint', 'Rect', 'RectRef', 'RoutePath', 'RouteTarget', 'RuleProperty', 'Ruleset', 'RulesetManager', 'Segment', 'Shape', 'ShapeFilter', 'ShapeRef', 'SignalType', 'SnapType', 'StepConstraint', 'Tech', 'Term', 'TermType', 'Text', ‘TextRef’, 'TextDisplay', 'TextDisplayRef', 'Topology', 'Transform', 'Unique', 'VectorInstance', 'VectorInstanceRef', 'ViewType'

It contains the following functions used for debugging:

'getDebuggerBreakOn', 'setDebuggerBreakOn', 'getDebuggerSourceTraceOn', 'setDebuggerSourceTraceOn', 'getDebuggerSourceTraceDepth', 'setDebuggerSourceTraceDepth'

It also contains the following utility functions and methods:

'DrcSettings', 'DrcSummary', 'fgDrc', 'ulist', 'cFloat'

cni.constants – contains useful symbolic names for enumerated constants, which are used for different methods and functions throughout the Python API:

Direction object: 'EAST', 'EAST_WEST', 'NONE', 'NORTH', 'NORTH_EAST', 'NORTH_SOUTH', 'NORTH_WEST', 'SOUTH', 'SOUTH_EAST', 'SOUTH_WEST', 'WEST' , ‘CENTER’, ‘ANY’

Orientation object: 'MX', 'MXR90', 'MY', 'MYR90', 'NONE', 'R0', 'R180', 'R270', 'R90'

FailAction object: 'ACCEPT', 'ANY', 'REJECT', 'USE_DEFAULT'

ViewType object: 'MASK_LAYOUT', 'SCHEMATIC', 'SCHEMATIC_SYMBOL', 'NETLIST', 'HIER_DESIGN'

Note that the cni.utils.importConstants(class) method can be used to selectively import only one or more of these different constant classes.

cni.aliases – contains

DloGen methods: 'addPin', 'addTerm', 'dbu2uu', 'dbu2uuArea', 'makeGrouping', 'save', 'setOrigin', 'uu2dbu', 'uu2dbuArea'

Tech methods: 'conditionalRuleExists', 'getGridResolution', 'getIntermediateLayers', 'getLayer', 'getPhysicalRule', 'physicalRuleExists'

cni.geo – contains

PhysicalComponent methods: 'abut', 'alignEdge', 'alignEdgeToPoint', 'alignLocation', 'alignLocationToPoint', 'fgAbut', 'fgAddEnclosingPolygon', 'fgDeriveLayer', 'fgMinSpacing', 'fgAnd', 'fgOr', 'fgNot', 'fgXor', 'fgSize', 'fgFill', 'fgMerge', 'fgPlace', 'getSpacing', 'place'

cni.utils – contains

Classes for attribute dictionaries, ordered dictionaries and parameter dictionary specifications: 'AttrDict', 'OrderedDict' and 'ParamDictSpec'

cni.integ.common – contains

Functions for PyCell development: 'createInstances', 'isEven', 'isOdd', 'renameParams', 'reverseDict'

Classes for PyCell development: 'Compare', 'Dictionary'

Functions for EDA tool integration: 'stretchHandle', 'stretchHandleCustom', 'autoAbutment'

Class for EDA tool integration: 'ProtocolString'

cni.integ.cdfUtil – contains

Classes for accessing CDF data stored on cell in library, as well as accessing SKILL DPL (disembodied property list): 'Symbol', 'CDF', 'DPL'

Deprecated Python API Elements

Although the Python API as a whole is subject to possible future changes to improve the overall functionality and ease of use, there are a small number of areas which are already planned to be improved or modified in the future. Thus, these elements of the Python API are deprecated, meaning that they should be used with the understanding that they may be changed in the very near future. Thus, Python programs written using such deprecated features of the Python API may need to be modified in the future by the user.

At this point in time, the following aspects of the Python API described in this document are deprecated, and are currently planned to be improved in a subsequent release:

1. The MultiPath class API which deals with different Path objects and sub-objects.

It is expected that additional features and capabilities will be added to this class in a future release of the Python API. It is possible that these improvements may change some aspects of the current methods and interfaces for this class.

1. The Dictionary class which is defined for PyCell development in the `cni.integ.common` module. Instead of this class, the AttrDict class in the Utility Classes should be used instead. This AttrDict class implements a Python dictionary, whose key/value pairs are usually accessed using attribute syntax.
2. The direct use of arbitrary Python objects as property values for the PropSet class will be accepted, but should not be used. Instead, the AppVal class should be used to assign Python objects as property values. For example, instead of using “rect1.prop[‘order1’] = [‘left’, ‘right’]”, the form “rect1.prop[‘order1’] = AppVal(‘PyObject’, [‘left’, ‘right’]) should be used.
3. The **_setEnableInstRecursion** and **_isEnabledInstRecursion** methods for the DloGen class which are used to allow the creation of recursive instances of the same supermaster are deprecated, and will be removed in a future release. These methods should not be used, since future releases of OpenAccess will no longer allow such recursive instances to be created. Since this capability will not be supported by OpenAcess, no replacement methods are planned to be provided.

Python Interface to C++ Classes

The various classes and methods defined for this Python API are actually implemented using the C++ programming language. These C++ classes are then `wrapped` so that the different class methods can be called from the Python programming environment. One major benefit of this approach is that it allows the Python API user to benefit from the increased system performance available from the use of C++ as an implementation language, while still enjoying the ease of use and convenience of programming in the Python language environment.

However, there are some subtle issues concerned with the interface between the Python and C++ programming environments. In particular, the use of two different programming environments necessarily involves some inherent tradeoffs and possible limitations. One of these limitations has to do with the interaction between C++ classes and Python classes. One of these limitations can occur in the case where the Python API author defines their own classes, which are derived from any of these C++ provided classes. If the Python API author then overrides an existing method already defined for this C++ class, other C ++ classes which should call this overridden Python method, will instead call the already existing C++ class method. The end result is that the Python user can not directly override

and re-define existing methods in the C++ classes. This should only be an issue, if the Python API author wanted to re-implement the basic class functionality which is already provided by the C++ classes.

As a simple example, assume that the Python API author wanted to derive their own Rect rectangle shape class, which was derived from the C++ Rect rectangle class. In addition, they also wanted to define their own bounding box calculation, by overriding the `getBBox()` method, which is already provided by the C++ rectangle class. The other C++ code which makes use of bounding box calculations would call the existing `getBBox()` C++ method, instead of calling the Python `getBBox()` method defined for this Python derived class.

11

STRETCH HANDLES AND AUTO-ABUTMENT

Conceptual Overview

In addition to providing the basic constructs which can be used to create parameterized cells, the Python API also provides capabilities to support interfaces with EDA tools, such as layout design editors. In particular, most interactive layout editing tools allow PCells to have features such as graphical stretch handles, as well as auto-abutment capabilities. A stretch handle for a PCell allows the designer to graphically use the mouse in the layout editing tool to stretch and resize different parts of the PCell layout, without the need to change PCell parameter values. When these shapes in the layout are modified using stretch handles, then corresponding PCell parameter values are automatically updated. The auto-abutment feature allows the designer to overlap two PCell instances, and then have the layout automatically adjusted to share structures, in order to minimize layout area. In the resulting layout, the two different PCell instances appear to be merged.

Synopsys has defined protocols to support both stretch handle and auto-abutment features. These two protocols can be used to provide a basic communication mechanism between the PyCell developer and the EDA interactive tool developer. These protocols are implemented by means of pre-defined properties which are associated with shapes in the PyCell layout and stored in the OpenAccess database. The values for these properties for stretch handles and auto-abutment are specified by the PyCell developer, and are then used by the EDA interactive tool developer. When the user stretches or abuts the associated shapes in the interactive layout editing tool, then the EDA tool can use these property values to update the PyCell parameter values.

This section describes requirements for the PyCell developer to support the stretch handle and auto-abutment features in their PyCell libraries. Note that a separate document is available for EDA tool developers (“PyCell EDA Tool Integration Guidelines”) which describes requirements to support these features in interactive layout editing tools.

Stretch Handles

Stretch handles provide a means for the user of the interactive layout editing tool to graphically manipulate parameter values for parameterized cells. These stretch handles are typically implemented in the layout editing tool as small diamonds which are attached

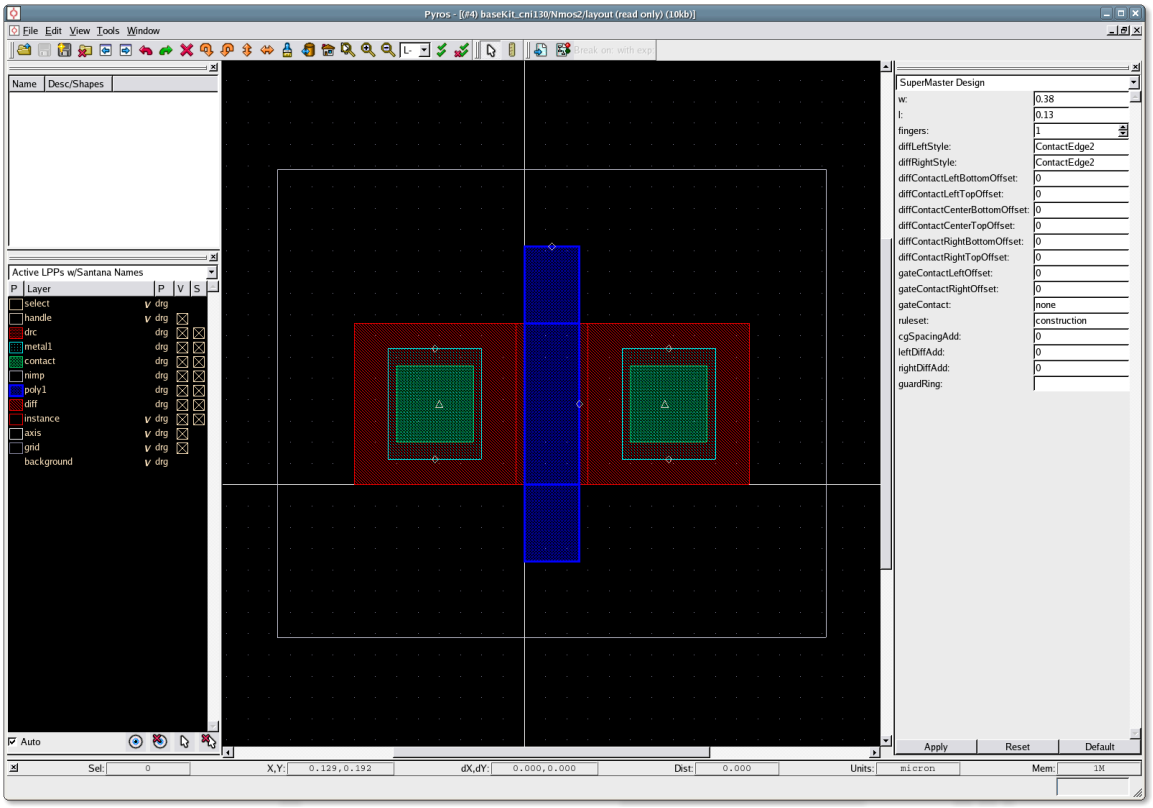
Feedback

to various shapes in the PCell layout. The user can select one of these stretch handle diamonds and then stretch and re-size the generated layout shape. This graphical stretching operation causes a PCell parameter value to be updated, and the PCell is re-instanced to create the corresponding submaster. This stretch handle is defined within the source code for the parameterized cell, but the graphical representation is manipulated within the EDA interactive graphical editing tool.

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

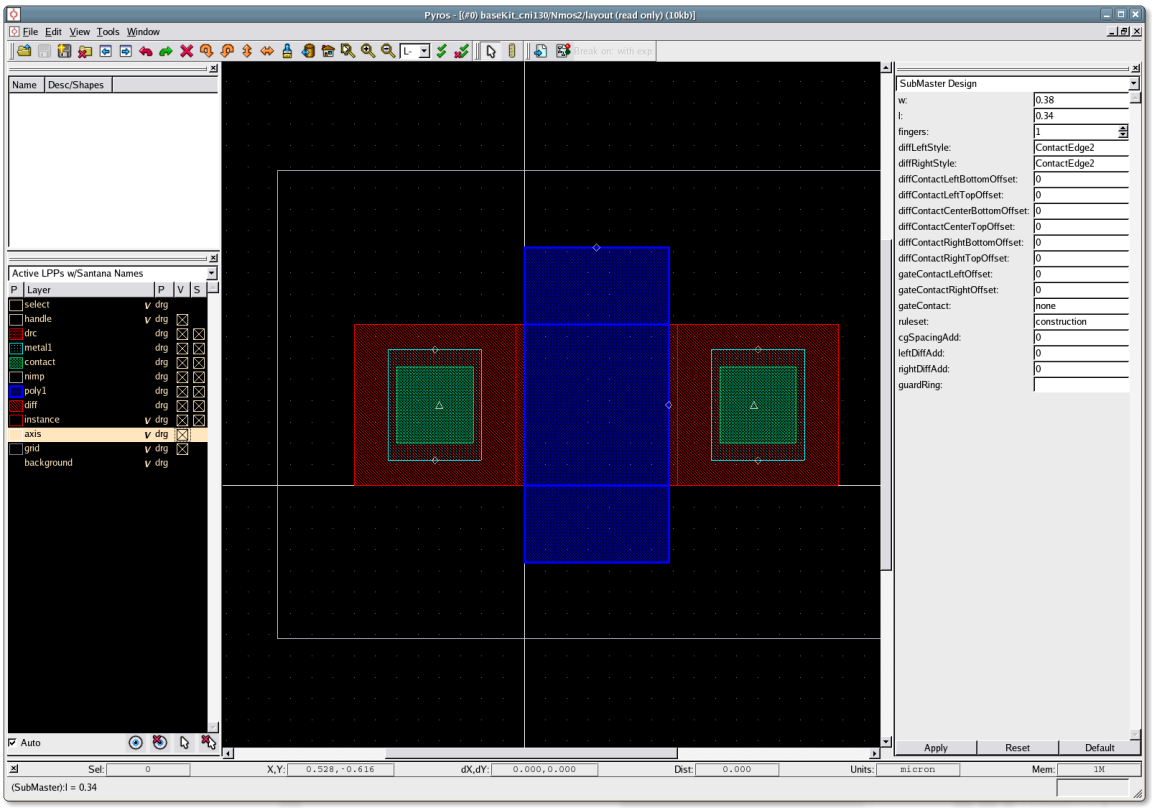
1. MOSFET transistor with default parameter values; stretch poly gate rectangle to the right, in order to increase the length parameter:

Select this stretch handle and drag to the right

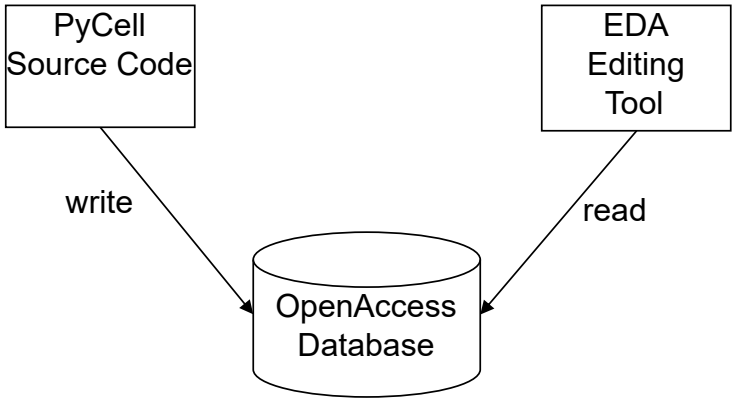


Feedback

1. 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.



The following is the sequence of steps which are required by the PyCell developer to support stretch handles in their PyCell designs:

1. Define stretch handles for all shapes (or instances or current design object) in the PCell layout which can be graphically stretched by the designer in the interactive layout editing tool. This is done in the PyCell source code, using the **stretchHandle**() function which is provided by the Python API.
2. Compile this PyCell source code into an OpenAccess database library file. This is done using the `cngenlib` utility, which compiles the Python PyCell source code into an OpenAccess library.
3. View the generated layout for this PyCell, using the Pyros layout viewer, or any OpenAccess layout editing tool which supports the stretch handle feature. Verify that these stretch handles operate as expected.

The Python API provides a function for defining stretch handles and attaching them to shapes (or instances or current design object) in the generated layout for the PyCell. This **stretchHandle**() function is defined in the cni.integ.common Python namespace, and defined as follows:

```
stretchHandle(Shape shape,
    string name,
    string parameter,
    Location location,
    Direction direction,
    string display = "",
    float minVal = None,
    float maxVal = None,
    string stretchType = "relative",
    string userSnap = "0.005",
    string userScale = "1.0",
    string key = None)
```

The *shape* parameter is the actual shape object in the PCell layout to which this stretch handle should be attached, while the *name* parameter is a unique name for this stretch handle. Note that this *shape* parameter can also be an instance or the current design object; for an instance, the *name* parameter should be the name of the instance and for the current design object, the *name* parameter will be ignored. In the case of a hierarchical PCell, the *shape* parameter can also be a reference to a shape in the PCell hierarchy, that is a ShapeRef object. In this case, the stretch handle property will be created on the corresponding instance object which contains the referenced shape. The *parameter* parameter is the name of the PyCell parameter being modified by the user. The *location* parameter specifies the location on the shape that should be used to attach the stretch handle, while the *direction* parameter specifies the directions in which the shape may be stretched (NORTH_SOUTH or EAST_WEST). The *minVal* and *maxVal* parameters are the minimum and maximum values for the associated PyCell parameter. Note that these minimum and maximum values can be expressed either as integer, floating-point or string

values. If string values are used, then these strings should represent numerical values, as described for the Python API **Numeric** class. The *stretchType* parameter is either `relative` or `absolute` and specifies how the stretched distance should be measured; `relative` is measured relative to the center of the *shape*, while `absolute` is the increment measured according to the absolute x and y directions. The *userSnap* parameter is the resolution value used for snapping the parameter value, while the *userScale* parameter is the scale factor used to multiply the change in PyCell parameter value. The *display* parameter for this **stretchHandle**() function is an optional parameter, which can be used to display a specified text string on the Text layer at the location of the stretch handle location point. This *display* text string can be used to provide additional identification for the stretch handle, in addition to the graphical display object normally used by the EDA layout editing tool for stretch handles.

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.

In addition, note that 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 PyCell. Parameters with multiple values are handled through the use of the optional *key* parameter. When this *key* parameter is None, then the PyCell parameter which is specified by the *parameter* string is a single-valued parameter. However, if this *key* parameter is any other string value, then it is the name of one of the multiple values for the multi-valued PyCell parameter. The value for this multi-valued PyCell 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, the shapes (or instances) which provide stretch handles are stored using an oaGroup, which is also named *pycStretch*. The EDA editing tool should first check for this oaGroup object stored on the PyCell master, before attempting to search all of the shapes (or instances) in the generated layout for 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 can not be stored using an oaGroup.

Feedback

Although it is not necessary for the PyCell developer to directly set any of the string-valued properties on the PyCell layout shapes which are used to define stretch handles, the following description is included to describe the underlying implementation of the **stretchHandle**() function provided by the Python API. This **stretchHandle**() function will directly create the required properties for the associated PyCell layout shapes (or instances or design object). These stretch handles are stored using string-valued properties on the shapes which are intended to be directly stretched or resized in the EDA interactive 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:

1. **name** – name for stretch handle; should be unique literal string within the PyCell.
2. **stretchType** – `relative` or `absolute`; `relative` is increment measured relative to the center of the shape (or instance or current design), while `absolute` is increment measured according to the absolute x and y directions.
3. **direction** – direction of measurement (NORTH_SOUTH or EAST_WEST).
4. **parameter** – the name of the PyCell parameter which is being modified by user.
5. **minVal** – minimum allowed value for the parameter being modified.
6. **maxVal** – maximum allowed value for the parameter being modified.
7. **location** – the location for the graphical stretch handle on the layout shape; this point should be specified using either a standard location, such as Location.UPPER_CENTER, or a custom location previously created on the shape by setCustomLocation().
8. **userScale** – scale factor used to multiply the change in parameter value.
9. **userSnap** – resolution value which should be used for snapping the parameter value.
10. **key** – the name used as a key to specify values for multi-valued parameters.
11. **shapeRef** – the name of the referenced shape inside an instance. This item will be stored only when the *shape* parameter is actually a reference to a shape in a hierarchical PCell. For example, if *shape* is a reference to a shape named "gate" inside an instance named "fet", then "gate" will be stored on the "fet" instance object.

This *pycStretch* string property associated with the shape in the PCell 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.

This *pycStretch* string property will be stored on any shapes, instances or design objects which are associated with stretch handles, using the **stretchHandle**() function..

Although this **stretchHandle**() function provided by the Python API is very flexible and provides a number of different options, there still may be situations in which the PyCell designer or EDA developer is not able to properly model specific stretch handle behavior using this function. In addition, it may be desired to directly access tool-specific features, which may not be supported by the **stretchHandle**() function. In such cases, the **stretchHandleCustom**() function can be used. This additional function simply encodes any arbitrary keyword argument as a function parameter, and then the PyCell designer or EDA developer handles the interpretation of these additional keyword arguments. This **stretchHandleCustom**() function is defined in the cni.integ.common Python namespace, and defined as follows:

stretchHandleCustom(Shape *shape*,

string *name*,

string *parameter*,

Location *location*,

Direction *direction*,

string *display* = "",

**kwargs)

The *shape* parameter is the actual shape object in the PCell layout to which this stretch handle should be attached, while the *name* parameter is a unique name for this stretch handle. Note that this *shape* parameter can also be an instance or the current design object; for an instance, the *name* parameter should be the name of the instance and for the current design object, the *name* parameter will be ignored. In the case of a hierarchical PCell, the *shape* parameter can also be a reference to a shape in the PCell hierarchy, that is a ShapeRef object. In this case, the stretch handle property will be created on the corresponding instance object which contains the referenced shape. The *parameter* parameter is the name of the PyCell parameter being modified by the user. The *location* parameter specifies the location on the shape that should be used to attach the stretch handle, while the *direction* parameter specifies the directions in which the shape may be stretched (NORTH_SOUTH or EAST_WEST). The *display* parameter for this **stretchHandleCustom**() function is an optional parameter, which can be used to display a specified text string on the Text layer at the location of the stretch handle location point. This **stretchHandleCustom**() function simply copies any optional keyword arguments and values which may be specified, and then appends them to the *pycStretch* string property which is associated with the shape in the PCell layout. It is then left for the EDA tool application to interpret these additional keyword arguments and values and perform any indicated operations.**Examples:**

```
# Note that complete source code for PyCell stretch handle
```

Feedback

```
# support can be found in the file "Mosfet2.py" included in
# the IPL library, which is part of PyCell Studio download.
# we use MOSFET as sample design for adding stretch handles
# Allow user to stretch poly gate rectangle to adjust width
# and length parameter values for transistor PyCell design.
# Stretching in NORTH_SOUTH direction changes width value,
# while the EAST_WEST direction changes length parameter.
# Assume that the polyGateRect variable contains the poly
# gate rectangle for the MOSFET transistor; both stretch
# handles will be assigned to this poly rectangle shape.
# create unique name for stretch handle
sname = Unique.name()
# define stretch handle to adjust width in NORTH_SOUTH direction
stretchHandle(
    name = sname,
    shape = polyGateRect,
    parameter = self.paramNames["w"],
    location = Location.UPPER_CENTER,
    direction = Direction.NORTH_SOUTH,
    stretchType = "relative",
    minVal = minWidth,
    userSnap = "%f" % (2.0 * self.tech.getGridResolution())
)
# define stretch handle to adjust length in EAST_WEST direction
stretchHandle(
    name = sname,
    shape = polyGateRect,
    parameter = self.paramNames["l"],
    location = Location.CENTER_RIGHT,
    direction = Direction.EAST_WEST,
    stretchType = "relative",
    minVal = minLength,
    userSnap = "%f" % (2.0 * self.tech.getGridResolution())
)
# The following simple example illustrates the use of
# a single stretch handle to modify multiple parameter
# values. As the stretch handle is stretched in the
# EAST_WEST direction, it modifies both the "l" and "w"
# length and width parameter values for a rectangle shape.
# Note that this is a complete parameterized cell example,
# consisting of a single rectangle shape on the "text" layer.
class simpleRectangle(DloGen):
    @classmethod
    def defineParamSpecs(cls, specs):
        specs('w', 1.0)
        specs('l', 1.0)
    def setupParams(self, params):
        self.x = params['w']
        self.y = params['l']
    # generate the layout for this simple Rectangle PyCell;
    # simply create single rectangle on fixed "text" layer.
    def genLayout(self):
        x = self.x / 2.0
```

Feedback

```
            y = self.y / 2.0
            r0 = Rect(Layer('text'), Box(-x, -y, x, y))
            # also assign stretch handle to both parameters;
            # note that the location is the same in both calls,
            # as this is really a single stretch handle.
            stretchHandle(
                name = 'r0',
                shape = r0,
                parameter = 'w',
                location = Location.CENTER_RIGHT,
                direction = Direction.EAST_WEST
            )
            stretchHandle(
                name = 'r0',
                shape = r0,
                parameter = 'l',
                location = Location.CENTER_RIGHT,
                direction = Direction.EAST_WEST
            )
    # The following example illustrates the use of parameters
    # with multiple values to define stretch handles for each
    # of the Source/Drain contacts for a multi-fingered MOSFET.
    # These stretch handles will allow the PyCell user to
    # add additional S/D contact metal at the top or bottom
    # of each contact, independently for each S/D contact.
    # The parameter values will be offset values for any
    # additional metal for each of these S/D contacts.
    # This "SDOffsets" parameter would be a string parameter
    # which would have values such as "top:0.05, bottom:0.01".
    # assume that Source/Drain contacts are saved in a list
    for i in range(1, self.fingers):

        sdContact = self.SDContacts[i]
        # create unique name for stretch handle
        handleName = "sdHandle_%d" % i
        # define stretch handle to adjust S/D contact width
        # in the NORTH_SOUTH direction from top of contact
        stretchHandle(
            name = handleName + "_top",
            shape = sdContact.getRect2(),
            parameter = "SDOffsets",
            location = Location.UPPER_CENTER,
            direction = Direction.NORTH_SOUTH,
            stretchType = "relative",
            minVal = 0.0,
            key = "top"
        )
        # define stretch handle to adjust S/D contact width
        # in the NORTH_SOUTH direction from bottom of contact
        stretchHandle(
            name = handleName + "_bottom",
            shape = sdContact.getRect2(),
            parameter = "SDOffsets",
```

```
        location = Location.LOWER_CENTER,
        direction = Direction.NORTH_SOUTH,
        stretchType = "relative",
        minVal = 0.0,
        key = "bottom"
    )
```

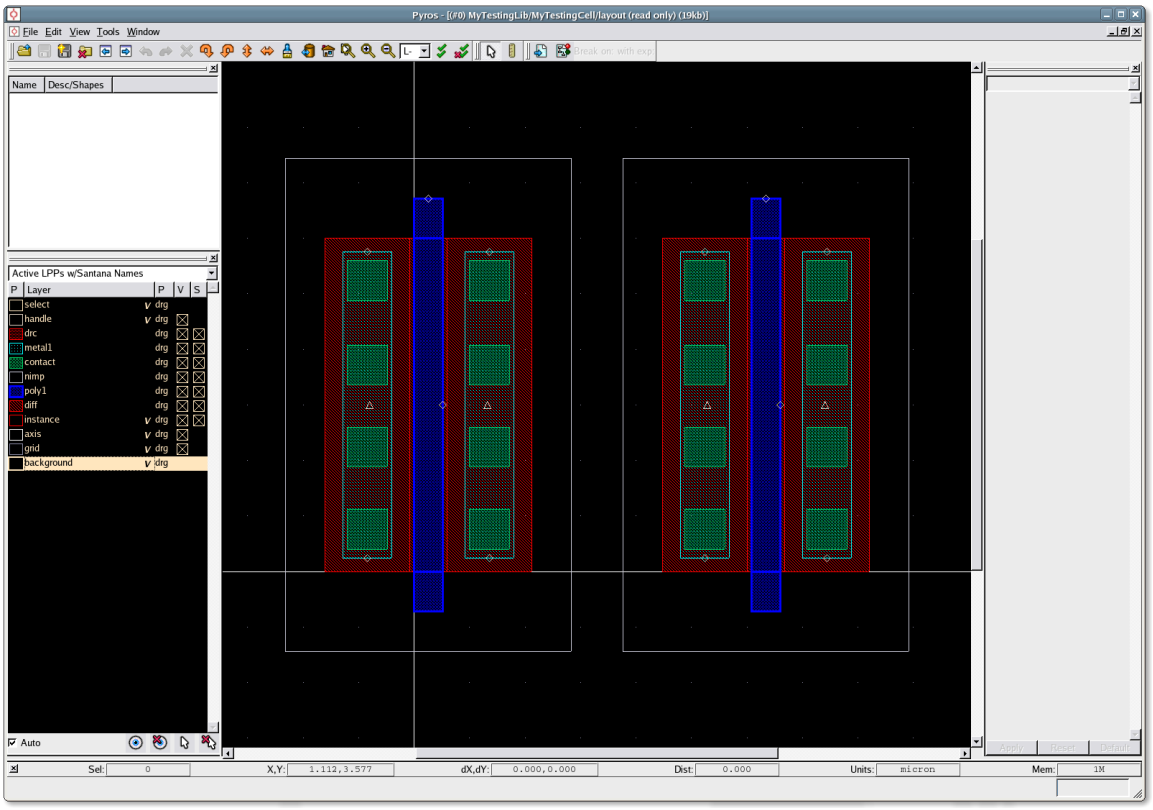
Auto-Abutment

Abutment is the process by which EDA interactive editing tools modify parameterized cell instances and place these instances in order to minimize design layout area. When two such instances are abutted, the generated layout can be modified, so that the two instances appear to be merged. As a typical 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. This abutment process then allows source and drain diffusion to be merged and shared between the two MOSFET 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:

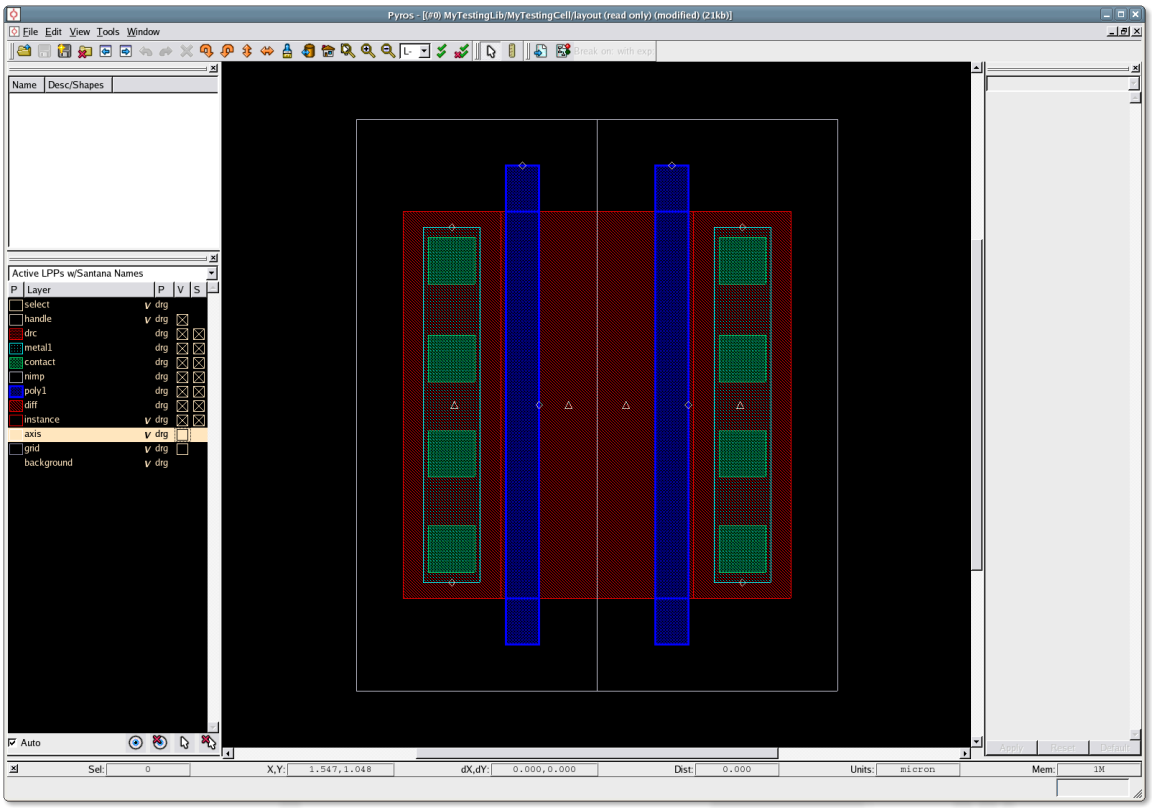
1. Two MOSFET transistor instances with the same dimensions; design rule correct spacing between these two instances:

Select this abutment triangle to move MOSFET instance

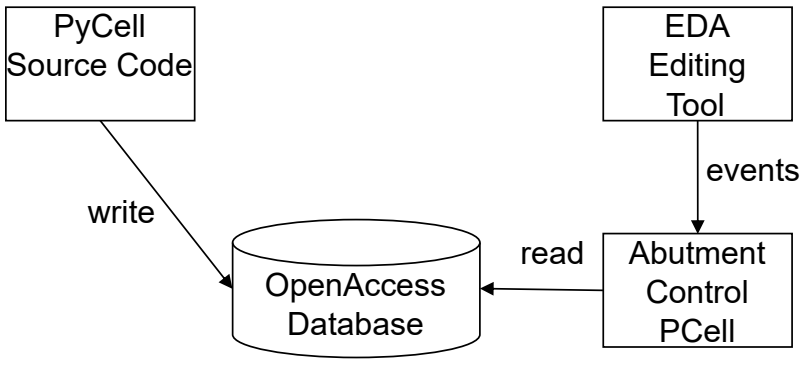

Pyros - [(#0) MyTestingLib/MyTestingCell/layout (read only) (19kb)]
File Edit View Tools Window
Name Desc/Shapes
Active LPPs w/Santana Names
P Layer P V S
select drg
handle drg
drc drg
metal1 drg
contact drg
nimp drg
poly1 drg
diff drg
instance drg
axis drg
grid drg
background drg
Auto
Sel: 0
X,Y: 1.112,3.577
dX,dY: 0.000,0.000
Dist: 0.000
Units: micron
Mem: 1M

Feedback

1. 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, in order to perform the requested abutment operations. This is as illustrated in the following diagram:



The following is the sequence of steps which are required by the PyCell developer to support this auto-abutment feature in their PyCell designs:

1. Define the auto-abutment properties for all pin shapes in the PCell layout which can be graphically abutted by the designer in the interactive layout editing tool. This is done in the PyCell source code, using the **autoAbutment**() function provided by the Python API.
2. Compile this PyCell source code into an OpenAccess database library file. This is done using the `cngenlib` utility, which compiles the Python PyCell source code into an OpenAccess library.
3. View the generated layout for this PyCell, using the Pyros layout viewer, or any OpenAccess layout editing tool which supports the auto-abutment feature. Verify that auto-abutment for each abuttable pin shape in the layout operates as expected.

The Python API provides a function for defining auto-abutment properties and attaching them to shapes in the generated layout for the PyCell. This **autoAbutment**() function is defined in the cni.integ.common Python namespace, and defined as follows:

autoAbutment(Shape *shape*,

float *pinSize*,

ulist[Direction] *directions*,

string *abutClass*,

list *abut2PinBigger*,

list *abut3PinBigger*,

list *abut2PinEqual*,

list *abut3PinEqual*,

list *abut2PinSmaller*,

list *abut3PinSmaller*,

list *noAbut*,

string *function* = "")

The *shape* parameter is the actual pin shape object in the PyCell layout which can be abutted, while the *pinSize* is the width of this pin shape. The *directions* parameter specifies the directions in which this pin shape may be abutted; this is a list of one or more abutment directions: NORTH, SOUTH, EAST or WEST. The *abutClass* parameter is a string which identifies the abutment class; this string is used by the interactive editing tool to ensure that two pin shapes from different PyCell instances can be abutted. The final

Feedback

function parameter specifies the name of any external abutment function which should be called; if no such abutment function is used, then this parameter is optional.

The next seven parameters are used to specify values for different types of abutment conditions, using a fixed format to specify values for these abutment conditions. These abutment conditions are categorized by the number of pin shapes which are connected to the same net (either 2 or 3 devices connected to the same net), as well as the relative size of the pin shapes which are being abutted. For example, the pin shape of the first PyCell instance (Pin A) may be smaller than, the same size, or bigger than the pin shape of the second PyCell instance (Pin B). These different abutment conditions are summarized in the following table:

Event	# Connections	Relative Size	Keyword
No abutment	0	N/A	noAbut
Abutment	2	Pin A < Pin B	abut2PinSmaller
	2	Pin A = Pin B	abut2PinEqual
	2	Pin A > Pin B	abut2PinBigger
	3	Pin A < Pin B	abut3PinSmaller
	3	Pin A = Pin B	abut3PinEqual
	3	Pin A > Pin B	abut3PinBigger

The *noAbut* parameter should be specified using a key-value pair, where the key is `spacing` and the value is the floating point value to be used for spacing between instances when no abutment is taking place. These other six abutment condition parameters (such as *abut2PinBigger*) are specified using three key-value pairs. The first key-value pair uses the key `spacing` to specify the spacing for the abutment condition. The second list of key-value pairs specifies the PyCell abutment parameters and their associated values for the abutment condition, when the PyCell instance is being moved for abutment. The third list of key-value pairs is specified in the same manner, but specifies the abutment parameters and values when the PyCell instance is stationary.

Note that this the **autoAbutment**() function will set all of the required pin shape properties for auto-abutment, except for the *pycShapeName* property; this property should instead be set by using the Shape **setName**() method for the pin shape.

Although it is not necessary for the PyCell developer to directly set any of the string-valued properties on the PyCell layout shapes which are used to define auto-abutment, the following description is included to describe the underlying implementation of the **autoAbutment**() function provided by the Python API. This **autoAbutment**() function will directly create the required abutment properties for the associated PyCell layout shapes.

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

1. *pycShapeName* – string property on each pin shape which can be abutted; this string property specifies a unique name for the shape, and is assigned by the PyCell developer. Note that the Python API Shape method **setName**() will assign the specified name to the shape, by setting this string property on the shape to the specified name.

In a similar fashion, the Shape method **getName**() will access this string property to obtain any name which has been assigned to the shape.

1. *pycPinSize* – float property on each pin shape which can be abutted; this float property specifies width of the pin shape. Note that this width information is not derived from the pin shape, in order to avoid any orientation dependent interpretation of width dimensions.
2. *pycAbutClass* – string property on each pin shape which can be abutted; this string property can be used to restrict the valid abutting parameterized cells. This string property can be checked by the interactive EDA editing tool to ensure that two pin shapes from different parameterized cell instances can be abutted.
3. *pycAbutRules* – string property on each pin shape which can be abutted; this string property is used to specify how the PyCell parameters are modified for different abutment events detected by the interactive EDA editing tool.
4. *pycAbutDirections* – string property on each pin shape which can be abutted; this string property is used to specify the valid abutment direction(s) for the abutting parameterized cell. This string value 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 is used to represent a list of lists of association lists. Each of these lists of association lists is composed of three association sublists: 1) the name of the abutment condition and the associated design rule spacing, 2) the PyCell parameter names and values to be used when the PyCell is being moved for abutment, and 3) the PyCell parameter names and values to be used when the PyCell is stationary during abutment. The name of the abutment condition can be set using the `_name` keyword, and the associated design rule spacing value can be set using the `_spacing` keyword. The names for these abutment conditions (using the `_name` keyword) are pre-defined, and should be one of the following names:

noAbut – no abutment

abut2PinBigger – abutment with 2 device connections, bigger pin width

Feedback

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 parameterized cell parameters and values for the moving PyCell instance and the stationary PyCell instance. Each of the key-value pairs in each of these association 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 the 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.

It should be noted that unlike the situation with the stretch handle feature, it is not required for the interactive EDA editing tool to directly read these abutment properties which are defined by the PyCell developer for abuttable shapes in the PCell layout. Instead, in order to provide a higher-level interface, Synopsysprovides an abutment control PCell. This abutment control PCell does not generate any layout geometries, but instead is used to control the abutment process for different shapes in the layout. This is done by processing specific abutment events which are generated by the EDA software.

Examples:

```
# note that complete source code for PyCell auto-abutment
# support can be found in the file "Mosfet2.py" included in
# the IPL library, which is part of PyCell Studio download.
# we use MOSFET as sample design for auto-abutment
# the MOSFET can be abutted either from the left side (WEST)
# or from the right side (EAST). The shape which we use for
# assigning auto-abutment properties is the left or right
# diffusion rectangle for the MOSFET. The corresponding PyCell
# abutment parameters are "diffLeftStyle" and "diffRightStyle".
```

Feedback

```
# define auto-abutment properties for left side diffusion;
# properties are set on the left/drain diffusion rectangle.
# set pycShapeName property
drainRect.setName("DRAIN")
autoAbutment(shape = drainRect,
             pinSize = self.w,
             directions = [Direction.WEST],
             abutClass = "cniMOS",
             abut2PinEqual = [ { "spacing" : 0.0 },
                     { "diffLeftStyle" : "DiffHalf" },
                     { "diffLeftStyle" : "DiffHalf" } ],
             abut2PinBigger = [ { "spacing" : 0.0 },
                     { "diffLeftStyle" : "DiffEdgeAbut" },
                     { "diffLeftStyle" : "DiffEdgeAbut" } ],
             abut3PinBigger = [ { "spacing" : 0.0 },
                     { "diffLeftStyle" : "ContactEdgeAbut2" },
                     { "diffLeftStyle" : "ContactEdgeAbut2" } ],
             abut3PinEqual = [ { "spacing" : 0.0 },
                     { "diffLeftStyle" : "DiffAbut" },
                     { "diffLeftStyle" : "ContactEdgeAbut2" } ],
             abut2PinSmaller = [ { "spacing" : 0.0 },
                     { "diffLeftStyle" : "DiffEdgeAbut" },
                     { "diffLeftStyle" : "DiffEdgeAbut" } ],
             abut3PinSmaller = [ { "spacing" : 0.0 },
                     { "diffLeftStyle" : "DiffEdgeAbut" },
                     { "diffLeftStyle" : "DiffEdgeAbut" } ],
             noAbut = [ { "spacing" : 0.4 } ]
)
# define auto-abutment properties for right side diffusion;
# properties are set on the right/source diffusion rectangle.
# set pycShapeName property
sourceRect.setName("SOURCE")
autoAbutment(shape = drainRect,
             pinSize = self.w,
             directions = [Direction.EAST],
             abutClass = "cniMOS",
             abut2PinEqual = [ { "spacing" : 0.0 },
                     { "diffRightStyle" : "DiffHalf" },
                     { "diffRightStyle" : "DiffHalf" } ],
             abut2PinBigger = [ { "spacing" : 0.0 },
                     { "diffRightStyle" : "DiffEdgeAbut" },
                     { "diffRightStyle" : "DiffEdgeAbut" } ],
             abut3PinBigger = [ { "spacing" : 0.0 },
                     { "diffRightStyle" : "ContactEdgeAbut2" },
                     { "diffRightStyle" : "ContactEdgeAbut2" } ],
             abut3PinEqual = [ { "spacing" : 0.0 },
                     { "diffRightStyle" : "DiffAbut" },
                     { "diffRightStyle" : "ContactEdgeAbut2" } ],
             abut2PinSmaller = [ { "spacing" : 0.0 },
                     { "diffRightStyle" : "DiffEdgeAbut" },
                     { "diffRightStyle" : "DiffEdgeAbut" } ],
             abut3PinSmaller = [ { "spacing" : 0.0 },
                     { "diffRightStyle" : "DiffEdgeAbut" },
```

Feedback

```
                        { "diffRightStyle" : "DiffEdgeAbut" } ],
                noAbut = [ { "spacing" : 0.4 } ]
    )
```

12

APPENDIX

The Python interpreter is built upon the standard Python interpreter, with the addition of a number of specific classes and methods which are used for creating parameterized cells. In addition, this Python interpreter also provides a `PyCell Explorer` module, which can be used to interactively create parameterized cell designs. This `PyCell Explorer` tool is also very useful for trying out different commands from the Python API, to see how they work. Since this `PyCell Explorer` tool can be used in conjunction with the viewer tool, it is possible to interactively enter Python API commands to the Python interpreter, and see the resulting physical design objects get created or modified in the viewer tool display window.

There are only three commands (Python functions) defined in this `PyCell Explorer` module. Two of these commands are used to either create or open an OpenAccess library, while the third command is used to interactively create a new DLO design object within the OpenAccess library. These three commands can be described as follows:

createLib(string *libName*, string *libPath* = '', string *techLibName* = '', string *techFilePath*='', *coreDlos*=True) – creates a new OpenAccess library, using the *libName* parameter for the name of this new library. If the *libPath* parameter is specified, then this new library will be created in the specified directory. Otherwise, the library will be created in the current working directory. Note that a technology binding must be created for this OpenAccess library. This is done by means of either the *techLibName* or *techFilePath* parameters. One (and only one) of these parameters must be specified, or else an exception is raised. The *techLibName* parameter specifies the name of the technology library which should be associated with this OpenAccess library, while the *techFilePath* parameter specifies the path to this technology library. If the *coreDlos* parameter is False, then none of the `core` PyCells defined for a technology library will be included in the library. By default, all of these `core` PyCells (Contact, AbutContact, ArrayInstContact and ViaCut PyCells) are included in the library. Note that the *libPath* and *techFilePath* parameters can be specified using environment variables and/or tilde home directory notation.

openLib(string *libName*, string *libPath* = '') – opens up an existing OpenAccess library, which has the name specified by the *libName* parameter. Note that this OpenAccess library should have been created by an earlier use of the "**createLib**()" function. If the optional *libPath* parameter is specified, then it should specify the file path location for this OpenAccess library; otherwise, the current working directory will be used by default. Note that this *libPath* parameter can be specified using environment variables and/or tilde home directory notation.

explore(string *cellName*, string *viewName* = '', bool *showGui* = False, bool *overwrite*=False) – creates a new DLO design object in the open OpenAccess library, and starts a new Python top-level interpreter. This top-level Python interpreter will have imported all of the various modules defined and used by the Python API (including cni.dlo, cni.constants, and cni.aliases modules). The *cellName* parameter must be used to specify the name of the cell being created in the OpenAccess library. If the *viewName* parameter is not specified, then `layout` will be used as a default view name for this DLO design object. If the *showGui* parameter is set to True, then the viewer tool will be invoked. If the *overwrite* parameter is set to False, and a DLO design object already exists in the OpenAccess library, then the user will be asked if it can be deleted.

The Python interpreter (`cnpy`) and the `PyCell Explorer` tool would then be invoked from the command line interface, as follows:

invoke Python interpreter

cnpy

import the `PyCell Explorer` module

>>> import cni.exp

create OpenAccess library named `demo` to store design;

if library already exists, then use openLib('demo') instead.

>>> createLib('demo', techLibName = 'cnTech_cni130')

create new DLO design object named `myCell` in this library

>>> explore('myCell', showGui = True)

At this point, all of the various classes, methods and functions defined for this Python API can now be used and commands can be interactively entered to the Python interpreter. Since the `PyCell Explorer` was invoked with the `showGui` flag set to True, the viewer tool will also be invoked along with the `PyCell Explorer` tool. For example, if the following Python command is entered:

rect1 = Rect(Layer('metal1'), Box(0, 0, 1, 1))

then the viewer tool will show a rectangle with these coordinates created and displayed on the 'metal1' layer. As interactive commands from the Python API are entered, the viewer tool window will be updated with the results from executing each interactive command. This direct interactive graphical feedback makes it very easy to see how various classes and methods defined in the Python API operate.

In order to exit from the editing session for the DLO design object, the `quit` command can be used. Note that the DloGen `save()` command should be entered to save the resulting DLO design object, before exiting the Python interpreter. Otherwise, the resulting DLO design object will not be saved in the OpenAccess design library.

Feedback

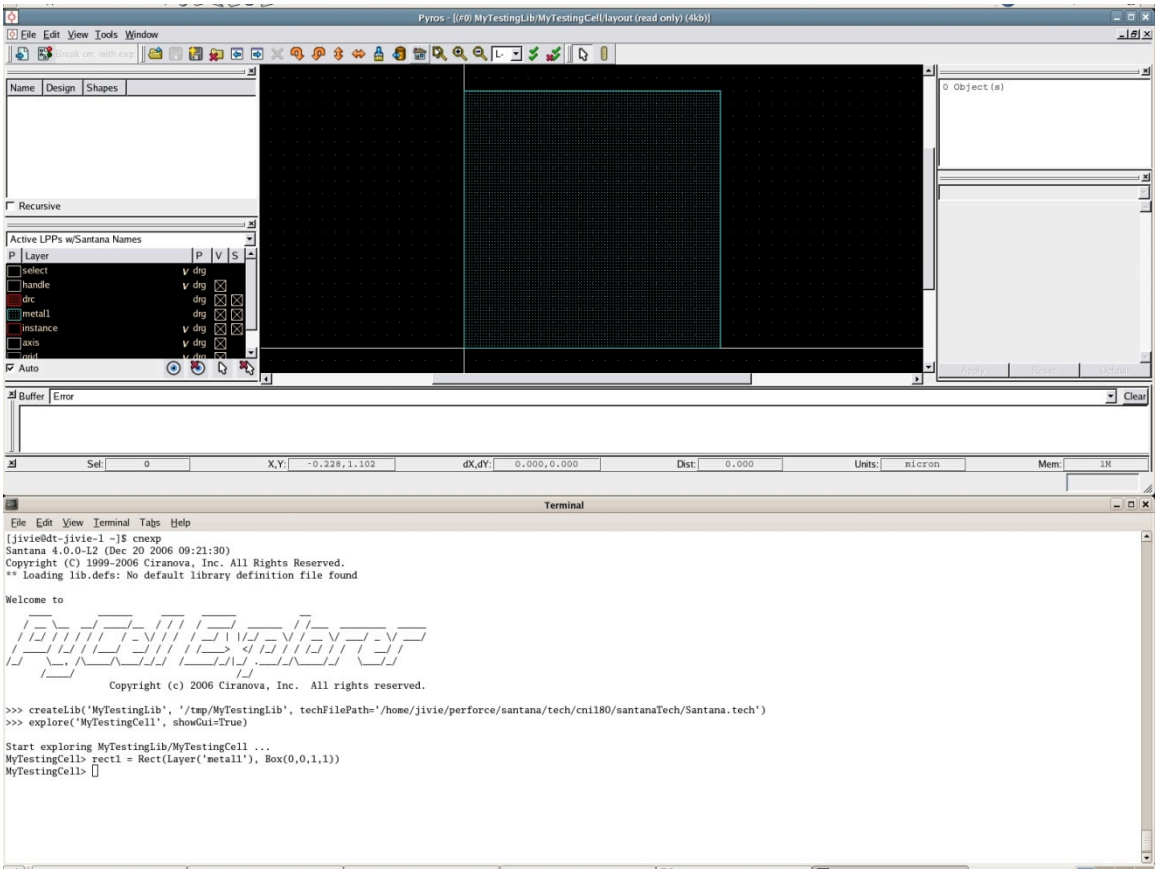
Note that there is also a special version of this `PyCell Explorer` tool which will create a default OpenAccess library, as well as associate a default technology file with this OpenAccess library. This version of the `PyCell Explorer` is very convenient to use whenever it is desired to try out different Python API commands to see how they work, without being concerned about the use of a specific technology file or OpenAccess library. For example, it is suggested that this version of the `PyCell Explorer` tool be used to try out the Python source code examples presented in this document.

In order to use this special version of the `PyCell Explorer` tool it is only necessary to invoke the `cnexp` command:

invoke PyCell Explorer

cnexp

If this command is successfully executed, then the Python interpreter is invoked, along with the viewer tool, shown as follows:



Note that we have typed in the Python `Rect()` command to create a rectangle shape on the `metal1` layer. This rectangle object is then automatically displayed in the viewer viewing window. This special version of the `PyCell Explorer` tool makes it as easy as

possible to experiment with different aspects of the Python API. In fact, it is recommended that this tool be used to enter any of the examples given in this document, to visually inspect the graphical layout which is generated by these design examples. This approach makes it easier to learn the Python API.

To exit the PyCell Explorer tool, use the **quit** command.