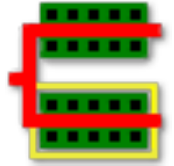


Glade

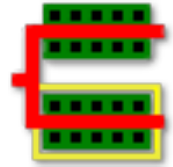
Peardrop Design Systems
30th June 2021



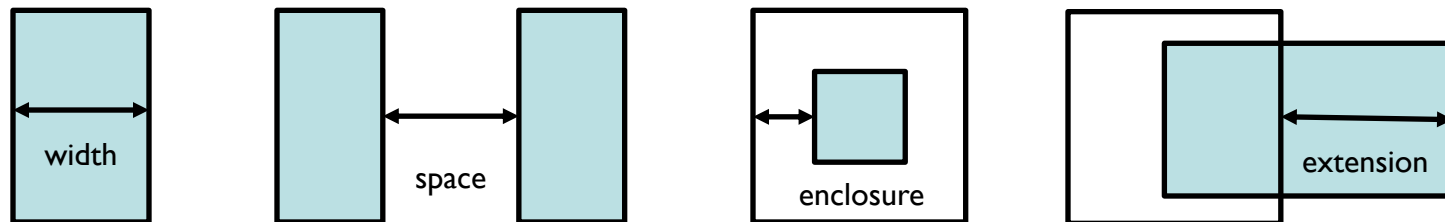
Agenda

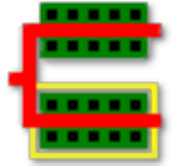
1. Introduction to DRC
2. Importing Shapes
3. Boolean Operations
4. Connectivity
5. DRC rules
6. Extraction
7. LVS
8. Net Tracer
9. Compare CellViews
10. Python

1. Introduction to DRC



- Design rules define how we draw transistors and their interconnect
 - E.g. width, spacing
- Many different geometric checks...

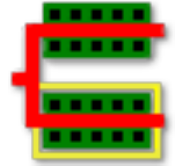




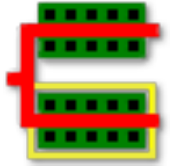
DRC In Glade

- Design Rule Checking (DRC) in Glade uses Python scripting
 - DRC rules files are scripts calling Python functions
- Boolean operations and DRC checks are performed by an edge based scanline algorithm (Bentley–Ottman)
- DRC commands for common operations
 - Boolean ops to create derived layers e.g. gate = poly AND active
 - Selection ops e.g. touching, abutting, inside/outside
 - Connectivity extraction for samenet/diffnet rules
 - DRC commands to check e.g. width, spacing, overlap etc.

A simple DRC example

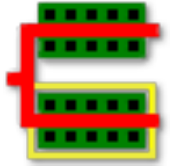


- A simple DRC script might look like this:
 - `from ui import *`
 - `cv = getEditCellView()`
 - `geomBegin(cv)`
 - `active = geomGetShapes("active", "drawing")`
 - `poly = geomGetShapes("poly", "drawing")`
 - `gate = geomAnd(active, poly)`
 - `geomWidth(gate, 0.18, "width < 0.18")`
 - `geomEnd()`
- 1. First we import the ui module so we can access the geom... Python functions
- 2. Next we get the cellView we want to check – in this case the current open cellView
- 3. We initialise the geometry engine with geomBegin, which takes a single arg – the cellView
- 4. We read in data on layers we wish to use
- 5. We perform some boolean operations to create derived layers
- 6. We perform a DRC check on the derived layer (in this case checking the width is not less than 0.18um)
- 7. Lastly we exit the geometry engine to free memory.



2. Importing Shapes

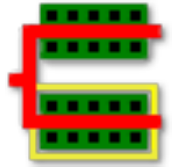
- `geomGetShapes()` is used to get all the shapes on a given layer
 - By default it flattens the hierarchy of the cellview
 - It creates an **edge file**. This is a temporary disk file in compact binary format that stores all shapes of a given layer as a set of edges for each polygon.
 - `geomGetShapes()` also merges shapes and orders polygon vertices as counterclockwise (internally the geometry engine stores polygons as counterclockwise, and holes as clockwise vertices).
 - The resulting edge file can be imagined as a '**layer**'. Layers generated from `geomGetShapes()` are known as '**original**' layers. Layers generated by subsequent boolean or selection functions are known as '**derived**' layers.



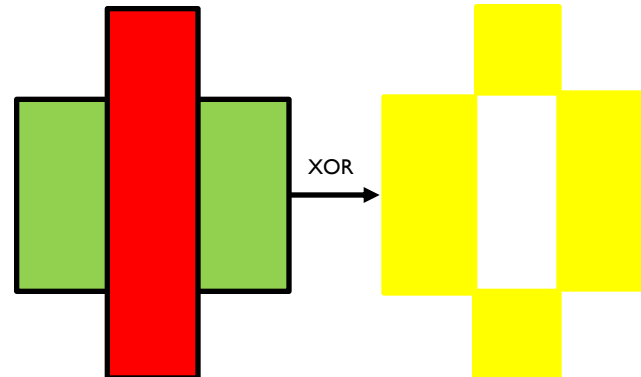
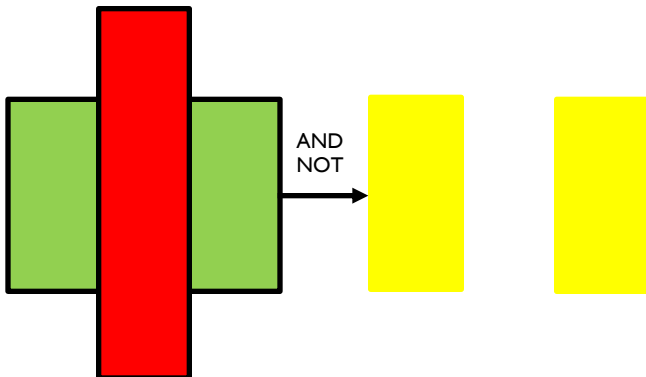
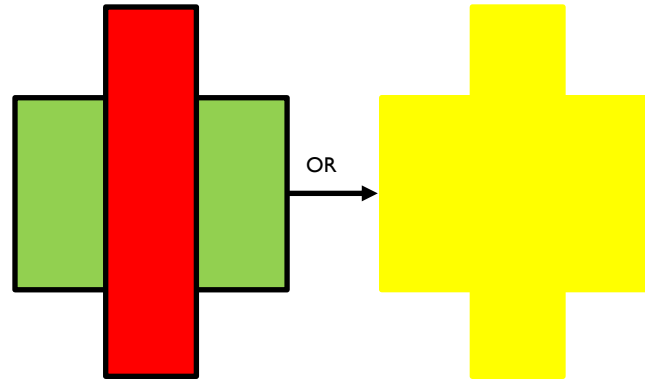
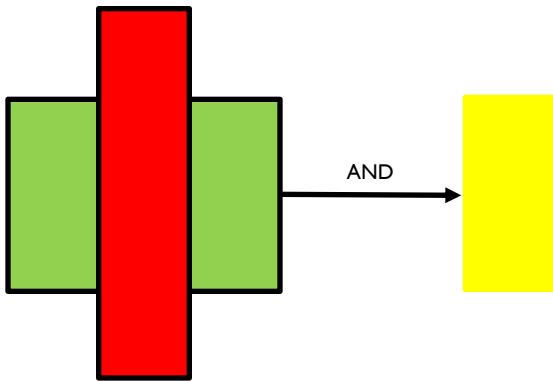
Importing Shapes

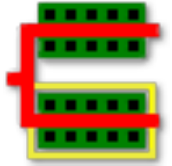
- Some specialized functions:
 - `outLayer = geomEmpty()`
 - Creates an empty layer
 - `outLayer = geomAddShape(layer, shape)`
 - Adds a shape to a layer.
 - `outLayer = geomAddShapes(layer, shapes)`
 - Adds a list of shapes to a layer
 - `outLayer = geomStartPoly(points)`
 - Creates a new layer from a list of points
 - `outLayer = geomAddPoly(layer, points)`
 - Adds a polygon from a list of points to an existing layer
- Useful for python scripts that create geometry

3. Boolean operations



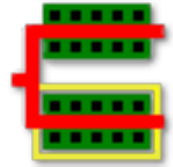
- Green OP Red -> Yellow





Boolean operations

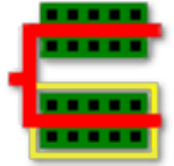
- Boolean operations used to create derived layers
 - `geomMerge()` # single layer OR
 - `geomOr()` # two layer OR
 - `geomAnd()` # two layer AND
 - `geomNot()` # inverse of layer, clipped to cellView bbox
 - `geomAndNot()` # subtracts a layer from another layer
 - `geomXor()` # XOR of two layers
 - `geomSize()` # up or down sizes a layer
 - `geomTrapezoid()` # converts polygons to trapezoids



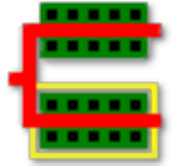
Selection operations

- Operations to select shapes based on some criteria
 - `geomTouching()` # shapes touching other layer shapes
 - `geomOverlapping()` # shapes overlapping
 - `geomInside()` # shapes inside others (may touch)
 - `geomContains()` # shapes inside but not touching internally
 - `geomOutside()` # shapes outside others (may touch or about)
 - `geomAvoiding()` # shapes outside and not touching or abutting
 - `geomInteracts()` # shapes that have any interaction
 - `geomButting()` # shapes with abutting edges
 - `geomCoincident()` # shapes with coincident edges
 - `geomHoles()` # get holes in shapes
 - `geomNoHoles()` # get shapes not including any holes
 - `geomGetTexted()` # get shapes that are labelled by some text
 - `geomGetNon90()` # get non Manhattan shapes
 - `geomGetNon45()` # get shapes that are not Manhattan or diagonal
 - `geomGetNet()` # get shapes that have named net info
 - `geomGetRectangles()` # get shapes that are rectangles
 - `geomGetPolygons()` # get shapes that are polygons
 - `geomGetVertices()` # get shapes with some number of vertices

Saving derived layers



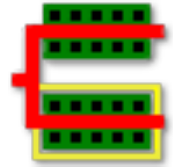
- Results from processing can be saved either as an error marker or as a new shape in the cellView
 - `saveDerived(layer, message)`
 - `saveDerived(layer, layerName, purpose)`
- The latter is useful for debug, or generating new layers in scripts



Labelling shapes

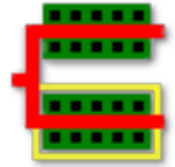
- `geomLabel()` will assign net names to shapes
 - Typical use is for extraction to label known nets
 - `geomLabel(metal1, "m1txt", "drawing")`
- The above uses text on the 'm1txt drawing' layer purpose pair to assign net names to shapes on metal1
 - Text origin must overlap the shape
- Labelled shapes will be used in connectivity extraction as starting points for connectivity tracing.

4. Connectivity extraction

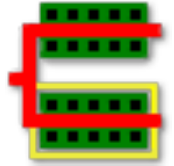


- The geometry engine can extract connectivity from shapes, using the `geomConnect()` function:
- `geomConnect([`
 - `[cont, active, poly, metal1],`
 - `[via1, metal1, metal2]`
 - `])`
- In this example we connect the active, poly or metal1 shapes to each other via the 'cont' layer. Similarly for metal1/metal2 by the 'via' layer.
- `geomConnect` will use net names from e.g. `geomLabel()` else it will assign generated names (n1, n2....) to connected shapes as net names.
- `geomConnect` will warn of shorts e.g if a shape labelled 'vdd' is eventually connected to another shape with a different label e.g. 'gnd'. It will report the coordinates and layers of the shorts.

5. DRC checking commands



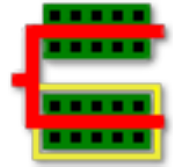
- DRC check commands check for specific rules:
 - `geomWidth()` : minimum width of a shape
 - `geomSpace()` : minimum space of shapes on 1 or 2 layers
 - `geomNotch()`: minimum space between edges of a shape
 - `geomArea()` : minimum / maximum area of a shape
 - `geomEnclose()` : enclosure of one shape by another
 - `geomExtension()` : extension of shape on one layer beyond shape on other layer's edge
 - `geomOverlap()` : minimum overlap of shape on one layer by shape on other layer.
- DRC commands generate markers on original layout that can be viewed by Verify->DRC->View Errors...
- DRC command also generate edge files that can be used:
 - `errorLayer = geomWidth(metal1, 0.4)`
 - Generates a derived layer 'errorLayer' with shapes that are the violations from the `geomWidth()` command.
 - Optional flag 'output_only' suppresses error marker generation



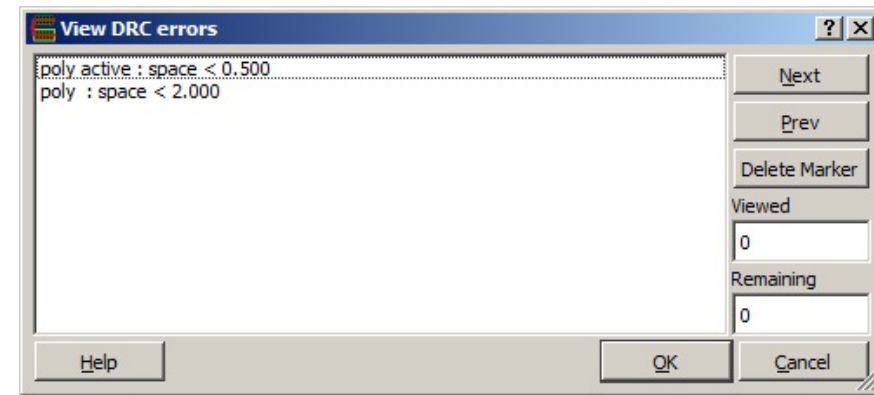
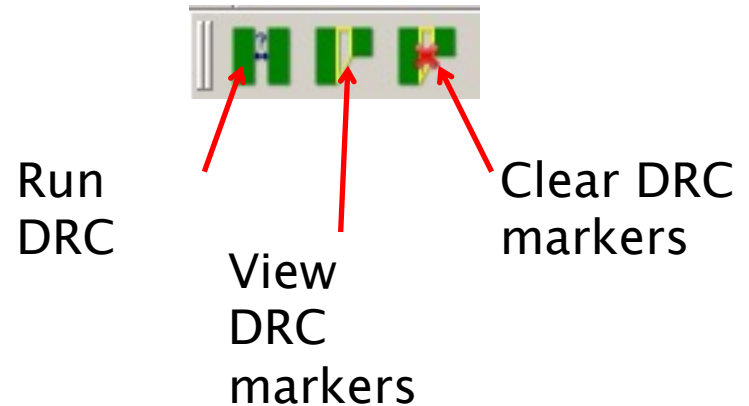
DRC checking flags

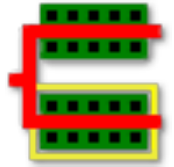
- Optional flags:
 - none
 - samenet
 - diffnet
 - vertical
 - horizontal
 - diagonal
 - project
 - parallel
 - abut
 - layer1
 - layer2
 - output_only
 - opposite
 - equals
 - not_equal
 - lessthan
 - lessorequal
 - greaterthan
 - greaterorequal
 - butting
 - coincident
 - outside
 - inside
 - over
 - not_over

Running DRC



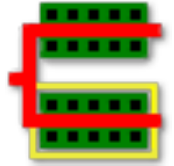
- Env var GLADE_DRC_FILE sets DRC rules file in 'Verify->DRC->Run' dialog
- Env var GLADE_DRC_WORK_DIR sets location of temporary files.
- Use Verify->DRC->View Errors... to display DRC marker viewer
 - Left click on rule will zoom in on first error for that rule
 - #Viewed / #Remaining show errors viewed and remaining to view.





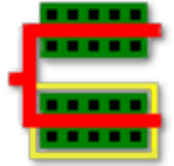
6. Extraction

- Extraction (Verify->Extract...) uses similar boolean processing as DRC.
 - Note that forming connectivity is optional for DRC, but mandatory for extraction!
- Extraction requires saveInterconnect() command to save connected shapes into extracted view.
 - This allows extraction of devices e.g. MOS, BJT, resistor etc.
- Devices are extracted using extract... commands.
 - Each needs a 'recognition region' i.e. a layer that uniquely identifies the type of device (e.g. gate = geomAnd(poly, active))
- Extraction uses PCells to form devices with a polygon outline created from the recognition region.
- Extracted view can be used for LVS or for netlisting (File->Export CDL...



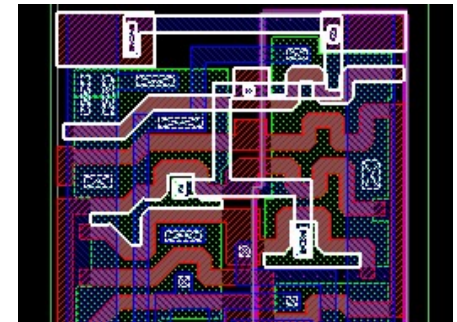
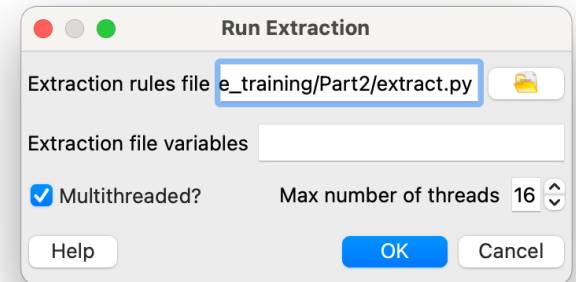
saveInterconnect

- saveInterconnect() is used to save shapes with connectivity to the extracted view when running extraction.
 - saveInterconnect([
 - [psub, "psub"],
 - nwell,
 - [ndiff, "od"],
 - [pdiff, "od"],
 - [polyg, "polyg"],
 - cont,
 - metal1,
 - via12,
 - metal2])
- Save layers **must** be derived from geomConnect()
- Any derived layer **must** be saved to a named layer
 - The layer will be created if not defined by the techfile

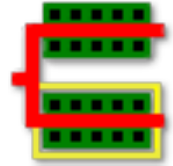


Running Extraction

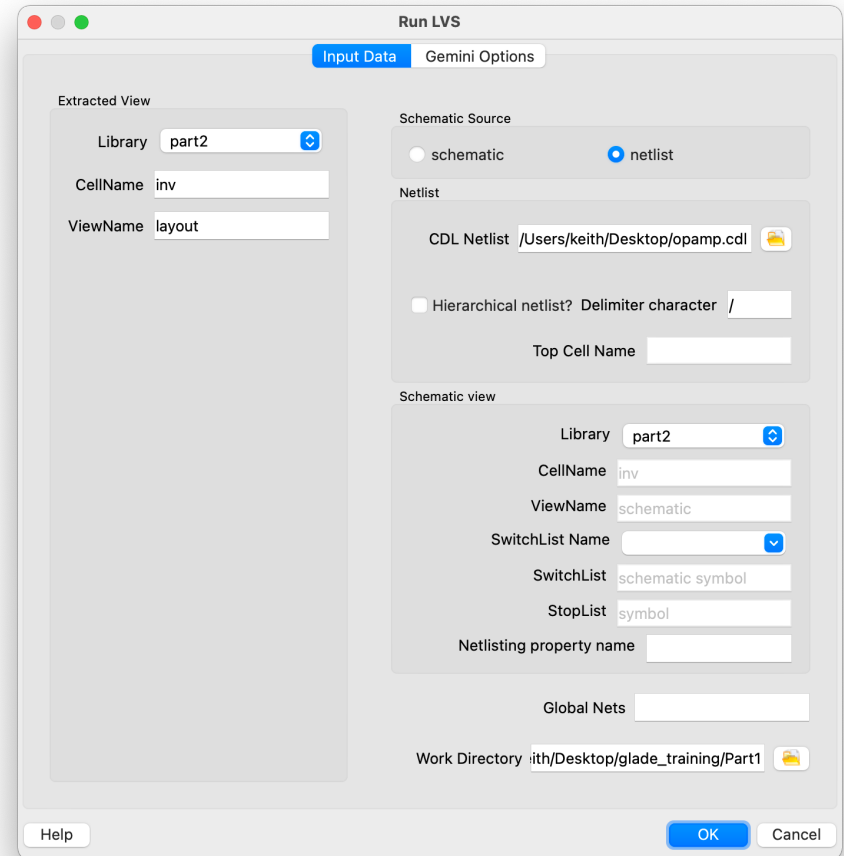
- Env var GLADE_EXT_FILE sets extraction rules file in 'Verify->DRC->Run' dialog
- Env var GLADE_DRC_WORK_DIR sets location of temporary files
- Set 'Selection Type' to 'Net' (F7 bindkey) to select all shapes on a net in the extracted view.
- Extraction will report any shorts found.



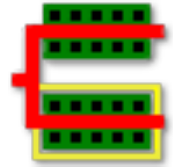
7. Running LVS



- Prerequisites:
 - Extraction must have completed successfully with no errors.
 - A schematic or a flat or hierarchical CDL/Spice netlist must be available.
 - Hier netlist uses cdlFlatten
 - Optionally, layout should have labels for primary IOs and power/ground (helps Gemini) or an equivalence file (matches layout net names to CDL/Spice net names)
 - Env var GLADE_NETLIST_FILE can be set to CDL netlist file name to preset the LVS form



Running LVS



- Gemini options
 - see Gemini documentation
 - (www-scf.usc.edu/~ee577/manual/gemini_man.ps)

Run LVS

Input Data Gemini Options

Gemini Flags

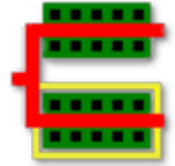
- ☐ Do not reduce different sized series transistors
- ☐ Do not reduce parallel MOS
- ☐ Do not reduce series MOS
- ☐ Warn for out-of-order series MOS
- ☐ Do not reduce series RLC
- ☐ Do not reduce parallel RLC
- ☐ Case insensitive net names
- ☐ Do not use local matching
- ☐ Match using properties (requires tolerance)
- ☐ Use subckt ports as equivalent nodes
- ☐ Do not optimise node labelling
- ☐ Warn for nets with zero connections
- ☐ Verbose mode

Gemini Variables

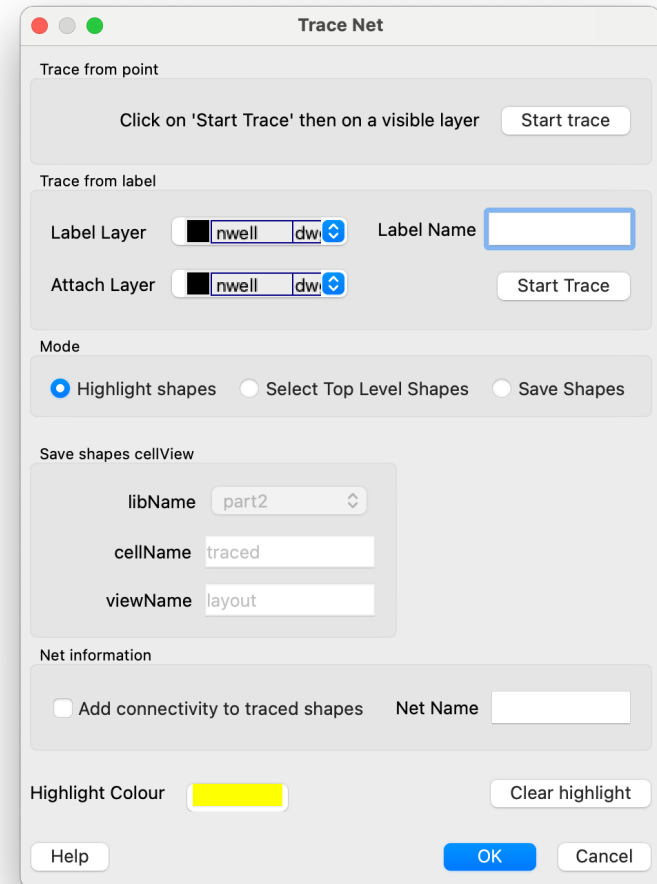
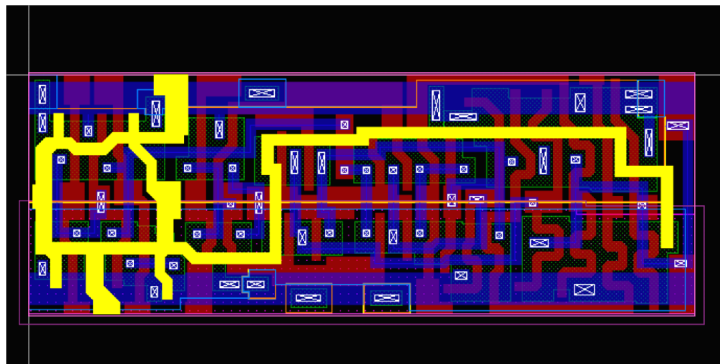
- ☐ Error limit 0
- ☐ Net error size limit 10
- ☐ No progress limit 2
- ☐ Suspect node limit 1
- ☒ Device parameter tolerance (%) 10.0
- ☐ Write equivalence file
- ☐ Read equivalence file

Help OK Cancel

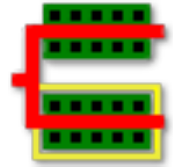
8. Net Tracer



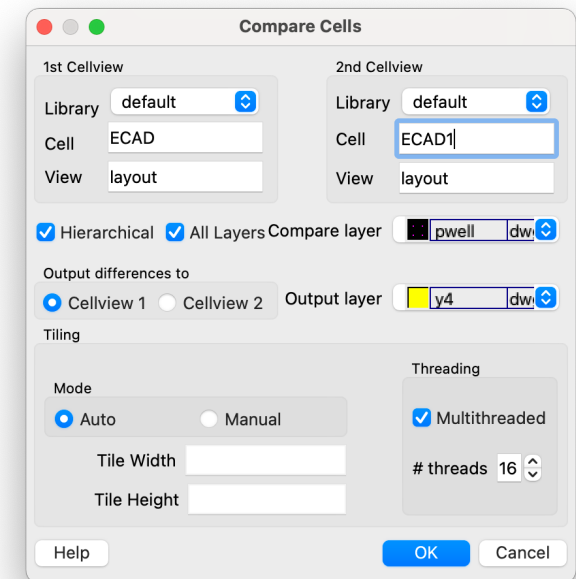
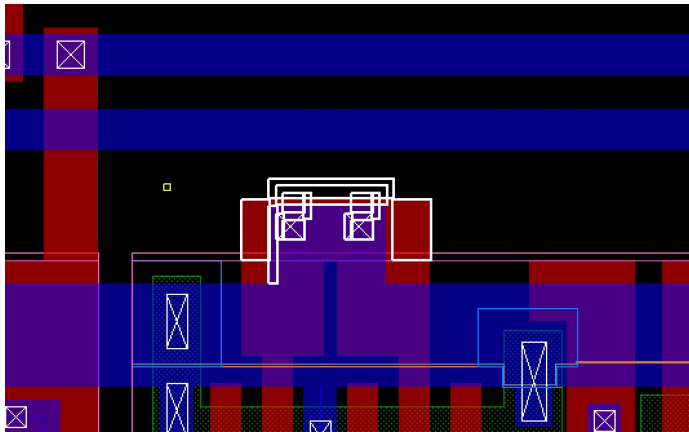
- Net Tracer traces shape connectivity
 - Uses techfile CONNECT rules
 - Can be set up using Set Layer Stack cmd
- Tracers from a starting point or a label
- Highlights traced shapes



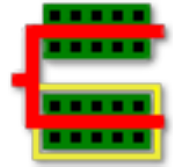
9. Compare Cells



- Compares cellViews using XOR
- Multithreaded
- Results can be stepped through using Verify->DRC->View Errors...



10. Python

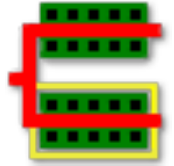


- Glade has an embedded Python interpreter.
- Database, GUI and geometry processing (DRC/Extract/LVS) code is wrapped using SWIG to give Python callable functions.
- Python code can be executed using File->Run Script... or typed into the command line.
- Python cmd line supports history (use up/down arrows) and standard ctrl character sequences.

```
>>> # INFO: Opened cell ECAD3 view layout
>>> cv = getEditCellView()
>>> print cv
<ui.cellViewPtr; proxy of C++ cellView instance at _2001e3ff_p_cellView>
>>> cv.bBox()
<ui.RectPtr; proxy of C++ Rect instance at _a008e2ff_p_Rect>
>>> cv.cellName()
'ECAD3'
>>>
```

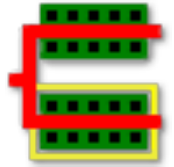
Python
command line

Python output
to the
message
window

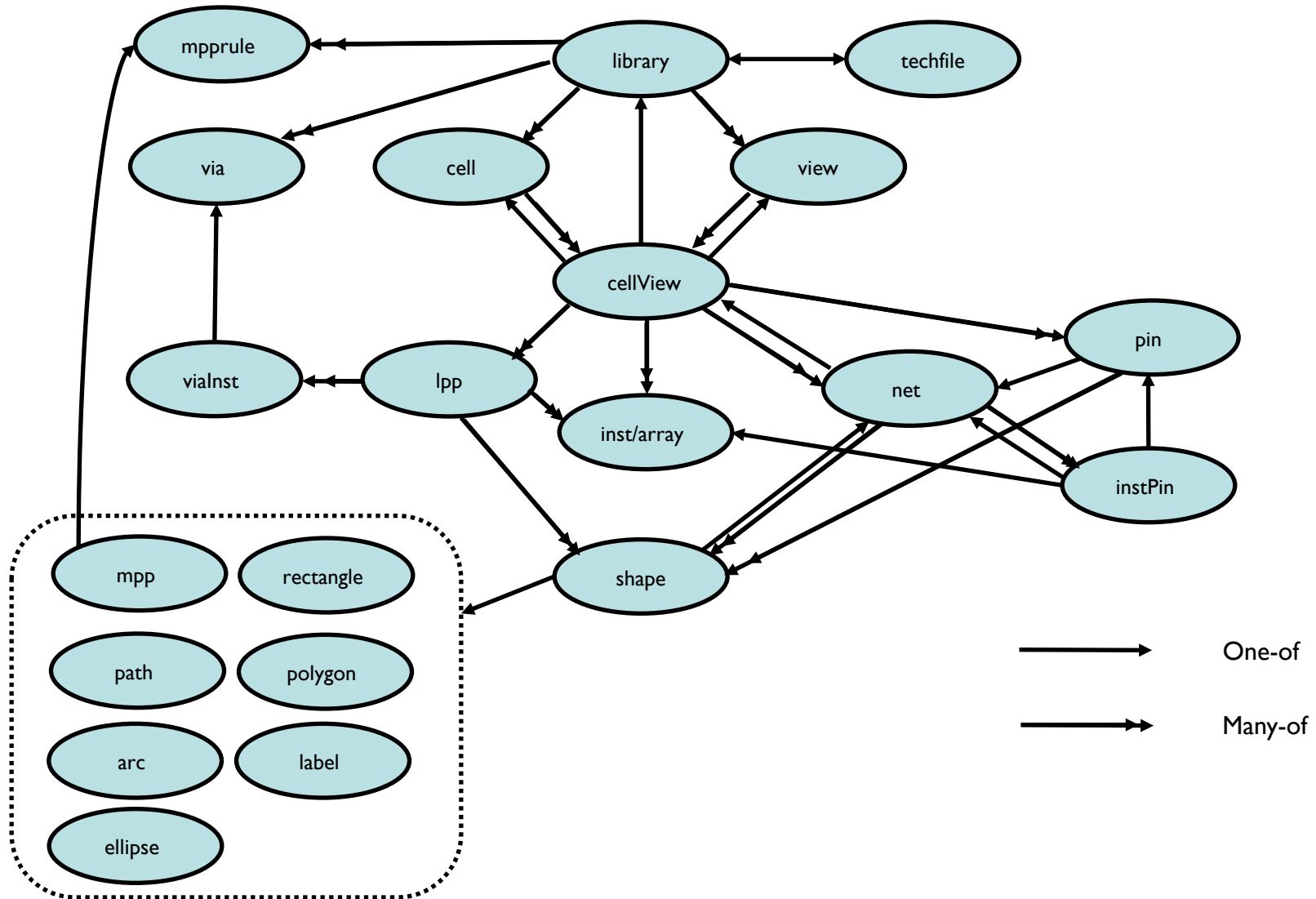


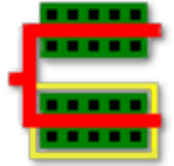
Python env vars

- Python looks for modules in PYTHONPATH env var!
 - Glade adds to this according to platform
 - %GLADE_HOME% (Windows & Mac)
 - \$GLADE_HOME/bin (Linux)
 - You may want to add paths to e.g. PCell libraries.
- Python distribution libraries are at
 - %GLADE_HOME%/Python383 (WIN32/64 – contains libs and DLLs)
 - \$PYTHONHOME (Linux/Mac, OS specific)



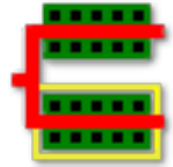
Python data model





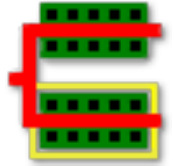
Accessor examples

- From a library:
 - `getCells()`
 - `getViews()`
- From a cellView:
 - `getLpps()`
 - `getInsts()`
- From a lpp:
 - `getShapes()`



Example Python code

```
• ui = cvar.uiptr                                # Get the reference to the ui instance
• lib = library("fred")                          # Create a library
• cv = lib.dbOpenCellView("test", "layout", 'w')  # Create a new cellView in the library
• tech = lib.tech()                             # Get the tech class associated with the library
• layer = tech.createLayer("layer1", "drawing")  # Create a layer we can draw on
• ui.openCellView(lib.libName(), cv.cellName(), "layout") # Open the cellView in a new window
• ptlist = [[1000,1000],[6000,1000],[6000,3000],[1000,3000]]
• poly = cv.dbCreatePolygon(ptlist, len(ptlist), layer, 1) # Create a polygon from the ptlist in dbu
• origin = Point(1000,1000)
• angle = 30.0
• trans = dbTransform64(angle, origin)           # Create a transform to rotate shape by
• poly.transform(trans)                          # Transform the polygon
• cv.update()                                    # Update the cellView and commit to db
• ui.winFit()                                    # Fit the window
```



9. Labs

- Glade_training/Part3
- DRC check/correct inverter from Lab2
- Run extraction
- Run LVS
- Python example