Design Rule Checks and Layout to Netlist Tool

Matthias Köfferlein, klayout.org

Introduction

This chapter will tell you ...

The basic idea of DRC and LVS

Briefly about the theory of DRC

About other applications of the DRC feature

Briefly about the theory of LVS

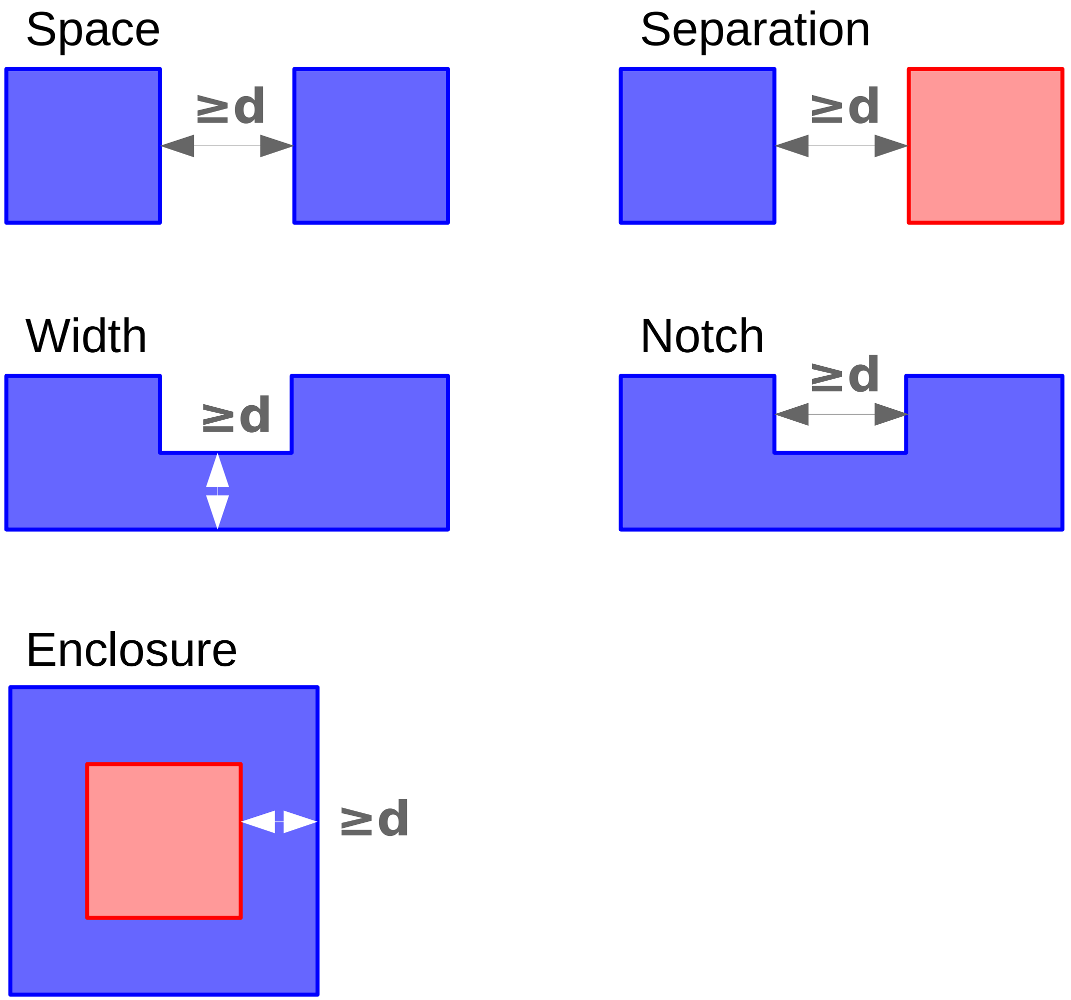
Basic Idea of DRC & LVS

- Design Rule Check (DRC) verifies that ...
  - Devices and interconnect structures can be manufactured with good yield
  - Good yield is important because a small likelihood of single-point fails adds to a high risk
- Layout vs. Schematic (LVS) verifies that ...
  - The layout represents the schematic

DRC Theory in a Nutshell

- The design manual lists the constraints of the manufacturing process as **design rules**
  - Basic rules include
    - Min space, width, enclosure, separation
    - Min area
  - Advanced rules
    - Width dependent spacing
    - Antenna rules
    - Max/min density or density variation
    - Arbitrary layout constraints
- The **DRC engine** provides features to implement these checks
- **DRC scripts** execute the checks to verify conformity

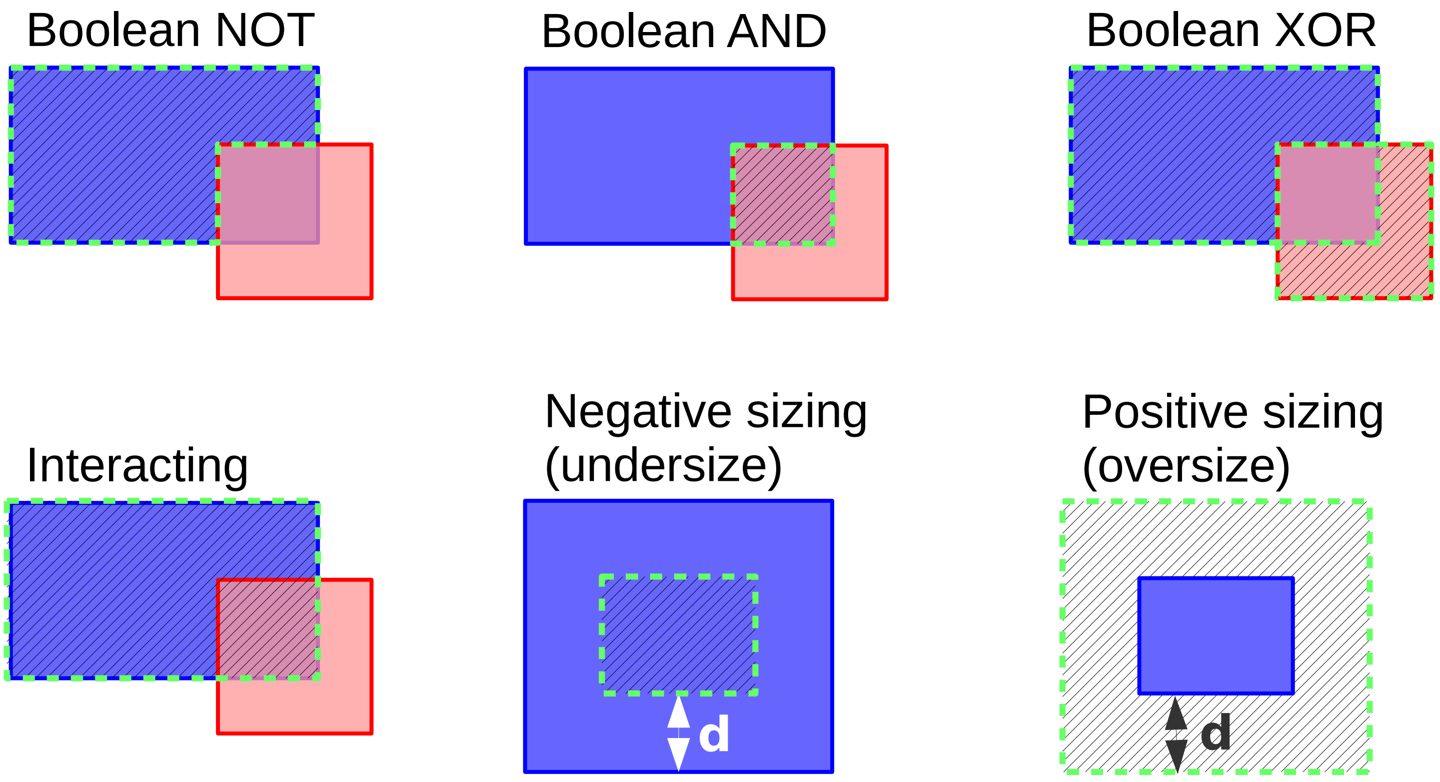
Basic DRC Rules



Layout Analysis and Preselection

- DRC rules often don't apply to raw mask data, but on
  - Combined masks → Boolean operations
  - Certain configurations → selection, interacting, ...
- DRC scripts allow computing temporary "layers" (= polygon sets) to represent intermediate geometry
- Edge layers allow representing parts of polygon borders

Some Polygon Layer Derivation Functions



More Applications of the DRC Features

- The output of DRC can be send to a new layout or back to the original layout
- → Manipulation of layout
  - Derive mask data from drawing
  - Apply technology bias (sizing)
  - Add computed content to drawn layout
- → Comparison of layouts
  - Using the XOR function
  - Apply post-XOR filter through sizing, ...

LVS Theory in a Nutshell

- The design manual describes the devices in terms of basic geometry and layer combinations
  - The LVS identifies the **devices** from their characteristic geometry
  - The LVS identifies their connection points ("**terminals**")
- The design manual describes the metal stack and further conductive layers
  - The LVS uses this information to derive the device connections from the wiring
  - The connection graph renders the **netlist**
- The netlist derived from the layout is **compared** against the design netlist to verify conformity

LVS Flow

- Preparation step
  - Derive device recognition and connecting layers
- Device recognition
  - Isolate devices
  - Identify and mark terminals
- Connectivity evaluation
- Netlist generation

Layout-to-netlist stage

- Netlist vs. netlist compare

Work in progress

Good Practice: Bottom-up Verification

- Blocks shall be LVS and DRC clean before being put together
  - Low risk of introducing new errors during combination of blocks
- Golden rule of physical implementation

DRC Hands-On

This chapter will tell you ...

How to write and run DRC scripts

How to debug them

About the elements of a DRC script

About layer types and basic functions

Example Technology

- Repo at

  https://github.com/klayoutmatthias/si4all

- Clone with git

  ```
  git clone https://github.com/klayoutmatthias/si4all.git
  ```

- Or download as zip

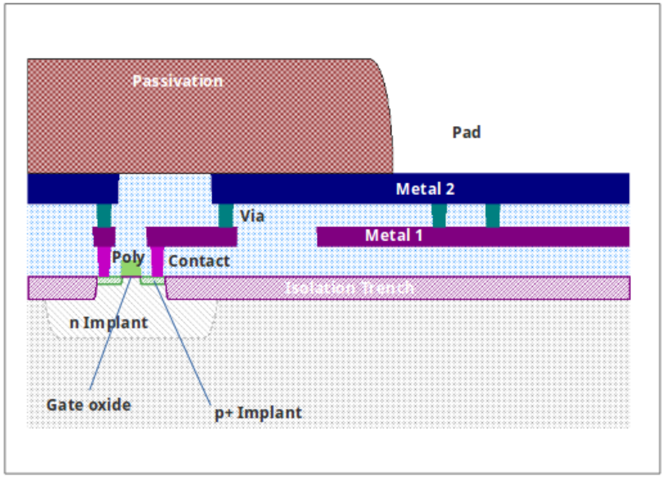
  https://github.com/klayoutmatthias/si4all/archive/master.zip

- Design manual link

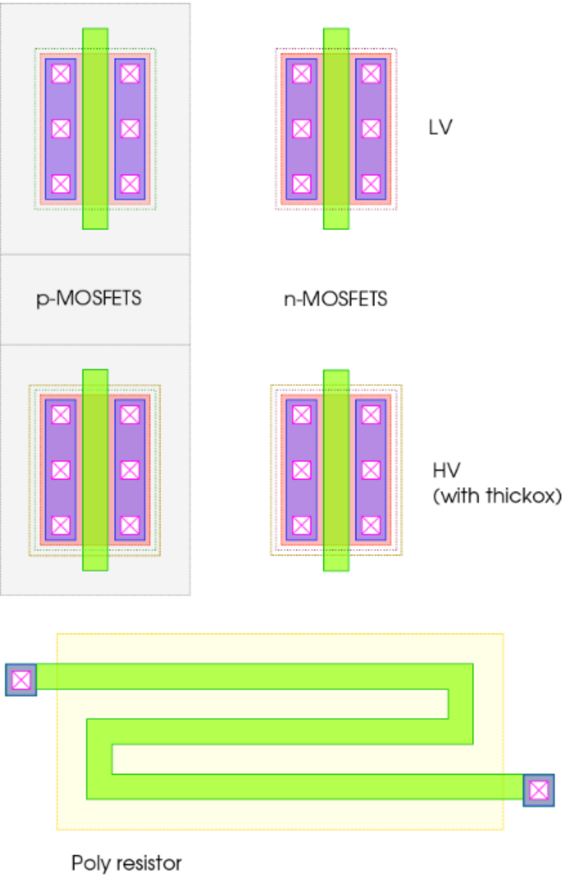
  https://github.com/klayoutmatthias/si4all/blob/master/dm.pdf

Example Design Manual

Process Description

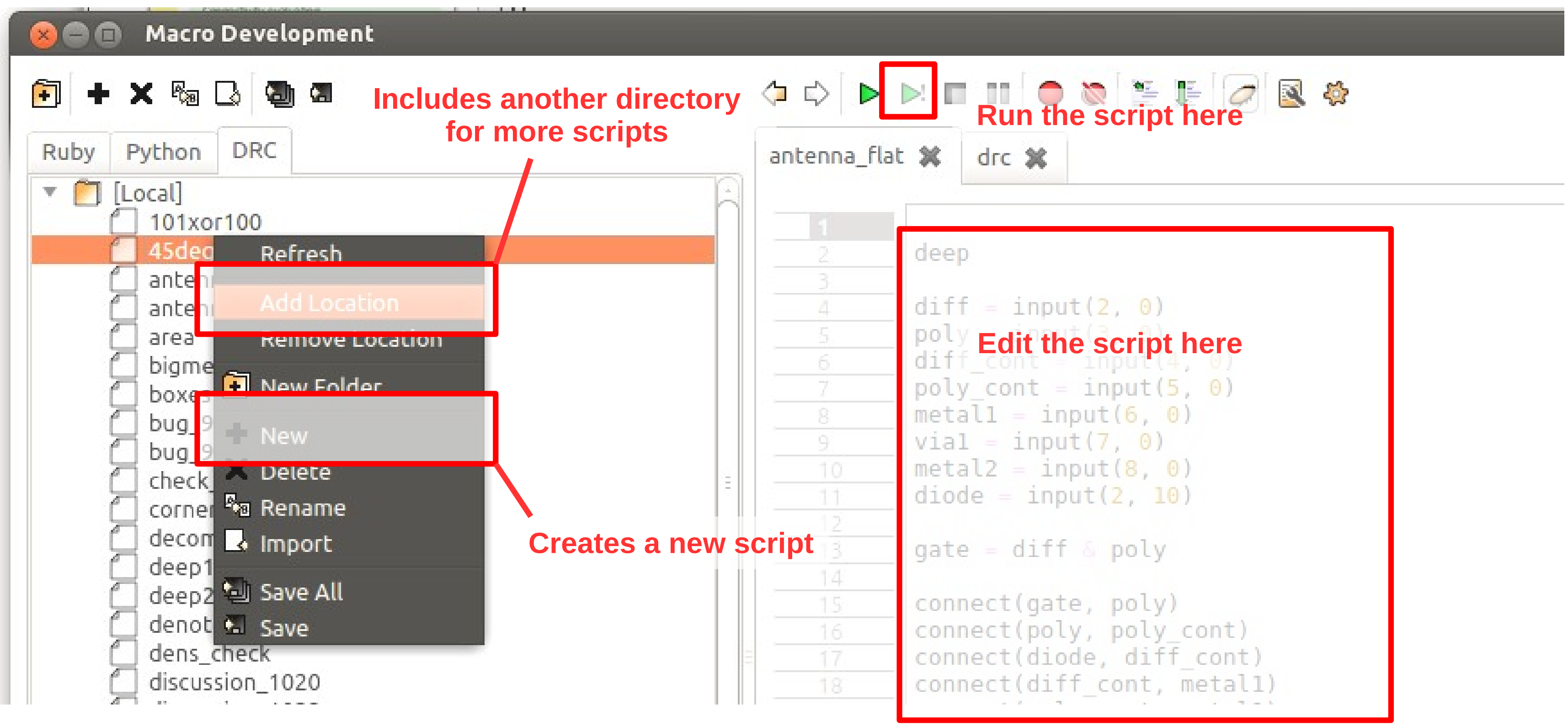


Devices



DRC scripts in KLayout

DRC scripts are written, tested and debugged in the Macro Development IDE (Tools / Macro Developmen IDE)



How to use the Examples

- Unpack zip or clone the git repo
- In KLayout use Tools / Macro Development IDE
- Chose the DRC tab
- Right-click into the script list
- Chose “Add location”
- In the file browser navigate to the “drc” folder in the sample you unpacked / cloned. Click “Ok”
- Double-click on the “drc” script to open it

DRC Script Elements: Preamble

Drawing Layers

GDS Layer Number	Layer Name	Comment
1	nwell	n implant (well)
2	diff	Diffusion (active area)
3	pplus	p+ implant marker
4	nplus	n+ implant marker
5	poly	Polysilicon
6	thickox	Thick oxide marker for high-Vt transistors
7	polyres	High resistance polysilicon marker for poly resistors
8	contact	Polysilicon or diffusion contact
9	metal1	First metal
10	via	Via between first and second metal
11	metal2	Second metal
12	pad	Pad opening
13	border	Drawing boundary

Implementation

```
report("DRC report")
```

Asks KLayout to create a marker database

```
# Drawing layers

nwell       = input(1, 0)
diff        = input(2, 0)
pplus       = input(3, 0)
nplus       = input(4, 0)
poly        = input(5, 0)
thickox     = input(6, 0)
polyres     = input(7, 0)
contact     = input(8, 0)
metal1      = input(9, 0)
via         = input(10, 0)
metal2      = input(11, 0)
pad         = input(12, 0)
border      = input(13, 0)
```

Reads the layer from the original layout from GDS layer 1, datatype 0

```
all_drawing = [
  :nwell, :diff, :pplus, :nplus, :poly,
  :thickox, :polyres,
  :contact, :metal1, :via, :metal2, :pad
]
```

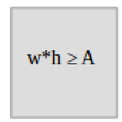
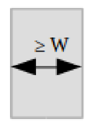
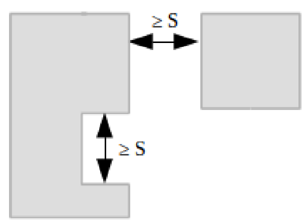
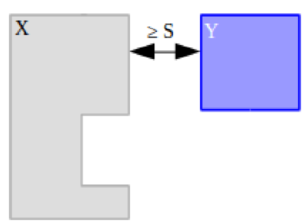
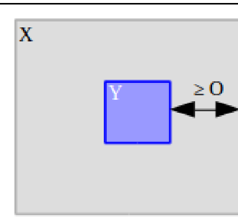
A list of variable names holding all drawing layers - needed later

DRC Script Concepts

- A DRC script is written in Ruby using special methods (a DSL)
- The basic data type is a layer
- Methods on layers
  - Manipulate layers
  - Derive new layers
- There are different types of layers:
  - Polygon layers for “filled” shapes. All original layers are polygon layers.
  - Edge layers holding edges (lines connecting two points). Edges may, but do not need to be connected.
  - Edge pair layers holding error markers (pairs of edges)
- Operations are executed in-flight, so their results can be used in conditions (if) or loops (while)

Basic Geometric Checks

Naming Scheme

Rule name	Description	Example
x_A	Min area of X **with_area**	
x_W	Min width of layer X **width**	
x_S	Min space of layer X **space**	
x_y_S	Min separation of layer X to Y **separation**	
x_y_O	Min enclosure of Y in X **enclosing**	

Example

DIFF_S	diff space	≥ 600 nm	diff space < 600 nm
DIFF_W	diff width	≥ 500 nm	diff width < 500 nm

Implementation

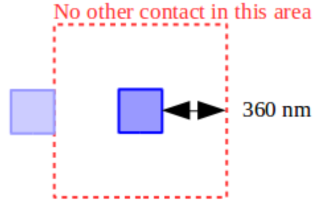
```
# ---------------------------------
# DIFF_S

min_diff_s = 600.nm

r_diff_s = diff.space(min_diff_s)
r_diff_s.output("DIFF_S: diff space < 0.6μm")

# ---------------------------------
# DIFF_W

min_diff_w = 500.nm

r_diff_w = diff.width(min_diff_w)
r_diff_w.output("DIFF_W: diff width < 0.5 μm")
```

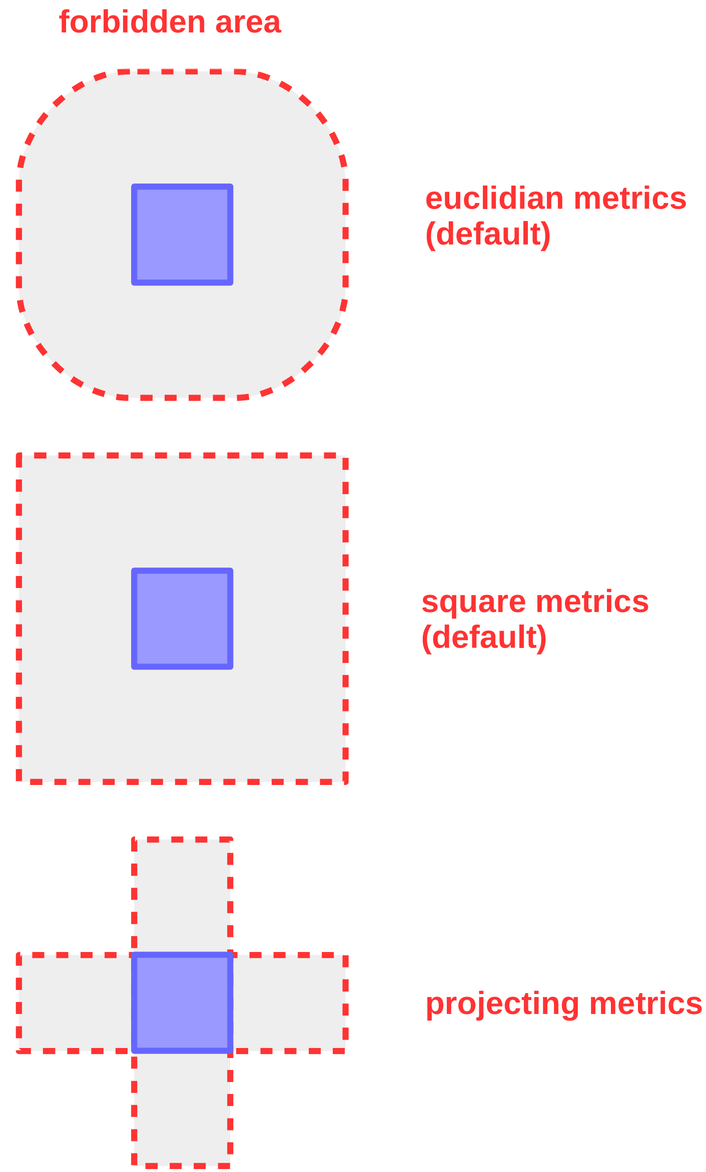
Check Anatomy

- `min_diff_s = 600.nm`
  - Stores the check target value in a variable so it can be changed easier later
  - Note the unit: 600.nm (with a dot!). Equivalent specs are: 0.6.um, 0.0006.mm. **Always use units!**
- `r_diff_s = diff.space(min_diff_s)`
  - “diff” is the original diffusion layer. “space” is the spacing check method. The space threshold is in the argument.
  - The result is an edge pair error layer that is assigned to the “r_diff_s” variable
- `r_diff_s.output("DIFF_S: diff space < 0.6µm”)`
  - Sends the error layer to the marker DB into this category

Metrics

Example

CONT_S	contact space (square metric)	≥ 360 nm	contact space < 360 nm
			

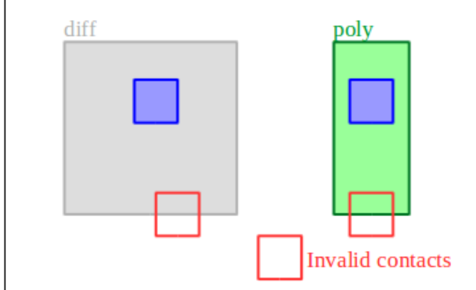


Implementation

```
# ---------------------------------
# CONT_S

min_cont_s = 360.nm

r_cont_s = contact.space(square, min_cont_s)
r_cont_s.output("CONT_S: contact space < 0.36 μm”)
```

Boolean Operations and Selectors

Example

CONT_X	contact is either inside poly or inside diff (but outside poly)		invalid contact
			

Implementation

```
# --------------------------------
# CONT_X

r_cont_x = contact -
   (contact.inside(diff) + contact.inside(poly))
r_cont_x.output("CONT_X: contact not entirely
   inside diff or poly")
```

How this works:

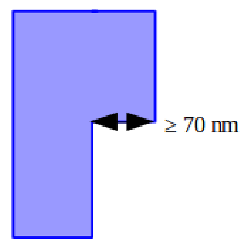
- “contact” are the original contact polygons. Same for “diff” and “poly”.
- “contact.inside(diff)” selects all contacts which are **entirely** inside diff. Same for “contact.inside(poly)”.
- “+” combines both results into one layer (boolean OR)
- “-” is the boolean NOT
- So the result layer contains all contacts which are not entirely inside diff or poly. Those are the error markers.

More Operations

- Booleans:
  - “a + b” (OR)
  - “a & b” (AND)
  - “a – b” (NOT)
  - “a ^ b” (XOR)
- Sizing (bias):
  - a.sized(d)
  - a.sized(dx, dy)
- Selectors:
  - a.interacting(b)
  - a.not_interacting(b)
  - a.inside(b)
  - a.outside(b)
  - a.overlapping(b)
  - a.touching(b)
  - a.not_inside(b)
  - a.not_outside(b)

Edge Operations

Example

POLY_X1	poly edge length	≥ 70 nm	poly edges with length < 70 nm
			

Implementation

```
min_poly_edge_length = 70.nm

r_poly_x1 = poly.edges.with_length(0,
                min_poly_edge_length)
r_poly_x1.output("POLY_X1: edge length < 0.07 µm")
```

How this works:

- “poly” are the original polygons.
- “edges” will dissolve the polygons into connecting lines.
- “with_length(0,L)” selects all edges with length between 0 and L (L itself is excluded!).
- Such edges are used as error markers to flag short edges.

More Edge Operations

- Booleans:
  - “a + b” (OR)
  - “a & b” (AND)
  - “a – b” (NOT)
  - “a ^ b” (XOR)

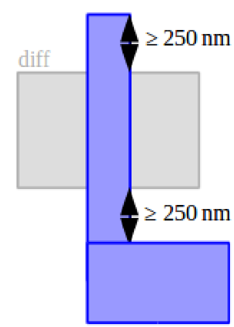
For some operations, the b operand may also be a polygon layer!

- Selectors:
  - a.interacting(b)
  - a.not_interacting(b)

- Polygonization
  - a.extended
  - a.extended_in
  - a.extended_out

Combined Operations I

Example

POLY_X2	poly extension over diff	≥ 250 nm	poly extension over gate < 250 nm
			

Implementation

```
min_poly_ext_over_diff = 250.nm

poly_edges         = poly.edges
poly_gate_edges    = poly_edges.interacting(diff)
other_poly_edges   = poly_edges.not_interacting(diff)

# ope_cd = “other poly edges close to diff”
ope_cd = other_poly_edges.separation(diff.edges,
   min_poly_ext_over_diff, projection).first_edges

r_poly_x2 = ope_cd.interacting(poly_gate_edges)
r_poly_x2.output("POLY_X2: poly extension over gate < 0.25 µm")
```

How this works:

- poly edges are decomposed into edges crossing diff (gate edges) and other edges.
- The other edges are checked against diff with “separation” with the POLY_X2 limit. From these edge pairs the poly edges are selected. poly is first argument to the **edge-wise separation**: poly is in first_edges.
- All such edges which directly connect to gate edges indicate a violation of this condition.

Combined Operations II

Example

METAL2_SW	metal2 space if at least one opponent is wide metal with width ≥ 3 µm	≥ 700 nm	metal2 space < 700 nm for wide metal2 (≥ 3 µm)

Implementation

```
# ---------------------------------
# METAL2_SW

min_metal2_s = 700.nm
min_metal2_wide_w = 3.um

narrow_metal2_markers =
    metal2.width(min_metal2_wide_w, projection)

wide_metal2 = metal2 - narrow_metal2_markers.polygons

wide_metal2_edges = wide_metal2.edges
narrow_metal2_edges = metal2.edges - wide_metal2_edges

r_metal2_sw = wide_metal2_edges
    .separation(narrow_metal2_edges, min_metal2_s)

r_metal2_sw.output("METAL2_SW: metal2 space < 0.7 µm
    for wide metal1 (>= 3 µm) to narrow/wide")
```

How this works:

- A “width” measurement creates markers for narrow metal. “projection” ensures the markers are well-formed (perpendicular to the original edges).
- The “polygons” operation will turn the markers back into polygons.
- A NOT operation forms the polygons that represent wide metal.
- Using these markers the metal2 edges are separated into edges for wide metal and narrow metal. A “separation” measurement implements the check.

Combined Operations II

Example

METAL1_X	metal1 density	≥ 20%, ≤ 80%	metal1 density < 20%

Implementation

```
# --------------------------------
# METAL1_x

min_metal1_dens = 0.2
max_metal1_dens = 0.8

metal1_area = metal1.area
border_area = border.area
if border_area >= 1.dbu * 1.dbu

  r_min_dens = polygon_layer
  r_max_dens = polygon_layer

  dens = metal1_area / border_area
  ds = '%.2f' % (dens * 100)

  if dens < min_metal1_dens
    r_min_dens = border  # use border as marker
  end
  if dens > max_metal1_dens
    r_max_dens = border  # use border as marker
  end

  r_min_dens.output("METAL1_Xa: metal1 density (#{ds}) below threshold of #20%")
  r_max_dens.output("METAL1_Xb: metal1 density (#{ds}) above threshold of #80%")

end
```

How this works:

- The “area” method computes the physical area counting multiple coverage once.
- The “border” marking layer is taken for reference. It’s area must be > 0 to avoid division by zero. The area is a float, so the compare should not be done against 0, but against a very small positive value.
- Compute the density from the area ratio and produce markers based on the results.

Global Operations

Example

GRID	Design grid	5 nm	All geometry is on-grid on a 5 nm grid

Implementation

```
# ---------------------------------
# ONGRID

grid = 5.nm

# we kept a list of all original layer’s variable
# names in “all_drawing”
all_drawing.each do |dwg|

  # a Ruby idiom to get the value of a
  # variable whose name is in "dwg" (as symbol)
  layer = binding.local_variable_get(dwg)

  r_grid = layer.ongrid(grid).polygons(10.nm)
  r_grid.output("GRID: vertexes on layer #{dwg}
    not on grid of 5 nm")

end
```

How this works:

- We kept a list of variable names (not the layers itself!) in “all_drawing” at the beginning of the file. Keeping names instead of layers means we can output their names in the message.
- This check is supposed to apply to all layers. We can iterate over all variable names, get the layer object and run the “ongrid” check.
- “ongrid” produces very small markers (dot-like edge pairs). Converting them to polygons with some minimum size (10 nm) enhances visibility.

Advanced Topics

- Raw and clean mode
  - Shapes are automatically merged by default (“clean mode”)
  - To address individual shapes, put an original layer into “raw” mode with the same method
- Tiling
  - By default, all operations will be performed on big, single sets of polygons, edges or edge pairs
  - This can lead to memory peaks
  - With tiling mode, the layout is cut into rectangular parts with a given size (one tile if the layout is smaller) and the engine works on the tiles one by one
  - This mode also supports distribution to multiple cores

More details: https://www.klayout.de/doc-qt4/manual/drc_runsets.html

DRC Batch Mode

- Two formats: .drc (plain text), .lydrc (XML)
- Plain text is easier to write with a text editor
- In batch mode, input and output files need to be specified:

```
# At the beginning of the script use:
source($input)
report("DRC report", $output)
```

  Set the “input” and “output” variables in the klayout batch mode (“-b”) call:

```
klayout -b \
  -rd input=myfile.gds \
  -rd output=drc_result.lyrdb \
  -r rules.drc
```

- To review the results:

```
klayout myfile.gds -m drc_result.lyrdb
```

- Use “verbose” to get a log, use “puts” to print your own messages

Online Resources

- Documentation links:

  https://www.klayout.de/doc-qt4/manual/drc.html

  DRC method references:

  https://www.klayout.de/doc-qt4/about/drc_ref.html

- For development master (future version):

  http://www.klayout.org/downloads/master/doc-qt5/manual/drc.html

  DRC method references:

  http://www.klayout.org/downloads/master/doc-qt5/about/drc_ref.html

- Description of KLayout’s marker DB format:

  https://www.klayout.de/rdb_format.html

Layout to Netlist and Deep Mode

This chapter will tell you ...

A little bout Deep DRC mode

A lot about layout and connectivity

Even more about layout and devices

Some preliminary information – watch for this:

This is work in progress!

Disclaimer

The Layout To Netlist feature is still under development. If you want to try it you'll need the lastet development version:

Binaries (look for version 0.26) from

http://www.klayout.org/downloads/master/

Sources from

https://github.com/klayout/klayout

Blog

https://github.com/klayout/klayout/wiki/Deep-Verification-Base

Deep Mode

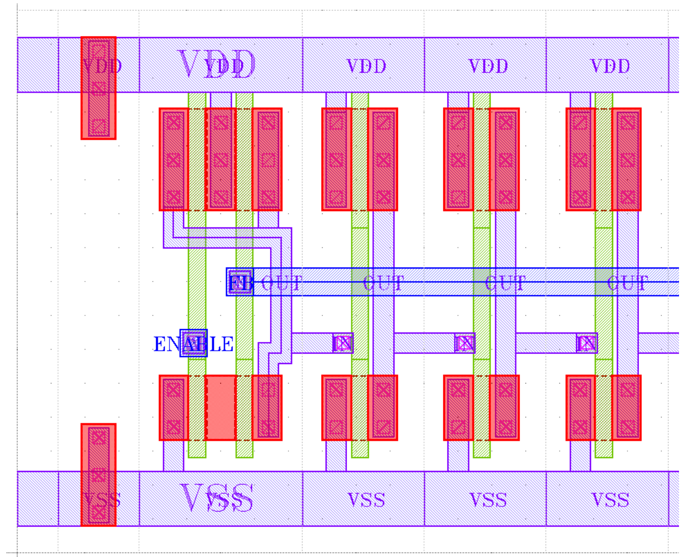
- Deep mode is a general add-on to DRC that enables hierarchical operations (development branch only)
- This mode is strongly recommended as otherwise the netlist does not have a subcircuit structure
- Deep mode is enabled by putting the “deep” statement in front of the script
- “Hierarchical mode” means: perform operations in subcells if possible.

Deep vs. Flat Mode

Flat Mode

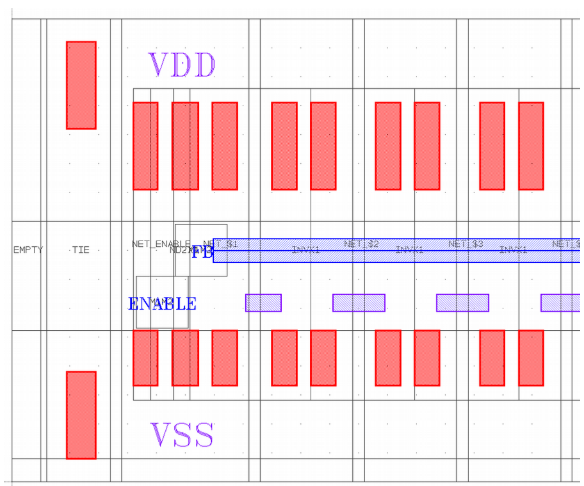
```
diff = input(2,0)
poly = input(5,0)
(diff – poly).output(...)
```

Render the same image when seen from the top cell ...



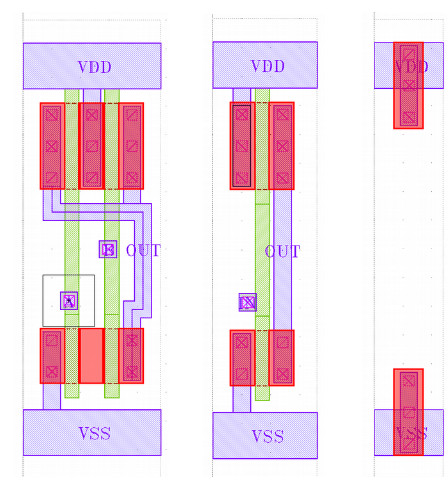
Deep Mode

```
deep
diff = input(2,0)
poly = input(5,0)
(diff – poly).output(...)
```



... but only deep mode leaves the source/drain shapes in their cells.

Devices can be assigned to these cells / circuits.



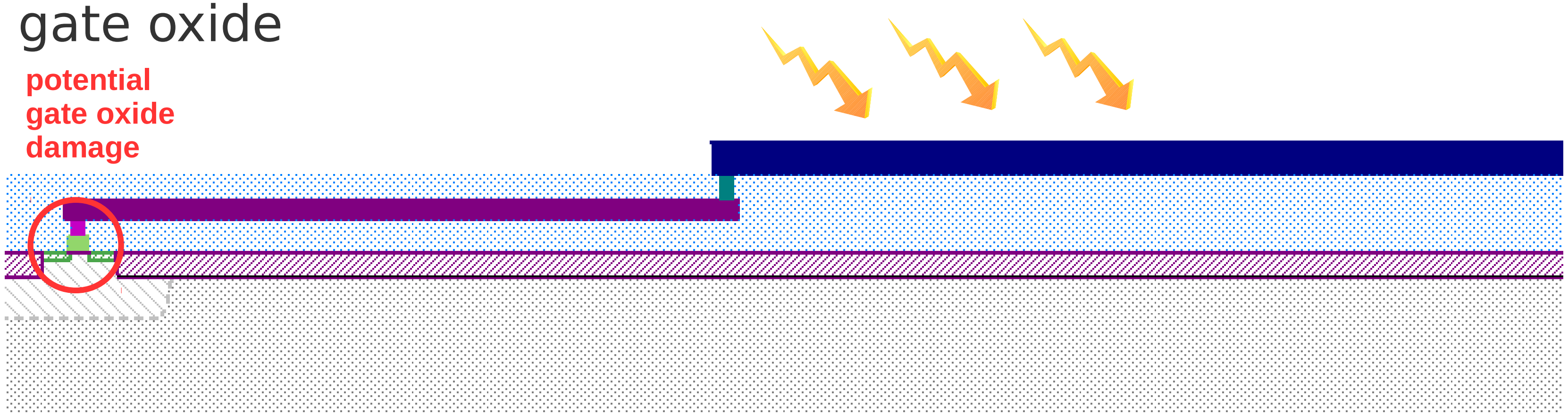
Deep Mode Limitations

- Most operations are available in deep mode, but:
  - Tiling is not compatible with deep mode
  - Currently there is no multi-core support
  - No raw mode – clean mode is implied
  - Some operations will create cell variants

This is work in progress!

DRC/Netlisting Crossover: Antenna Check

- The antenna effect is the accumulation of charge during plasma etch resulting in a potential damage or degradation of gate oxide



- The risk is measured in terms of antenna ratio (metal area / gate area) on each connected cluster. Bigger ratio = higher risk.
- For this you need to derive the net clusters – a typical netlisting act.

Steps to Transform a Layout to a Netlist

- **Device recognition**
  - Identify devices and produce markers for the device terminals. The netlister will later connect to the devices through these markers.
- **Netlisting**
  - Trace the wires through the connections made over vias and between shapes on the same layers. All connected shapes form one net.
  - Include terminal shapes of the devices and use the information from these shapes to identify the device and the terminal it is connected to this net.
  - Trace nets over hierarchy: form connections between cells. Connections between cells are called pins.
- **Netlist formation and simplification**
  - From the nets, pins and device terminals form the hierarchical netlist graph
  - Simplify the netlist by device combination, elimination of empty instances (e.g. vias) and removal of floating nets

Running The Netlister

- Most of the netlisting part is now available as a special feature set of the DRC script framework
- After the netlist has been built, you can
  - Use it inside DRC for antenna checking
  - Write the netlist to a file in Spice format
  - For the advanced user: Use the Ruby netlist API to access the netlist and the LayoutToNetlist database (links layout to nets)

Yet To Do / Plan

- A netlist viewer / browser, analogous to the DRC marker browser
- Closing the verification loop with a netlist-vs-netlist compare
- Enhanced integration of netlisting feature into a script language and provide as “LVS” feature parallel to DRC

Example (same as for DRC)

- Repo at

  https://github.com/klayoutmatthias/si4all

- Clone with git

  ```
  git clone https://github.com/klayoutmatthias/si4all.git
  ```

- Or download as zip

  https://github.com/klayoutmatthias/si4all/archive/master.zip

- Design manual link

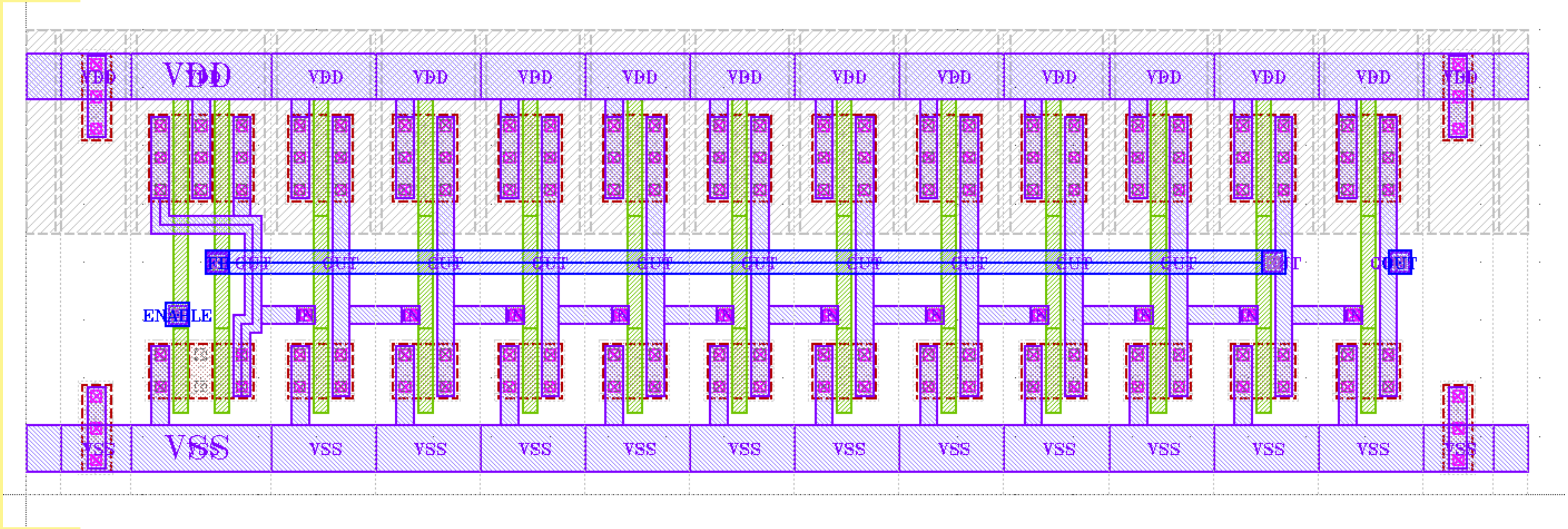
  https://github.com/klayoutmatthias/si4all/blob/master/dm.pdf

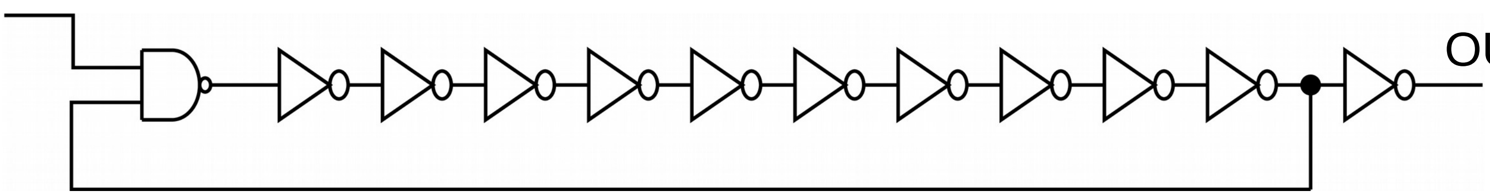
- Netlist specific samples in this package:

  Extraction Script: drc/netlist.lydrc

  Layout: ringo.gds

Example Layout





Anatomy of a Netlister Script

- Input phase
  - Fetch the original layers
- Derived layer computation
  - E.g. source/drain area is “diffusion – poly”
- Device extraction
  - Derive the device instances and place terminal markers on specific layers
- Network formation
  - Specify connections between conducting layers
- Netlist simplification (optional)
- Netlist output

Netlist Script Anatomy: Input Phase

Sample:

```
# Hierarchical extraction

deep

# Drawing layers

nwell       = input(1, 0)
diff        = input(2, 0)
pplus       = input(3, 0)
nplus       = input(4, 0)
poly        = input(5, 0)
thickox     = input(6, 0)
polyres     = input(7, 0)
contact     = input(8, 0)
metal1      = input(9, 0)
via         = input(10, 0)
metal2      = input(11, 0)
```

How this works:

- “deep” enables hierarchical mode which is recommended for netlisting (otherwise the netlist will be flat)
- “input” reads layers from the layout as in DRC
- In contrast to DRC, labels are important for netlist extraction as they add names to nets. “input” pulls both polygons and texts, although formally the resulting layer will be a polygon layer.

  To only pull polygons, use “polygons” instead of “input”. To only pull labels use “labels”.

Netlist Script Anatomy: Derive Computed Layers

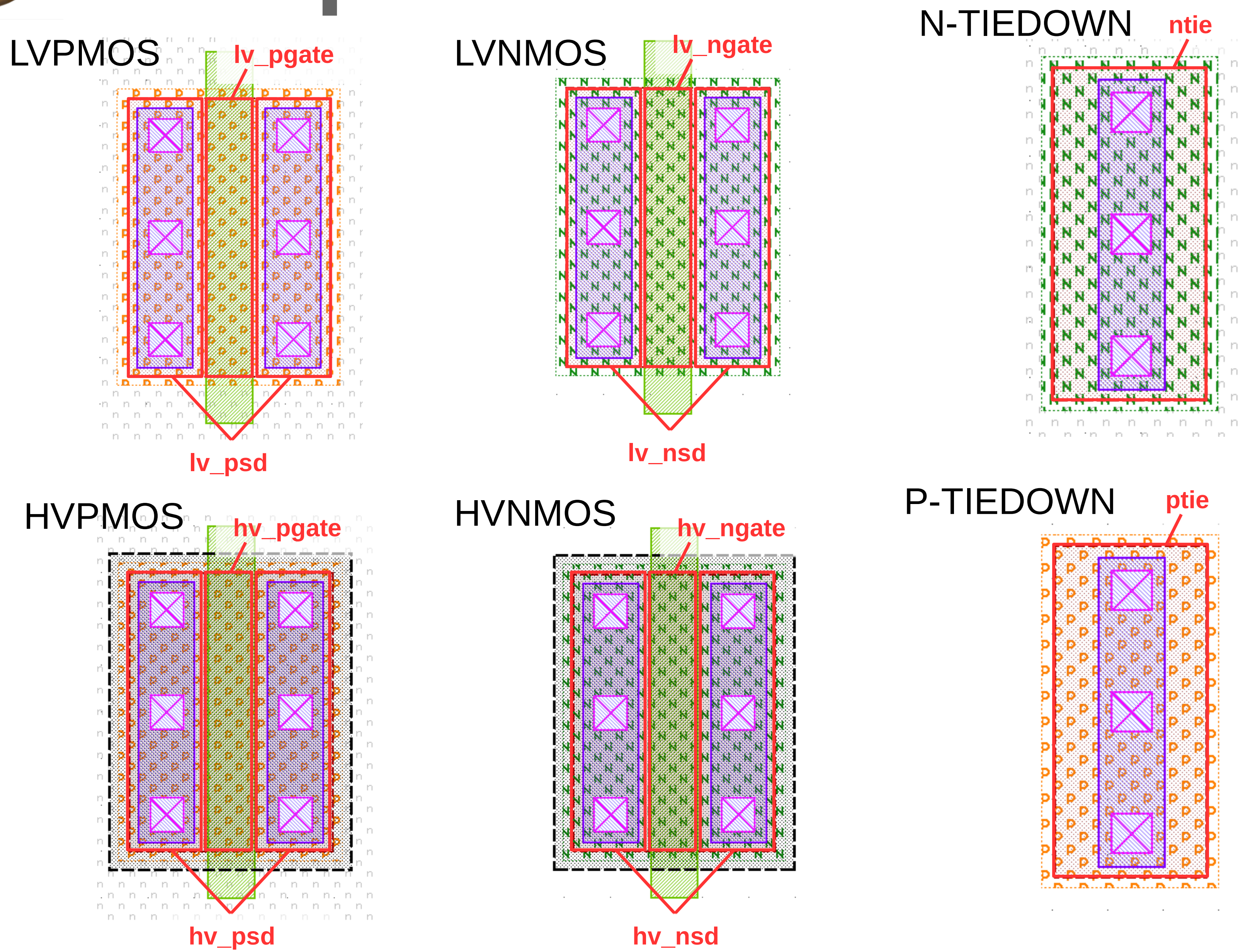
Sample:

```
bulk          = make_layer

diff_in_nwell = diff & nwell
pdiff         = diff_in_nwell - nplus
ntie          = diff_in_nwell & nplus
pgate         = pdiff & poly
psd           = pdiff - pgate
hv_pgate      = pgate & thickox
lv_pgate      = pgate - hv_pgate
hv_psd        = psd & thickox
lv_psd        = psd - thickox

diff_outside_nwell = diff - nwell
ndiff        = diff_outside_nwell - pplus
ptie         = diff_outside_nwell & pplus
ngate        = ndiff & poly
nsd          = ndiff - ngate
hv_ngate     = ngate & thickox
lv_ngate     = ngate - hv_ngate
hv_nsd       = nsd & thickox
lv_nsd       = nsd - thickox
```

How this works:

- The "bulk" layer is a virtual layer representing the substrate (bulk) of the wafer. It's required for the device extraction. There is no layout for this, so we create an empty layer with "make_layer" (don't use "polygon_layer" as this one is flat)
- The other layers are derived purely with boolean operations.

Which Layers are Computed?



Netlist Script Anatomy: Device Extraction

Sample:

```
# PMOS transistor device extraction

hvpmos_ex =
  RBA::DeviceExtractorMOS4Transistor::new("HVPMOS")

extract_devices(hvpmos_ex,
  { "SD" => psd, "G" => hv_pgate,
    "P" => poly, "W" => nwell })

lvpmos_ex =
  RBA::DeviceExtractorMOS4Transistor::new("LVPMOS")

extract_devices(lvpmos_ex,
  { "SD" => psd, "G" => lv_pgate,
    "P" => poly, "W" => nwell })

# NMOS transistor device extraction

lvnmos_ex =
  RBA::DeviceExtractorMOS4Transistor::new("LVNMOS")

extract_devices(lvnmos_ex,
  { "SD" => nsd, "G" => lv_ngate,
    "P" => poly, "W" => bulk })

...
```

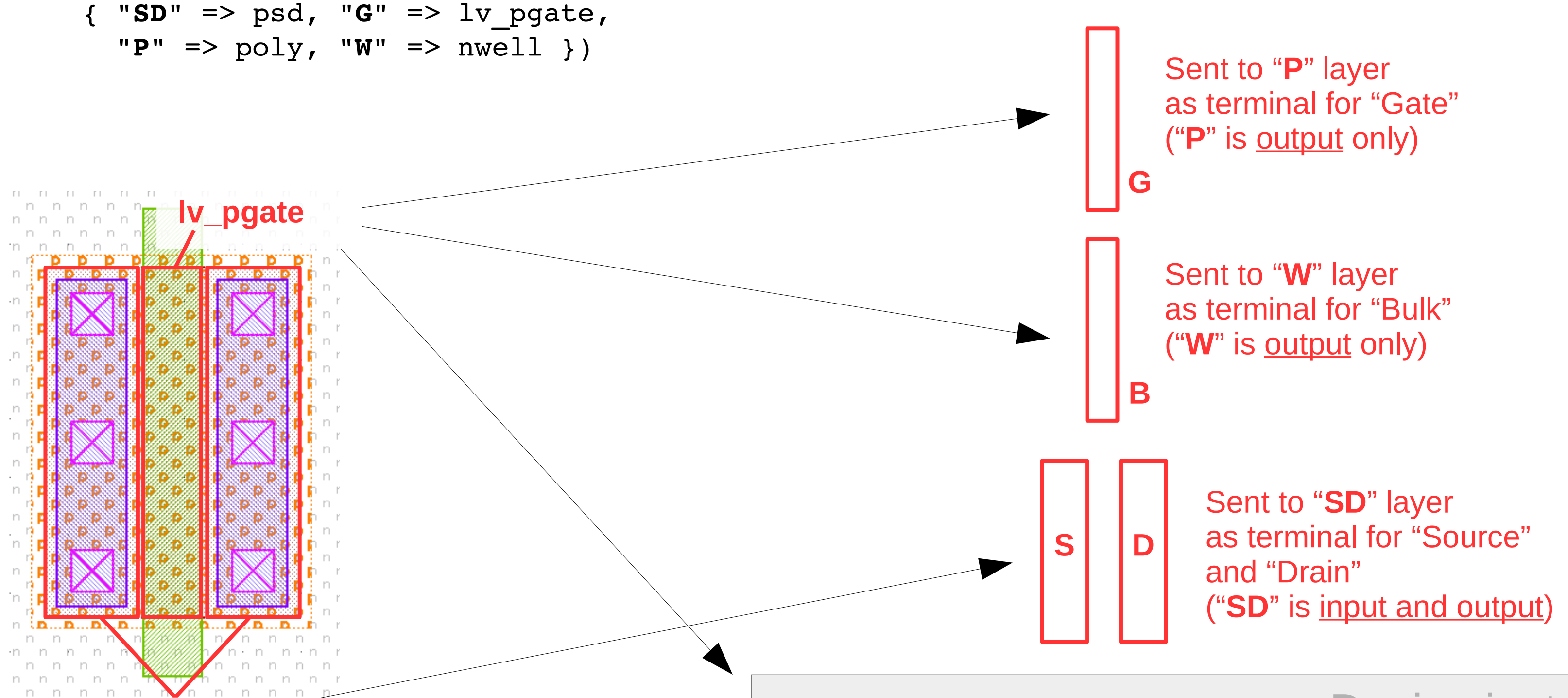
How this works:

- The device extraction is delegated to device-specific classes. “Device extractors” are instances of those classes. Some classes are provided by KLayout. More classes can be defined in Ruby code. Device extractors identify devices from shape clusters, deliver the terminal shapes and the device parameters measured from the shapes.
- “extract_devices” implements device extraction. The hash provided lists device recognition layers against device class specific symbols.

What the Device Extractor does

```
lvpmos_ex =
  RBA::DeviceExtractorMOS4Transistor::new("LVPMOS")

extract_devices(lvpmos_ex,
  { "SD" => psd, "G" => lv_pgate,
    "P" => poly, "W" => nwell })
```

NOTE: no terminals are created on the "**G**" layer – this layer is input only!



Device instance

```
M$1 S G D B MLVPMOS
+ L=0.25U
+ W=1.5U
+ AS=0.6375U AD=0.6375U
+ PS=3.85U PD=3.85U
```

Gate shape length and width

source/drain area

source/drain perimeter

Netlist Script Anatomy: Network Formation

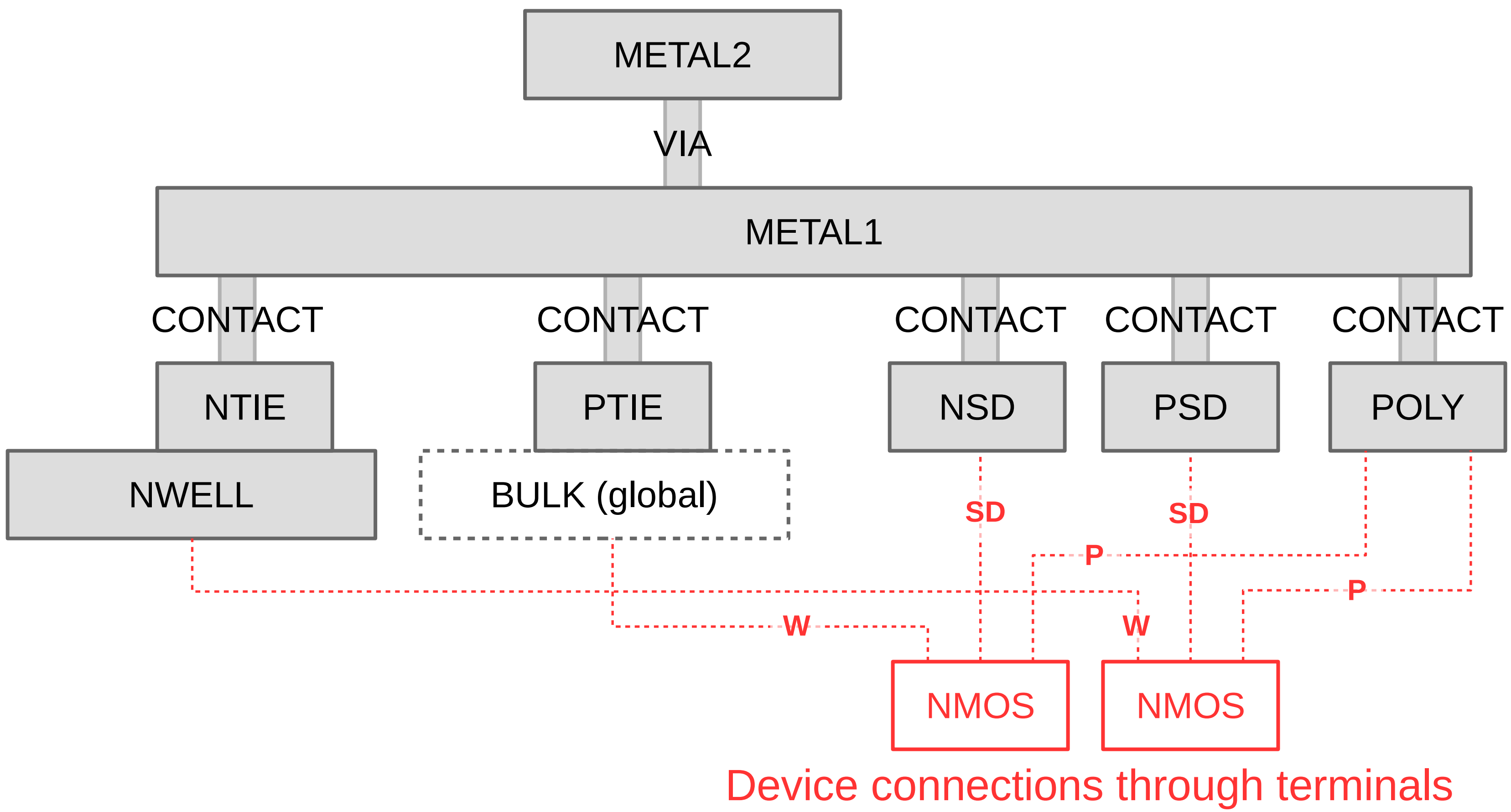
Sample:

```
# Define connectivity for netlist extraction

# Inter-layer
connect(contact, ntie)
connect(contact, ptie)
connect(nwell,   ntie)
connect(psd,     contact)
connect(nsd,     contact)
connect(poly,    contact)
connect(contact, metal1)
connect(metal1,  via)
connect(via,     metal2)

# Global connections

# ptie will make an explicit connection to “BULK”
# (the substrate)
connect_global(ptie, "BULK")

# “bulk” is the layer introduced so the n-MOS
# transistory can produce “B” terminals. These
# need to be connected to the global “BULK” net too.
connect_global(bulk, "BULK")
```

How this works:

- “connect” forms a connection between two layers. The layers can be original layers or computed layers. Labels on those layers will be taken as (hierarchy-local) net names. Pure label layers can be included as well for the purpose of assigning net names.
- “connect_global” will make connections of the named global nets. Global nets automatically make connections between across cells.
- Note that the device terminal shapes need to be included too (device extractor output layers)

Connectivity Visualized



Netlist Script Anatomy: Netlist Simplification

Sample:

```
# Compute the netlist:
# This will trigger the actual netlisting
# process!
netlist = l2n_data.netlist

netlist.combine_devices
netlist.make_top_level_pins
netlist.purge
netlist.purge_nets
```

How this works:

- “l2n_data” gives access to the Layout-to-netlist database. “netlist” is the netlist object.
- “combine_devices” creates single devices from multi-finger transisors, parallel or serial resistors etc.
- “make_top_level_pins” creates pins on the top level circuit for named nets (those with a label)
- “purge” purges all empty subcircuits (created from via cells for example)
- “purge_nets” purges all floating nets

Netlist Script Anatomy: Netlist Output

Sample:

```
writer = RBA::NetlistSpiceWriter::new

path = "ringo_simplified.cir"
netlist.write(path, writer, "Netlist comment")
```

This is work in progress!

How this works:

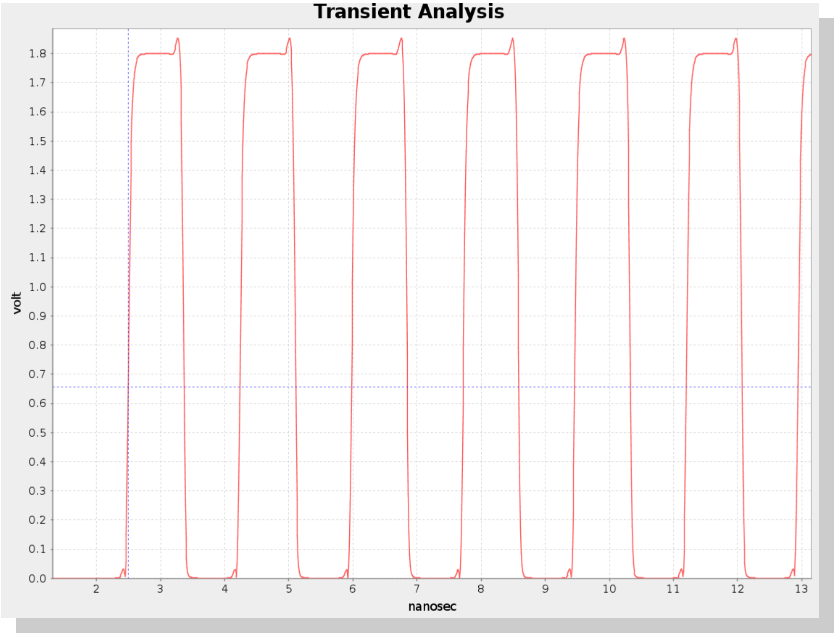
- The “NetlistSpiceWriter” class provides the spice format writer (no other format available currently)
- This object also allows customizing the output through a “delegate”
- “netlist.write” will write the netlist to the given file. The other arguments to “write” are the writer object and a comment to include in the netlist

Simulating the Netlist

- Our netlist lacks parasitic R/C elements
- We cannot expect realistic results, but still check the functionality

Testbench:

```
* 180nm models from
* http://ptm.asu.edu/modelcard/180nm_bulk.txt:
.INCLUDE "models.cir"

.INCLUDE "ringo_simplified.cir"

VDD VDD 0 1.8V
VPULSE EN 0 PULSE(0,1.8V,1NS,1NS)

XRINGO FB VDD OUT EN 0 RINGO

.TRAN 0.01NS 100NS

.PRINT TRAN V(EN) V(FB) V(OUT)
```



L2N Database

- The LayoutToNetlist database is an object providing:
  - The geometry of the nets
  - The Netlist object for the extracted netlist (schematic)
  - Serialization and deserialization (Open question: good format for this purpose?)
- It offers an API for retrieving the geometry for a given net
- The netlist itself is accessed through the Netlist API
- Further information here:

  https://www.klayout.org/downloads/dvb/doc-qt5/code/class_LayoutToNetlist.html

L2N Scripting Examples

- Retrieve the flat shapes for a certain net inside a DRC script:

```
# Gets the Circuit object for the top level cell
top_circuit = l2n_data.netlist.circuit_by_name(source.layout.top_cell.name)

# Finds the Net object for the "VDD" net
net = top_circuit.net_by_name("VDD")

# outputs the shapes for this net to layers 2000 (ntie), 2001 (ptie), 2002 (nwell)
# etc ...
{ 2000 => ntie,    2001 => ptie,
  2002 => nwell,   2003 => nsd,
  2004 => psd,     2005 => contact,
  2006 => poly,    2007 => metal1,
  2008 => via,     2009 => metal2 }.each do |n,l|
  DRC::DRCLayer::new(self, l2n_data.shapes_of_net(net, l.data, true)).output(n, 0)
end
```

This is work in progress!

L2N Scripting Examples

- Probe a net at a specific position:

```
# This is where we want to probe
probe_point = RBA::DPoint::new(10.0.um, 7.0.um)

# Looks for a net at the given point on layer metal1
net = l2n_data.probe_net(metal1.data, probe_point)

# Prints the net name to the macro development IDE console:
net && puts("Net at #{probe_point}: #{net.name}")
```

This is work in progress!

L2N Scripting Examples

- Dump all nets to cells “NET_...” with subcells “CIRCUIT_...” for the subcircuits

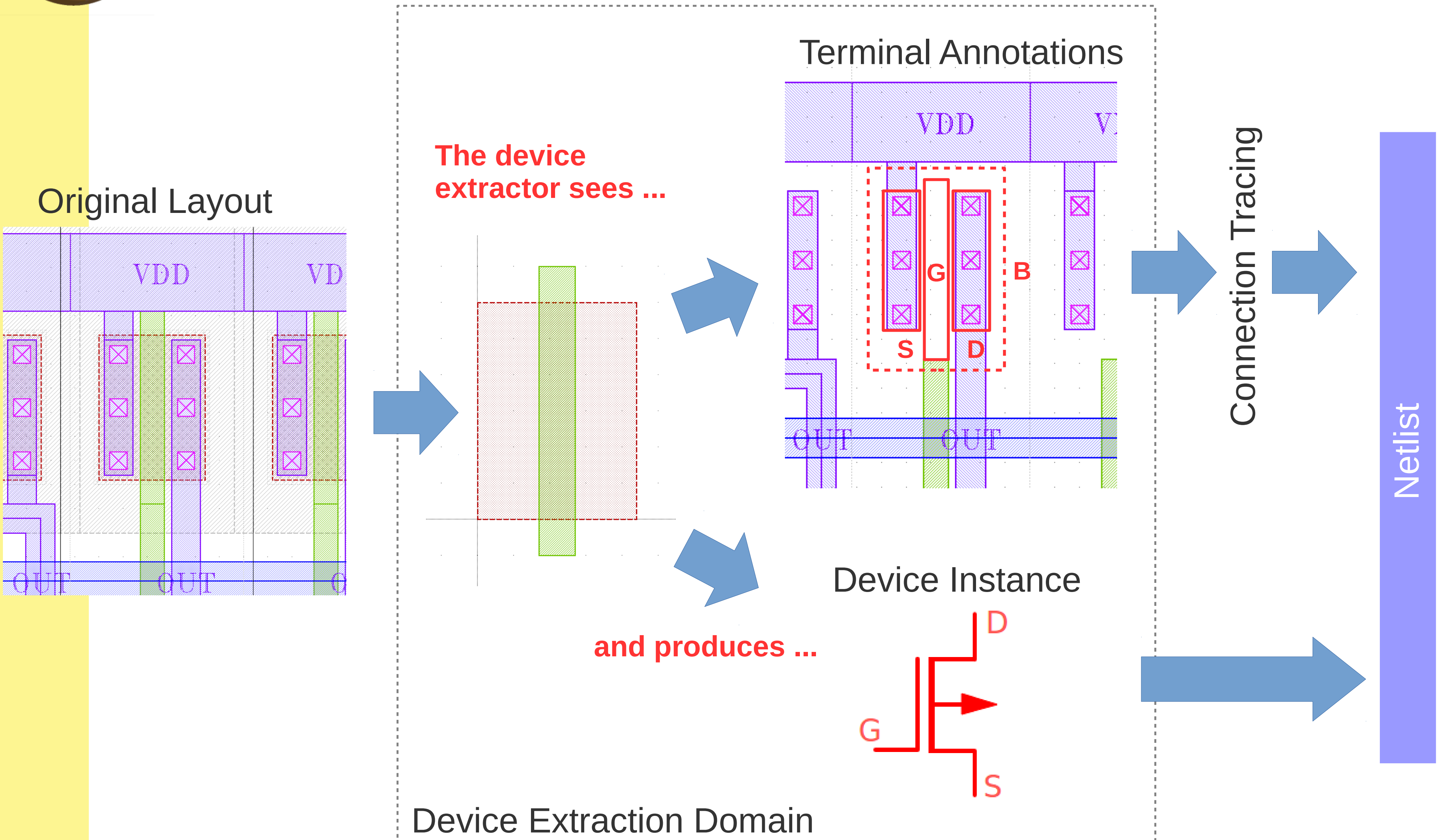
```
# build a map of layer indexes (in target layout) to layer objects
# (maps ntie to layer 2000, ptie to 2001, nwell to 2002 etc ...)
lmap = {}
{ 2000 => ntie,    2001 => ptie,
  2002 => nwell,   2003 => nsd,
  2004 => psd,     2005 => contact,
  2006 => poly,    2007 => metal1,
  2008 => via,     2009 => metal2 }.each do |n,l|
  lmap[source.layout.layer(n, 0)] = l.data
end

# builds the new hierarchy with the cells for nets and circuits
# CAUTION: this will modify the ORIGINAL layout!
cellmap = l2n_data.cell_mapping_into(source.layout, source.cell_obj)
l2n_data.build_all_nets(cellmap, source.layout, lmap, "NET_", "CIRCUIT_")
```

This is work in progress!

Custom Device Extraction

- Right now, only a simple MOS (3 or 4 terminal) device recognition scheme is provided
- More will follow, but in general the variability of devices is huge
  - e.g. capacitors come as metal plates, gate oxide caps, well capacitors, combs, fingers, ...
- So KLayout provides a flexible recognition scheme

Device Extraction Domain

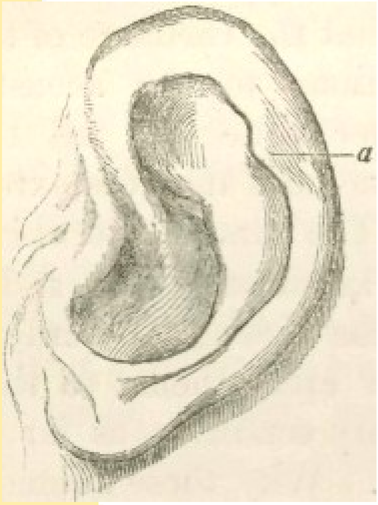


Flexibility: Device Extractor Classes

- Device extraction for a specific kind of device is delegated to a Device Extractor Class
- It is possible to implement a device extractor within DRC's Ruby code based on **RBA::GenericDeviceExtractor**
- For example see
  - Doc: http://www.klayout.org/downloads/master/doc-qt5/code/class_GenericDeviceExtractor.html
  - drc/custom_device.lydrc
- A device extractor class needs to reimplement
  - **setup** to define the layers involved
  - **get_connectivity** to define the relationship between these layers
  - **extract_devices** to turn shapes on these layers into device definitions and terminals
- **For more details please see documentation and sample code**

That’s it for now ...

Thank you
for listening!
:-)



contact@klayout.de

http://www.klayout.org