



KLayout DRC/LVS FAQ + Tutorial

Frequently Asked Questions – Explained



Prologue and Disclaimer

This is not a beginner's tutorial
about DRC or LVS with KLayout

:(

For a basic introduction see here:

<https://www.klayout.de/doc-qt5/manual/>



Structure

Demo files and source code available here:

<https://gitlab.com/klayoutmatthias/fsic-2022-demo-files/-/tree/main/klayout-talk>
(shown as "...")

Master topics

- General
- DRC
- LVS



General



Raw scripts vs. DRC / LVS

Raw scripts:

- Can be either **Python** or **Ruby**
- Act directly on the application API
- Rich capabilities (UI generation, PCell coding, Layout creation, computational geometry ..)
- API knowledge required

```
import pya

layout = pya.Layout()

top = layout.create_cell("Top")
metal = layout.layer(1, 0)

width = 30.0
height = 50.0
pitch = 80.0
angle_delta = 0.5

for n in range(0, 120):

    box = pya.DBox(0.0, 0.0, width, height).moved(-width / 2, -height / 2)
    # Only polygons can be rotated
    poly = pya.DPolygon(box)
    poly = pya.DCplxTrans(1.0, angle_delta * n, False, pitch * n, 0.0) * poly

    top.shapes(metal).insert(poly)

layout.write("discussion1749.gds")
```

<https://www.klayout.de/forum/discussion/1749>

DRC and LVS scripts:

- Use Ruby always
- Act as facade for API (A “domain specific language”)
- Simple language for verification and layout manipulation purposes
- Direct access to API is possible, but not encouraged

```
all = input(1, 0)
skew = all.edges.without_angle(0).without_angle(90)
skew.space(300.nm).without_internal_angle(45.0 .. 50.0).output(100, 0)
```

<https://www.klayout.de/forum/discussion/1896>



How to run DRC from Python?

Yes, the preferred way is through `pya.Macro`

```
1
2 import os
3
4 # simple layout generated: single cell, one layer, one rectangle
5
6 ly = pya.Layout()
7
8 l1 = ly.layer(1, 0)
9
10 top_cell = ly.create_cell("TOP")
11 top_cell.shapes(l1).insert(pya.DBox(0, 0, 1.0, 2.0))
12
13 # passes the "ly" layout object to DRC script as "$layout_for_drc"
14 # global variable
15
16 drc_interpreter = pya.InterpreterRubyInterpreter()
17 drc_interpreter.define_variable('layout_for_drc', ly)
18
19 # runs DRC script "test.drc" relative to this file
20 # -> this will modify the layout object inside DRC
21
22 pya.Macro(os.path.join(os.path.dirname(__file__), "test.drc")).run()
23
24 # shows the modified layout in the main window
25
26 pya.MainWindow.instance().create_view()
27 pya.LayoutView.current().show_layout(ly, False)
```

Demonstrates:

- Calling DRC from Python
- Sharing objects between Python and DRC (Ruby)

executes DRC script

DRC script: test.drc – adds layer 2/0

```
1
2 source($layout_for_drc, $layout_for_drc.top_cell)
3
4 input(1, 0).sized(0.5.um).output(2, 0)
```

.../python/call_drc_from_python/drc_from_python.lym



How to modularize DRC/LVS?

Method 1: the standard way “instance_eval”

```
include_file = File.join(File.dirname(__FILE__), "include_me.drc")  
instance_eval(File.read(include_file), include_file)
```

evaluates file in
DRC context

Issue: code inside included file can modify variables in calling scope, but not create new ones

```
metall = nil
```

need to define variable here, so the included file can pass the value back to the calling scope

```
include_file = File.join(File.dirname(__FILE__), "include_me.drc")  
instance_eval(File.read(include_file), include_file)
```

```
m1_space = metall.space(420.nm)
```

“include_me.drc”:

```
metall = input(33, 0)
```




How to modularize DRC/LVS?

Method 2: KLayout preprocessor

```
# %include include_me.drc
```

```
m1_space = metall.space(420.nm)
```

Included file “include_me.drc”:

```
metall = input(33, 0)
```

- This is not standard Ruby!
- KLayout text-substitutes the pseudo-comment by the included file
- Paths are resolved relative to calling file
- The interpreter's source file and line number information may not be accurate



DRC



How to check every layer?

Example: grid check on all layers

```
report("on grid")  
  
design_grid = 0.1  
  
layers each do |layer|  
  input(layer).ongrid(design_grid)  
  .output("#{layer.to_s}_on_grid", "#{layer.to_s} grid violations")  
end
```

gives you every layer that is in the source layout

Direct execution allows for dynamic scripting in contrast to table- or graph-based DRC tools:

- Loops
- Branches



How to improve speed?

1. Mode

Default Mode

- Flat polygon handling
- Single-threaded
- No overhead
- Use for small layouts
- No side effects

Tiled Mode

- Need to optimize tile size
- Finite lookup range
- Output is flat
- Multithreading enabled
- Scales with #CPUs
- Scales with layout area
- Predictable runtime and memory footprint

Deep Mode

- Preserves hierarchy in many cases
- Very fast / very slow
- Typically less memory
- Does not predictably scale with #CPU
- Performance not predictable
- Mainly used for LVS layer preparation
- Still somewhat experimental

Under development

Note that you can switch modes and tile parameters during execution!



How to improve speed?

2. Use low power alternatives

- Multichannel operations: `and_not`, `split_*`

```
diff = input(17, 0)
nwell = input(7, 0)
(ndiff, pdiff) = diff.andnot(nwell)
```

- Use sparse layers for first operands in commutable operations
 - The first operand determines the complexity
- Avoid implicit polygon merging
 - avoid huge connected regions with many holes (meshes, inverted layers)
 - raw mode avoids merging
 - boolean operations do not merge – prefer those
 - DRC functions will often merge inputs



How to improve speed?

3. More options

- Edge-mode operations may be faster than polygon operations

```
layer = input(67, 20)

layer.space(0.17).output(1000, 0)

layer.edges.space(0.17).output(1001, 0)
```

renders same result, except
shielding is not available

- Disable shielding
(https://www.klayout.de/doc-qt5/manual/drc_runsets.html#k_11)
- Disable figure breaking in deep mode
(https://www.klayout.de/doc-qt5/about/drc_ref_global.html#h2-1095)



How to improve speed?

4. Feed me samples

Debugging performance issues is a tedious and time consuming process

- Needs full access to a representative test case with layout, script and other inputs
- Synthetic test cases usually highlight the wrong problem

Sadly, real-world testcases usually cannot be shared

But:

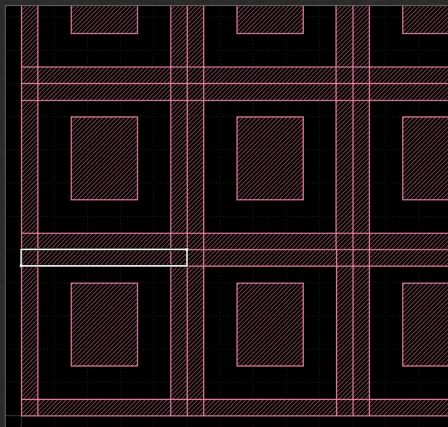
- It is often possible - with some effort - to break down a testcase into a reduced one which reproduces the problem while not disclosing secrets

Talk to your boss!

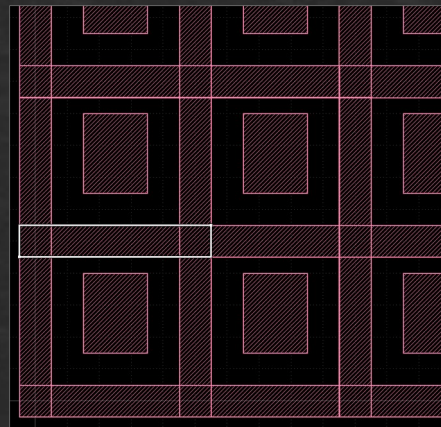


How to improve speed?

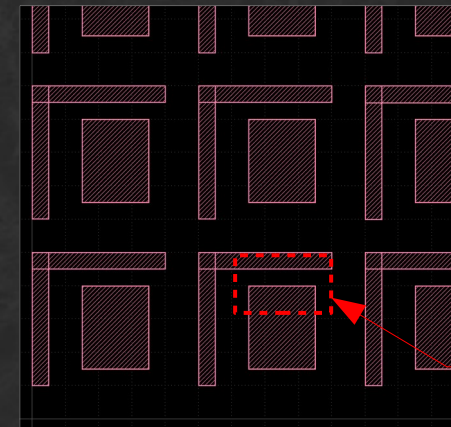
5. Core code optimization (example)



touching cells



overlapping cells



200x200
array

space
violation

separated cells

Space check	touching	overlapping	separated
tiled (10x10) 1 thread	6.10 s	6.17 s	1.02 s
deep Preliminary	~1300 s (0.27) 131 s 0.200 s *)	~1300 s (0.27) 127 s 0.210 s *)	0.04 s

*) without pre-merging: error marker duplication happens

.../drc/slow_and_fast_deep_mode



How to reduce memory?

1. Release shape memory (“forget”)

Every DRC statement is a function call, every layer is a variable holding the layer’s shapes (directly or indirectly)

DRC execution is single-pass

A layer may still be used: it cannot be released automatically

Use “forget” if you do not need the layer anymore:

```
diff = input(17, 0)
nwell = input(7, 0)
(ndiff, pdiff) = diff.andnot(nwell)
```

```
diff.forget
nwell.forget
```

releases memory for these layers:
after “forget” you can no longer use them



How to reduce memory?

2. Mode

Default Mode

- Flat polygon handling
- Worst option in terms of memory
- Use only for small layouts

Tiled Mode

- Low internal memory usage
- Results are still flat
- Flat output is not memory efficient
- Good for DRC - result is supposed to be empty :)
- Bad for computing dense intermediate layers or manipulating layouts

Deep Mode

- Temporary memory required for analysis
- Results are often hierarchical, hence memory efficient
- See notes on next slide ...

Note that you can switch modes and tile parameters during execution!



How to reduce memory?

Deep mode pitfalls

Shape propagation / flatten may occur if

- One of the operands of an operation is flat (specifically “big rectangle NOT something”)
- Hierarchical meshes get merged - e.g. as input to “size” and DRC checks
- Cell variant formation is required (e.g. grid check with off-grid instances)

Merging

- Needed for operations that need merged polygons (e.g. size)
- Happens internally
- Huge polygons may appear up in the hierarchy
- Raw mode disables merging

Shape propagation

- Deep mode considers shape interactions
- Computation happens on the hierarchy level where these interactions happen first
- Beware of flat inputs – operations will happen on top level in flat mode

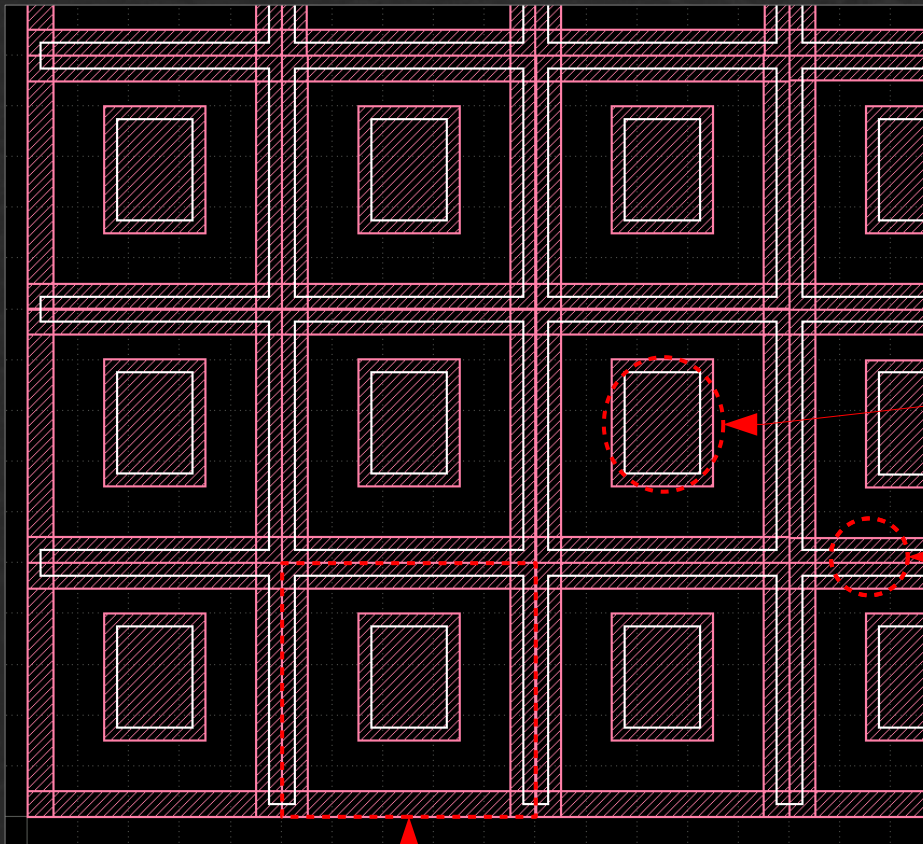
Cell variants

- Cells are duplicated and instances are reassigned
- They are needed for operations which are not translation or rotation invariant
- E.g. snapping, scaling, transformations
- Magnified instances usually create variants



How to reduce memory?

Deep mode pitfalls: merging



deep

```
layer1 = input(1, 0)  
layer1.sized(-0.05.um).output(100, 0)
```

Output (white):
hierarchical processing
single polygon inside subcell

Output (white):
huge merged polygon
(160k points) - flat

input (1/0)
200x200 stitched array of non-overlapping cells



Why doesn't KLayout read Calibre decks?

SVRF (Calibre's verification language) is protected IP, so it cannot be implemented in FOSS tools

The recommended approach is to read DRC decks from a common source (e.g. tables, Python code ..) and to supply **generators** for different target tools



What is “Universal DRC”?

“Universal DRC” is a feature of KLayout ≥ 0.27

Feature:

```
output = input.drc(function)
```

Think of “drc” as a “for each” loop on every cluster of “layer” and other inputs. “function” is a combined expression delivering shapes which are collected in “output”.

These combined expressions can involve inputs from different sources:

- “primary”: the shapes from “input”
- “foreign”: shapes from “input”, outside current cluster
- “secondary”: shapes from other layers

“function” is a combination of atomic functions, methods and operators.

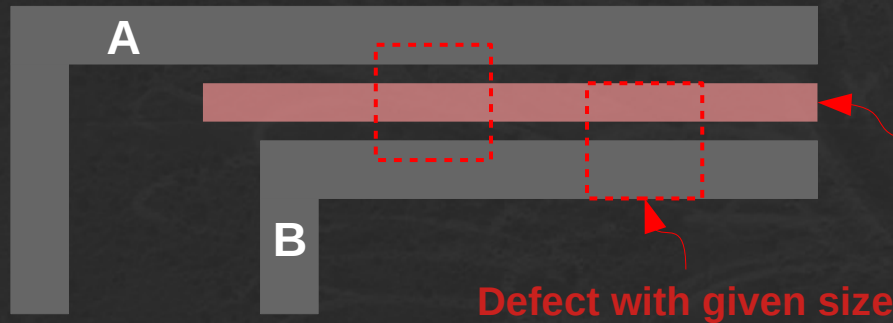
Application:

- Extended DRC checks (combined checks, more relations than “less”)
- Combined filters
- New applications using “foreign”



What is “Universal DRC”?

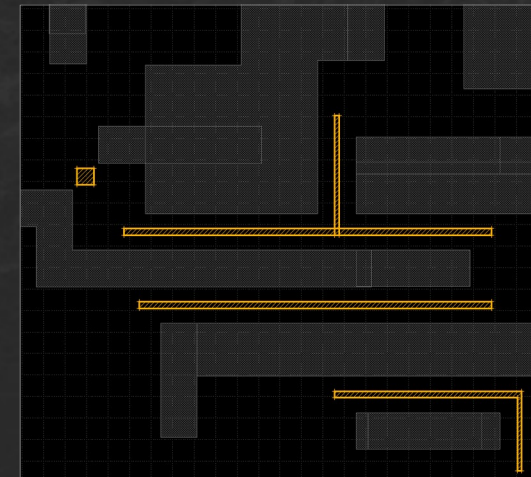
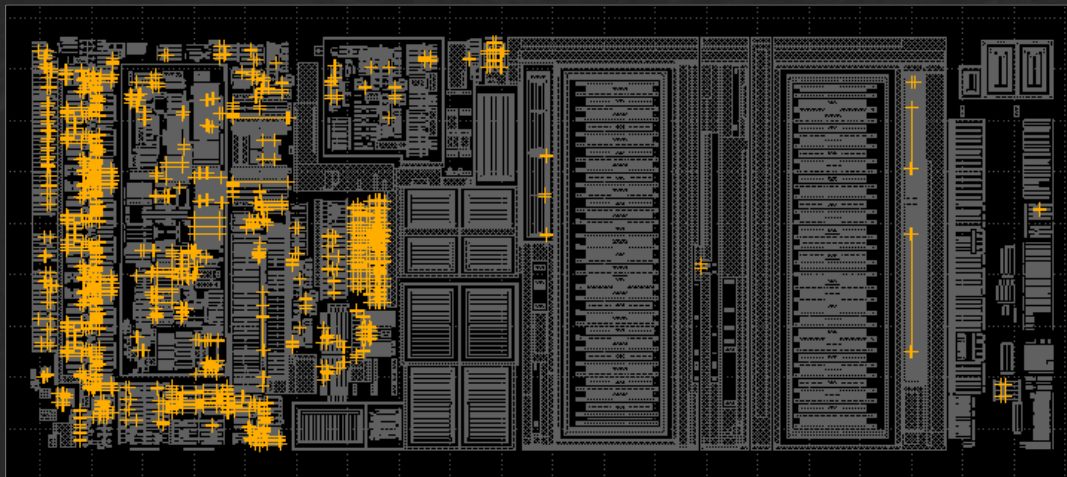
Example: “critical area”



“critical area”

A defect hitting this area with its center will short wires A and B
CA is a metric for defect sensitivity.

```
critical_area = layer.drc(primary.sized(0.5 * def_size) &  
                          foreign.sized(0.5 * def_size))
```



.../drc/universal_drc/drc.lydrc



How to implement width-dependent space checks?

```
# This assumes the following rules:  
# width < 0.25: space >= 0.4  
# width >= 0.25: space >= 0.6
```

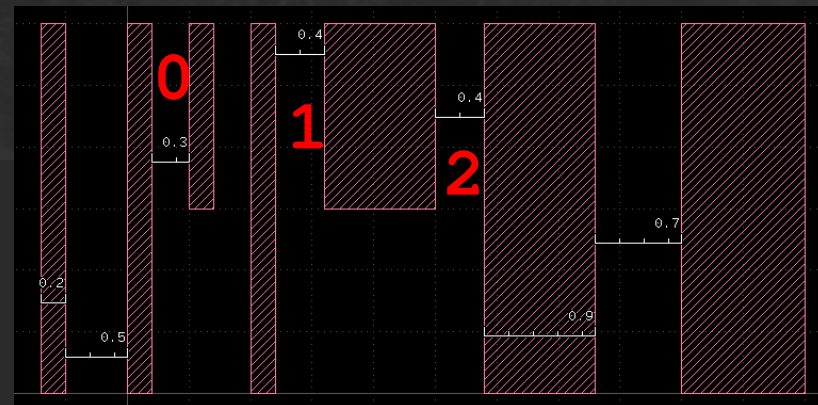
```
report("Narrow/wide check")
```

```
l1 = input(1, 0)
```

```
# any  
l1.drc(space < 0.4).output("Space < 0.4")
```

```
# wide vs. any  
l1.drc((width >= 0.25) & (space < 0.6))  
  .output("Wide (>=0.25) space to any < 0.6")
```

```
# wide vs. wide  
l1.drc(width >= 0.25)  
  .first_edges|  
  .space(0.6).output("Width (>= 0.25) space to other wide < 0.6")
```



(0)

(1)+(2)

(2)

"first_edges" will select the counter-clockwise edges suitable for "space"

../scripts/drc/.../drc/width_dependent_space



Where is “connected”?

A: It is not there (yet)

In other systems this feature allows checking for space only if the shapes are (not) connected

Implementation requirements:

- Needs a net annotation on the shapes
- Easy in flat, difficult in tiled or deep mode
- Proposal: employ the hierarchical net representation which “l2n” database provides
 - i.e. introduce space checks between **nets**

TODO



How to do a radius check?

No built-in function there, but this idea:

```
report("Radius check")
```

```
l1 = input(1, 0)
```

```
radius = 1.0
```

```
tolerance = 0.02
```

```
# generate a sharp-corner version by under/over/undersize
```

```
l1_sharp_corners = l1.sized(-radius).sized(2 * radius).sized(-radius)
```

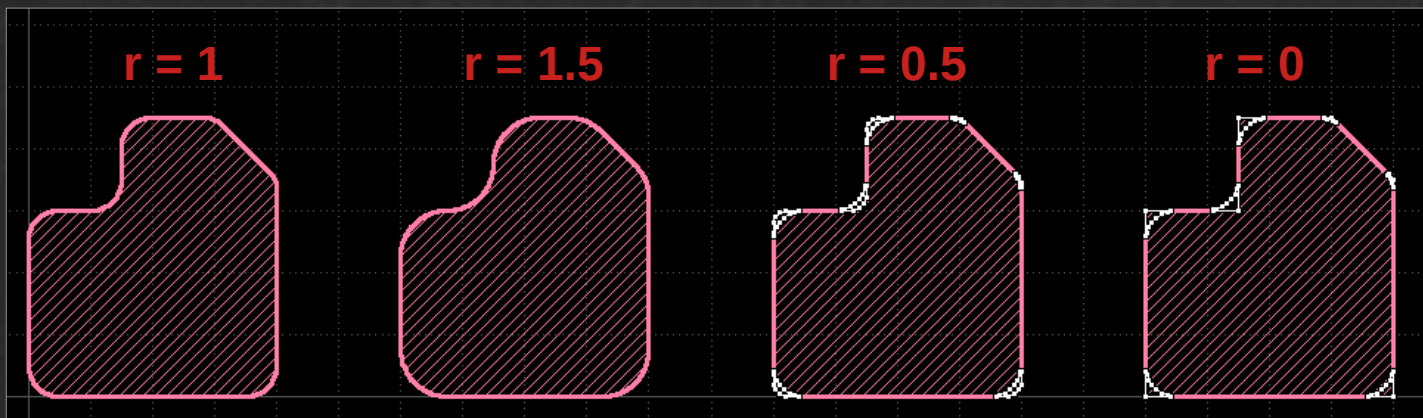
```
# apply rounded_corners to generate a reference shape and take difference
```

```
diff = (l1_sharp_corners.rounded_corners(radius, radius, 32) ^ l1)
```

```
# compare vs. actual one
```

```
diff.sized(-tolerance / 2).sized(tolerance / 2).output("errors")
```

Checks Radius $\geq 1 \mu\text{m}$



.../drc/radius_check



How to properly do an enclosure check?

There is an enclosing/enclosed feature, but it does not recognize shapes being outside the enclosing area!

Correct implementation: twofold check

```
report("Enclosure Check")
```

```
l1 = input(1, 0)
```

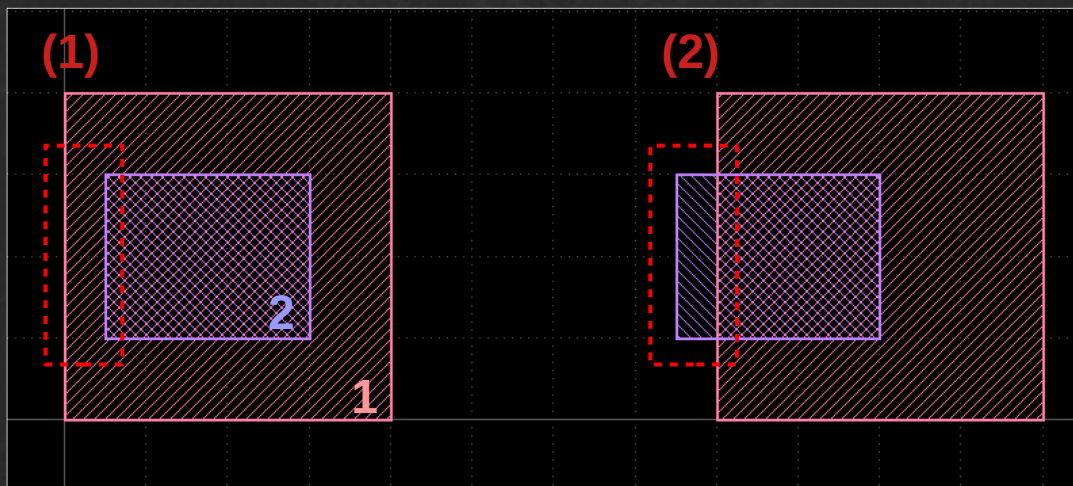
```
l2 = input(2, 0)
```

```
l2.enclosed(l1, 0.2).output("l2 inside l1 < 0.2")
```

```
(l2 - l1).output("l2 not inside l1")
```

sees (1)

sees (2)



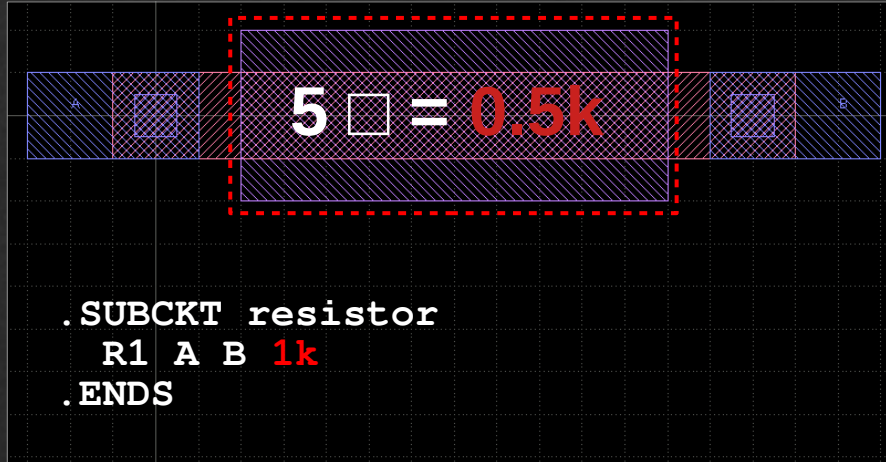
.../drc/enclosure_check



LVS



How to ignore device parameters?



```
schematic("schematic.cir")

deep

report_lvs

metal      = input(1, 0)
resistor   = input(2, 0) # resistor marker
via        = input(3, 0)
metal2     = input(4, 0)

(metal_res, metal_no_res) = metal.andnot(resistor)

extract_devices(resistor("RES", 100), { "C" => metal_no_res, "R" => metal_res })

connect(metal_no_res, via)
connect(via, metal2)

compare
```

100 Ω/□

To force a match use “tolerance” or “ignore_parameter”
https://www.klayout.de/doc-qt5/manual/lvs_compare.html#h2-136

```
connect(metal_no_res, via)
connect(via, metal2)
```

```
tolerance("RES", "R", :relative => 1.0)
```

```
compare
```

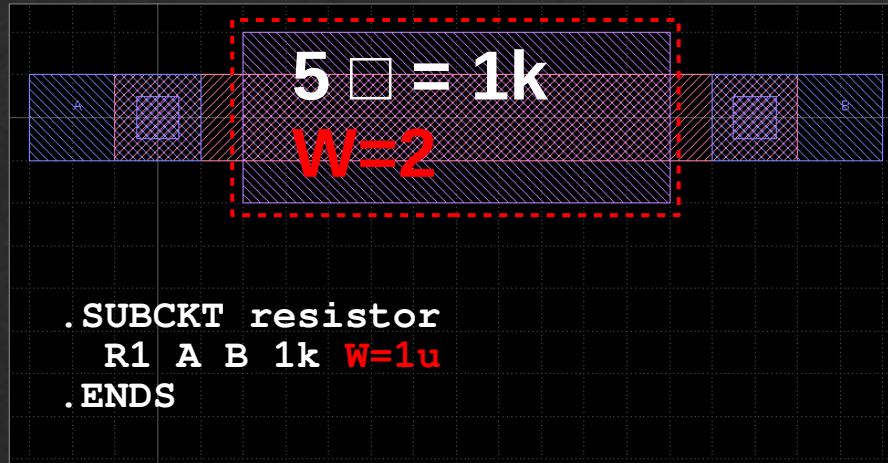
**100% tolerance =
“match always”**

RESISTOR	RESISTOR	RESISTOR
▶ ⬆ Nets		
▼ ⬆ Devices		
▶ RES	\$1 / RES [R=0.5k, L=10, W=2, A=20, P=24]	1 / RES [R=1k]

.../lvs/ignore_parameters/lvs.lylvs



How to enable device parameters?



```
schematic("schematic.cir")

deep

report_lvs

metal      = input(1, 0)
resistor   = input(2, 0) # resistor marker
via        = input(3, 0)
metal2     = input(4, 0)

(metal_res, metal_no_res) = metal.andnot(resistor)

extract_devices(resistor("RES", 200), { "C" => metal_no_res, "R" => metal_res })

connect(metal_no_res, via)
connect(via, metal2)

compare
```

200 Ω/□

“W” is a secondary parameter and not compared by default.
To force a **mismatch** use “enable_parameter”

https://www.klayout.de/doc-qt5/manual/lvs_compare.html#h2-227

```
connect(metal_no_res, via)
connect(via, metal2)
```

```
enable_parameter("RES", "W")
```

```
compare
```

Devices			
RES	/		1 / RES [R=1k, W=1]
A			A (1)
B			B (1)
RES	\$1 / RES [R=1k, L=10, W=2, A=20, P=24]	/	
A	A (1)		
B	B (1)		

.../lvs/additional_parameters/lvs.lylvs



How to write device subckts?

Spice output can be customized with a “Spice Writer Delegate”

- Delegate pattern: implement some aspects externally
- Delegates are used to redirect the flow to custom code

```
class SubcircuitModels < RBA::NetlistSpiceWriterDelegate

  def write_header
    emit_line(".INCLUDE 'models.cir'")
  end

  def write_device(device)
    str = ... # build SPICE device string
    emit_line(str)
  end

end

custom_spice_writer = RBA::NetlistSpiceWriter::new(SubcircuitModels::new)
...
target_netlist("extracted.cir", custom_spice_writer, "Extracted by KLayout")
```

Called initially to write some header

Called to write a device

- Full code: https://www.klayout.de/doc-qt5/manual/lvs_io.html#h2-37



How to read device subckts?

Spice input can be customized with a “Spice Reader Delegate”

- Similar to Spice Writer Delegate, but for reading and somewhat more complex
- Several levels of integration for tailoring the parser process
 - Atomic: translate net names
 - Spice card: parse card strings into element data
 - Devices: build devices from parsed element data
 - Subcircuits: filter model subcircuits
- Full code:
https://www.klayout.de/doc-qt5/manual/lvs_io.html#h2-146



How to make new devices?

KLayout offers some standard devices, but they may not be sufficient

Example: MOS capacitor in n well

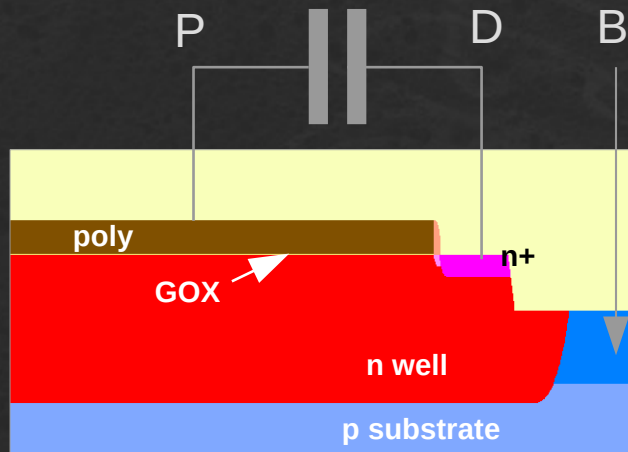
- Similar to PMOS transistor, but S/D implant is n+
- Specified with W, L instead of area
- Standard Spice C element may not be applicable
- Multiplier N instead of plain cap value adding for parallel devices

Requirements:

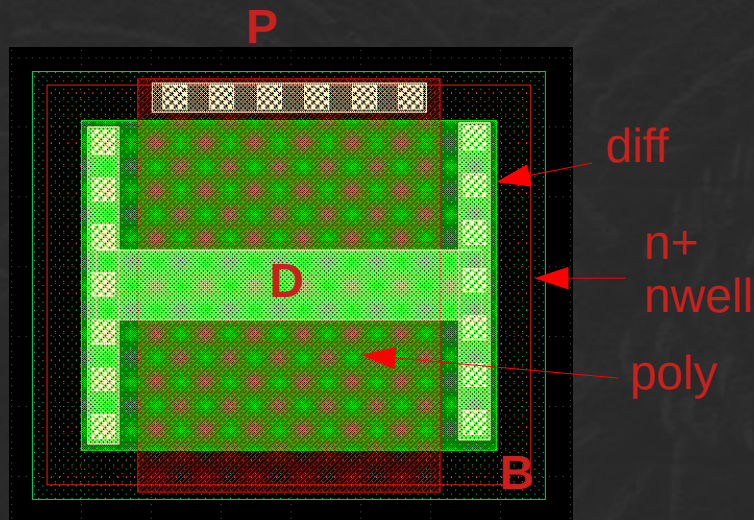
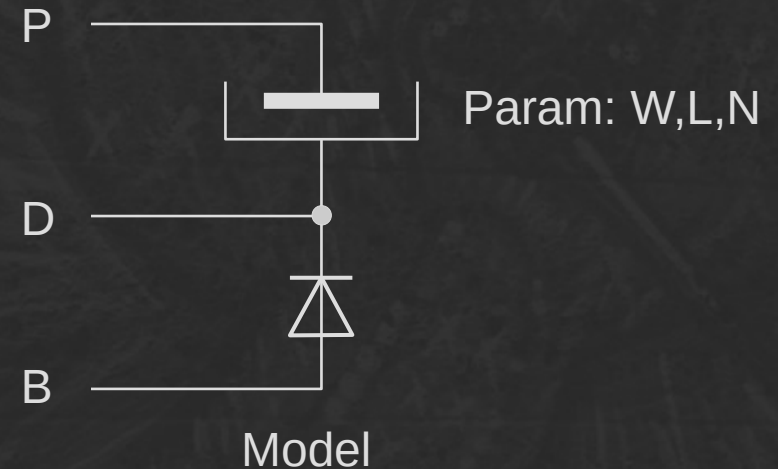
- Special extractor delivering W, L
- Spice reader / writer for using subcircuit models
- Special device combination rules



How to make new devices?



n-well MOS cap device



Layout

```
X1 G D B MOSCAPN
+ W=10u L=20u N=2
```

Spice



How to make new devices?

Full code is here: [.../lvs/custom_device](#)

Basic components:

- “Device Extractor” (defs.rb)
 - A subclass of RBA::GenericDeviceExtractor
 - Initializes the “device class”
 - Defines the geometry collector
 - Generates the devices from geometry
- “Combiner” (defs.rb)
 - A subclass of RBA::GenericDeviceCombiner
 - Implements parallel / serial combination of devices
- Spice reader and writer delegates (spice.rb)
 - Subclasses of RBA::NetlistSpiceReaderDelegate and RBA::NetlistSpiceWriterDelegate
 - Specify how devices are read or written from or to Spice files
- LVS script (lvs.lylvs)
 - The standard LVS script making use of the new devices



How to make new devices?

Device Extractor: creating the device class

The “device class” is the device “data sheet” – it specifies terminals, parameters, model name etc.

You can create one from scratch or use one of the predefined classes for a basis (enables standard Spice elements like “R”, “C” or “M”)

```
def MOSCAPNExtractor.make_device_class(name)

  dc = RBA::DeviceClass::new
  ...

  dc.combiner = MOSCAPNDeviceCombiner::new

  gate = RBA::DeviceTerminalDefinition::new("G", "Gate")
  dc.add_terminal(gate)
  ...

  dc.add_parameter(RBA::DeviceParameterDefinition::new("W", "Width", 0.0, true, 1e-6))
  ...

  return dc
end
```

name, description, default value, is_primary, si_scaling



How to make new devices?

Device Extractor: setting up ..

Reimplement “setup” to register the device class and define the extraction layers

```
def setup  
    self.name = "MOSCAPN"  
  
    # Create and register the device class  
    dc = MOSCAPNExtractor.make_device_class(@name)  
    self.register_device_class(dc)
```

```
#0 self.define_layer("A", "Cap area (for recognition)")  
#1 self.define_layer("P", "Poly")  
#2 self.define_layer("D", "Diffusion")  
#3 self.define_layer("tP", "Poly Terminal")  
#4 self.define_layer("tD", "Diffusion Terminal")  
#5 self.define_layer("tB", "Body Terminal")
```

```
end
```

input layers

output layers (terminal pins are placed there)



How to make new devices?

Device Extractor: geometry collection

The device extractor collects shapes for devices along a cluster definition based on a “connectivity” scheme. This is not electrical, but logical.

Connected shapes are clustered together.

```
def get_connectivity(layout, layout_layers)
```

```
  a = layout_layers[0]  
  p = layout_layers[1]  
  d = layout_layers[2]
```

see layer ids mentioned before

```
  conn = RBA::Connectivity::new
```

```
  conn.connect(a, a)
```

```
  conn.connect(a, d)  
  conn.connect(d, p)
```

include a self-connection to
join shapes into connected
regions

```
end
```



How to make new devices?

Device Extractor: turning geometry into device

```
def extract_devices(layer_geometry)
```

```
    (a, d, p) = layer_geometry
```

```
    a.merged.each do |a_polygon|
```

```
        a_region = RBA::Region::new(a_polygon)
```

```
        w_edges = a.edges.inside_part(d)
```

```
        w = w_edges[0].distance_abs(w_edges[1].p1) * sdbu
```

```
        l_edges = a.edges.inside_part(p)
```

```
        l = l_edges[0].distance_abs(l_edges[1].p1) * sdbu
```

```
        device = self.create_device
```

```
        device.set_parameter("W", w)
```

```
        device.set_parameter("L", l)
```

```
        self.define_terminal(device, 0, 3, a_polygon)
```

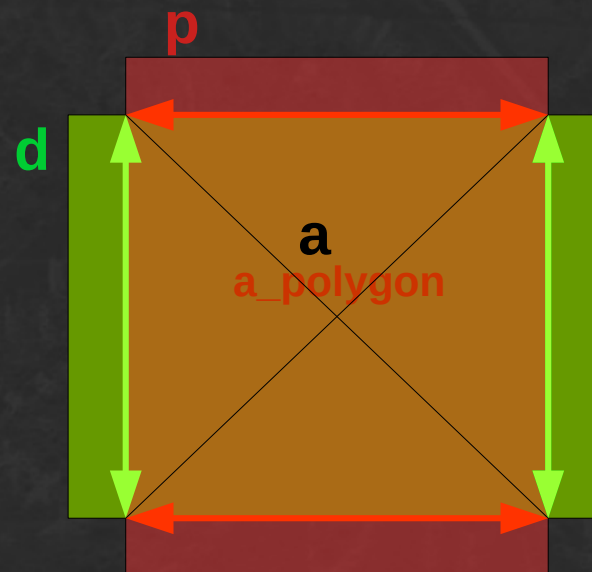
```
        self.define_terminal(device, 1, 4, a_polygon)
```

```
        self.define_terminal(device, 2, 5, a_polygon)
```

```
    end
```

```
end
```

gets called for each cluster
(potentially multiple devices)



places shapes for terminals
NOTE: terminal, layer by index (0, 1, 2 ...)



How to make new devices?

Device Combiner

The device combiner checks if devices can be combined, computes the resulting parameters and rewires the devices so that one is the combined one and the other becomes disconnected.

```
def combine_devices(a, b)
```

```
    if !same_net(a, b, "B")
        return false
    end
```

```
    if !same_parameter(a, b, "W")
        return false
    end
```

```
    a.set_parameter("N", a.parameter("N") + b.parameter("N"))
```

```
    b.disconnect_terminal("B")
    b.disconnect_terminal("G")
    b.disconnect_terminal("D")
```

```
    return true
```

```
end
```

here we only combine devices
which are geometrically identical

Adds "N" multipliers



How to make new devices?

Device Combiner

The device combiner checks if devices can be combined, computes the resulting parameters and rewires the devices so that one is the combined one and the other becomes disconnected.

```
def combine_devices(a, b)
```

```
    if !same_net(a, b, "B")
        return false
    end
    ...
    if !same_parameter(a, b, "W")
        return false
    end
    ...
```

here we only combine devices
which are geometrically identical

```
    a.set_parameter("N", a.parameter("N") + b.parameter("N"))
```

Adds "N" multipliers

```
    b.disconnect_terminal("B")
    b.disconnect_terminal("G")
    b.disconnect_terminal("D")
```

```
    return true
```

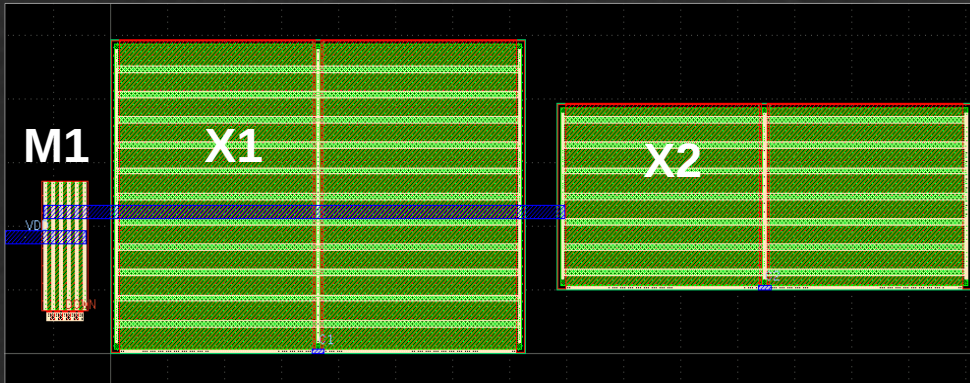
```
end
```



How to make new devices?

Demo layout and schematic: [.../lvs/custom_device](#)

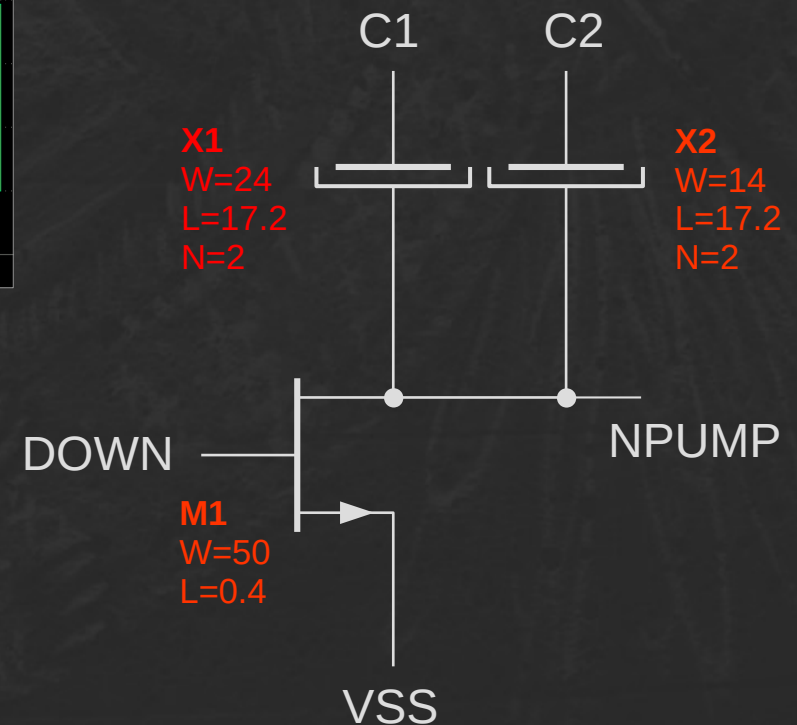
Layers taken from Sky130



mos_cap.gds

TOP	TOP	TOP
↳ Pins		
↳ Nets		
↳ S3 ↔ NPUMP	S3 (3)	NPUMP (3)
↳ C1	C1 (2)	C1 (1)
↳ C2	C2 (2)	C2 (1)
↳ DOWN	DOWN (2)	DOWN (1)
↳ SUBSTRATE ↔ GND	SUBSTRATE (4)	GND (3)
↳ VSS	VSS (2)	VSS (1)
↳ Devices		
↳ MOSCAPN	S8 / MOSCAPN [W=24, L=1 / MOSCAPN [W=24, L=17.2, N=2]	
↳ MOSCAPN	S6 / MOSCAPN [W=14, L=2 / MOSCAPN [W=14, L=17.2, N=2]	
↳ NMOS	S1 / NMOS [L=0.4, W=50, 1 / NMOS [L=0.4, W=50]	

lvs.lylvs



schematic.cir



Looking for more?

Your community:

<https://www.klayout.de/forum/>

Your documentation source:

<https://www.klayout.de/doc-qt5/manual/>



Thank you for listening!