AUTOMATED DRC ERROR SOLVER FOR LVS CLEAN LAYOUTS

A Dual Degree Project Report

Submitted by

Vishwajeet Anand

In partial fulfilment of requirements
for the award of the degree of

BACHELOR OF TECHNOLOGY

&

MASTER OF TECHNOLOGY

DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS
CHENNAI-600036

JULY 2021

THESIS CERTIFICATE

This is to certify that the thesis entitled **Automated DRC Error Solver for LVS Clean Layouts** submitted by **Vishwajeet Anand (EE16B128)** to the Indian Institute of Technology Madras, in partial fulfilment for the award of the degree of **Bachelor of Technology and Master of Technology,** is a bona fide record of research work carried out by him under my supervision. The contents of this thesis, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. Janakiraman Viraraghavan
Research Guide
Associate Professor
Department of Electrical Engineering
Indian Institute of Technology Madras
Chennai – 600 036.

Place: Chennai

Date: July 2021

ACKNOWLEDGEMENTS

I would like to express my earnest gratitude to my guide **Dr. Janakiraman Viraraghavan** for providing me the opportunity to work under his guidance. I am grateful to him for providing his extremely valuable inputs, insights and feedback on my work and for his time and support throughout the project.

A special thanks to **Manda Sashank**, who was my fellow associate in doing this project. This project would not have been possible without his contributions and insightful observations.

ABSTRACT

Keywords: Design Rule Check (DRC), Layout v/s Schematic (LVS)

The Modern layouts are large and complex. Making those free of DRC (Design Rule Check) and LVS (Layout v/s Schematic) violations can be a cumbersome process. Especially, making a circuit free of DRC violations can be a complex and very repetitive process thus making it very time consuming. Large layouts consist of many small layout blocks which needs to be connected through metal interconnects and these are the connections that cause most of the DRC violations.

This DRC cleaning process can be automated by using an algorithm that tries to solve as many of these violations as possible without causing any new violations, thus not only saving a lot of time but also saving a lot of effort that goes into solving these violations manually.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

IITM	Indian Institute of Technology Madras
VLSI	Very Large System Integration
DRC	Design Rule Check
LVS	Layout v/s Schematic
GDSII	Graphic Design System II
IC	Integrated Circuit
SSI	Small Scale Integration
MSI	Medium Scale Integration
LSI	Large Scale Integration
CalibreLV	Calibre LITHOview
GUI	Graphical User Interface

CHAPTER 1

INTRODUCTION

1.1 Motivation

The number of components fitted into a standard size IC represents its integration scale. We started off with SSI which consisted of less than 100 components or around 10 gates. Then we advanced to MSI which consisted of less than 500 components and have less than 100 gates. We further advanced to LSI where the number of components was between 500 and 300000, while having less than 1000 gates. Currently we are in the VLSI phase which consists of more than 300000 components and less than 10000 gates.

A design rule is a geometric constraint imposed on circuit board, semiconductor device, and integrated circuit (IC) designers to ensure their designs function properly, reliably, and can be produced with acceptable yield. If design rules are violated the design may not be functional. A Design Rule Checker checks all the combination of geometries present in the design for possible errors from the deck of design rules defined in the technology and returns error polygon coordinates which shows the exact location of the violations. With the advancement in technology the density of components is increasing rapidly while the die size and gate length are decreasing, thus making it increasingly difficult to solve any DRC violations manually.

The picture shown below is a zoomed part of an actual layout which is LVS clean. There were 21,000 errors in that layout. The layout has dimensions a little under 1330μm x 7.8μm and there are at little over 67,000 transistors in the layout. It takes 15 hours to manually fix the DRC errors. So this is a real life scenario and it is evident that making a layout free of DRC violations is very cumbersome and time consuming process.

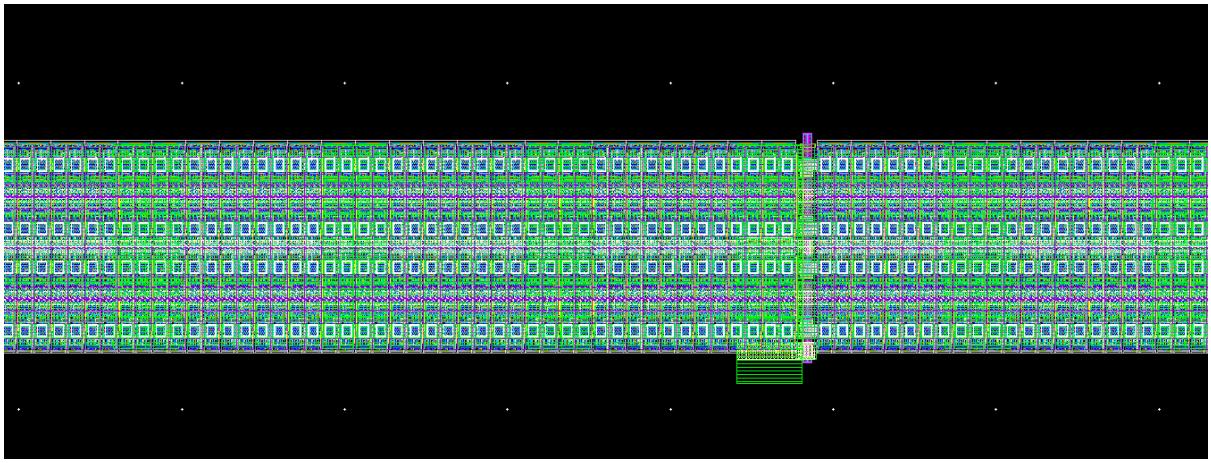

Figure 1.1 Zoomed in part of an LVS clean layout

1.2 Objective of the work

To automate the DRC cleaning process by using an algorithm that tries to solve as many of these violations as possible without causing any new violations, thus not only saving a lot of time but also saving a lot of effort that goes into solving these violations manually.

The algorithm is also expected to work with design created in any environment and also irrespective of the technology scale. The program is also expected to work with any design rules. It is also expected to maintain the LVS cleanliness after solving the errors.

1.3 Assumptions

- The layout is LVS clean
- The basic building blocks are DRC clean (i.e., poly-Si to poly-Si errors, notch errors etc. are not present)
- It is assumed that errors occur during the routing step, which means only metal to metal errors and metal to contact errors are considered.
- The error is only caused by higher metals i.e. above Metal 3.
- Each metal layer can only run in one direction, either horizontal or vertical. No metal layer should run at an angle.

CHAPTER 2

Classification of Errors

In order to solve the DRC errors, they need to be identified first. Electric Circuit Simulator has been used till now to identify these errors. Now these errors should be classified into different categories so that they can be solved in a step wise manner. Various test cases have been used for this purpose so as to cover every possible DRC violation.

General idea is that, after running the DRC, a metal layer or contact that is responsible for causing the error is selected and is moved by the smallest possible unit of distance (which in case of Electric circuit simulator is 0.5 units) in X direction. After that the error message is checked and again the same metal layer is moved by the same amount in the Y direction and the error message is checked again. These two steps will help in classifying the basic errors.

Preference for selecting error causing layer

- Metal layer > Metal to Metal contact > Metal to Poly contact > Transistor contacts (p-act, n-act)
- When an error block is moved through certain steps in the right direction, it is assumed that it does not cause any new error. If it causes any new error, then we select the other error causing layer and if that also creates some new errors then the original error is considered unsolvable (re-routing should be done).

2.1 X-axis Errors.

X-axis errors are those errors for which the error can be solved by moving one of the error causing block in +ve X direction and also by moving the other block in –ve X direction.

Finding an X-axis error:

- The error causing layer is moved by 0.5 units along +ve X axis. The DRC is run and the changes in the error message are observed.
- Next the block is moved by 0.5 along +ve Y axis. The DRC is run and the changes in the error message are observed. No change in the error message.
- The error can be solved by moving the block along X axis by required amount in the correct direction.

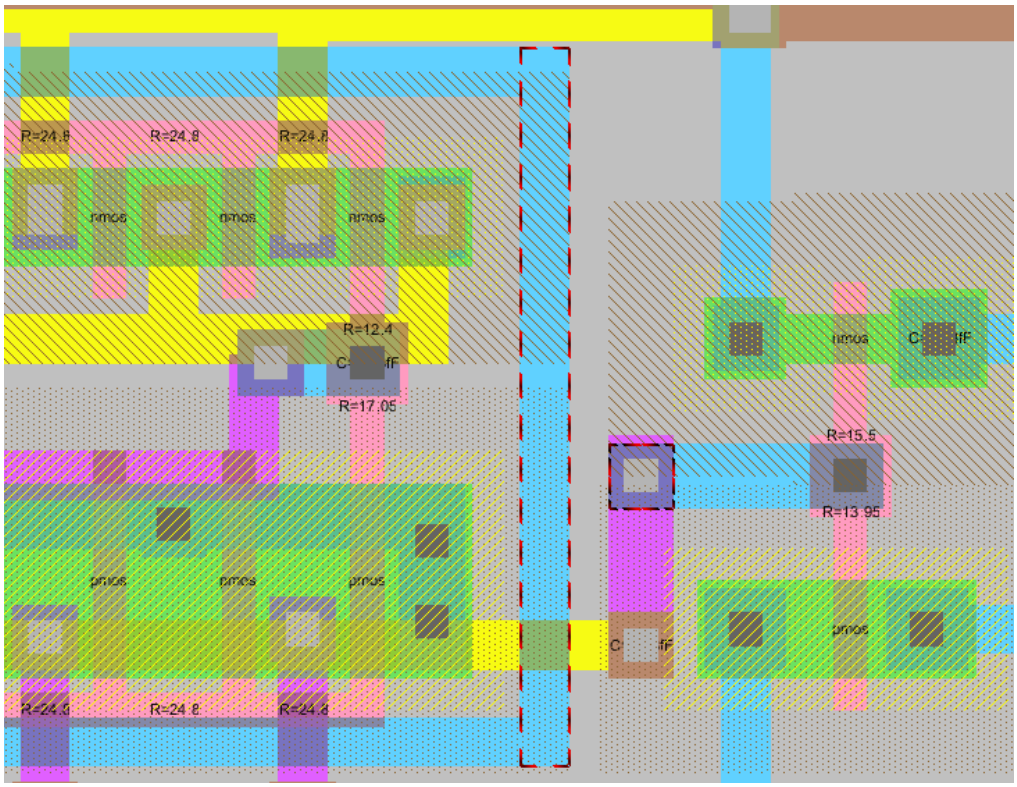

Figure 2.1: X-axis error

2.2 Y-axis Errors

Y-axis errors are those errors for which the error can be solved by moving one of the error causing block in +ve Y direction and also by moving the other block in –ve Y direction.

Finding an Y-axis error:

- The error causing layer is moved by 0.5 units along +ve Y axis. DRC is run and the changes in the error message are observed.

- Next the block is moved by 0.5 along +ve X axis. DRC is run and the changes in the error message are observed. No change in the error message.
- The error can be solved by moving the block along Y axis by required amount in the correct direction.

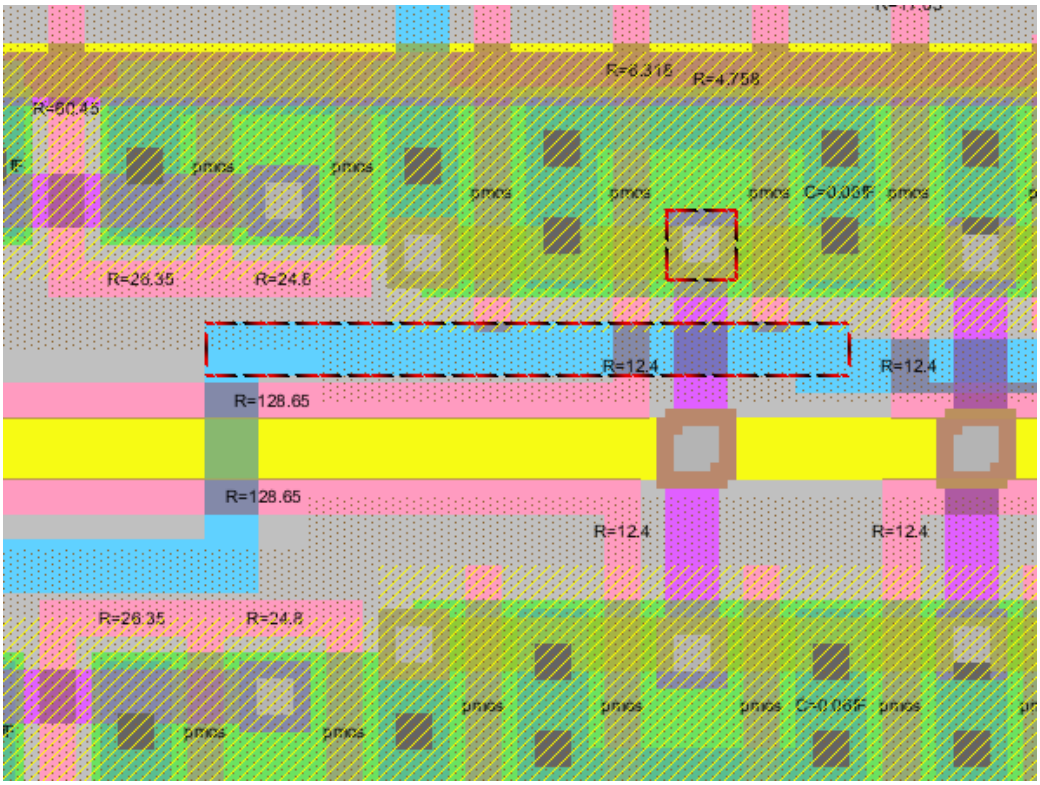

Figure 2.2: Y-axis error

2.3 Special Errors

Special errors include XY-axis errors in which the error block needs to be moved in both X & Y axis to solve the error.

Finding a special error:

- The error causing layer is moved by 0.5 units along +ve X axis. DRC is run and the changes in the error message are observed.
- Next the block is moved by 0.5 along +ve Y axis. DRC is run and the changes in the error message are observed.

- The error can be solved by moving the block along both X, Y axis by required amount in the correct direction.

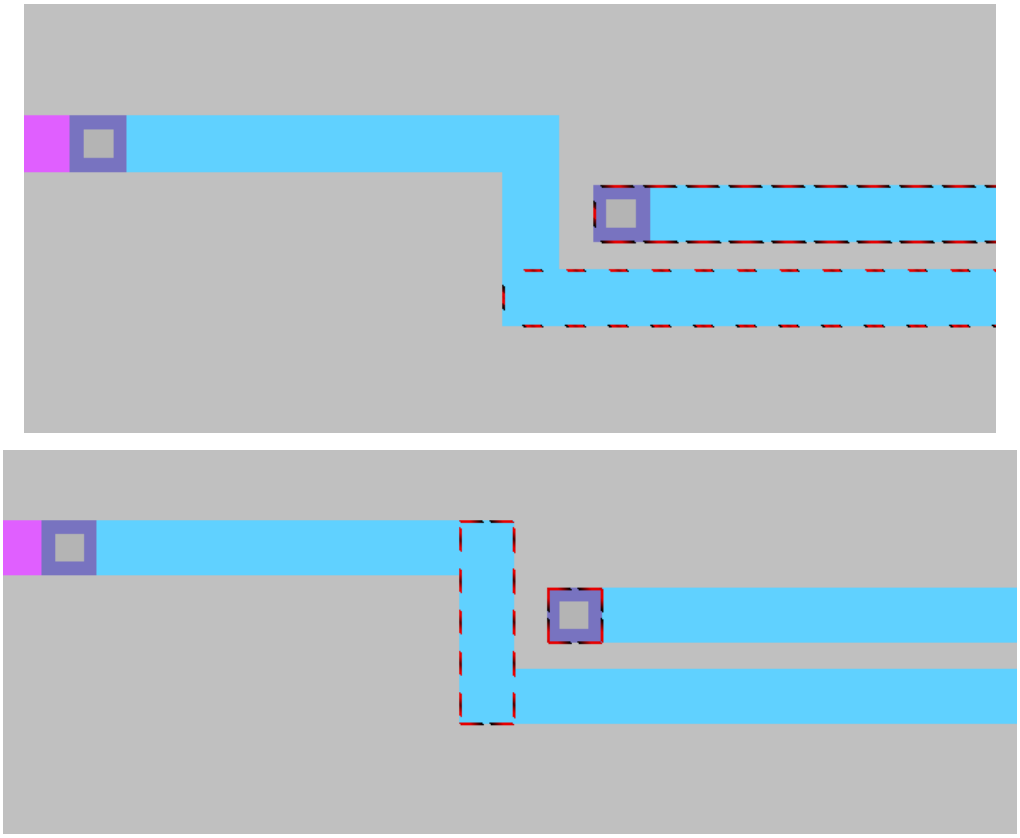

Figure 2.3: Special XY-axis error

2.3 Unsolvable Errors

Special errors include XY-axis errors in which the error block needs to be moved in both X & Y axis to solve the error.

Finding an Unsolvable error:

The error shown in Figure 2.4 looks like an X axis error, but it is unsolvable. If the M1 block is moved towards the right by the required amount, then the original error will vanish, but this will cause a new spacing error with the p-act. So, moving the block along X axis will not solve the error without creating a new one.

- If the M1 layer is moved along Y-axis (in both directions) here, then it will create new errors and also disturbs the structure.
- So, the only way to solve this error is by rerouting or changing the metal layer to M3 (or higher metals).
- So, when the error block is moved by some amount, again and again in both X and Y directions (one after the other), if the original error does not vanish without creating new errors, then it is classified as an “Unsolvable Error.”

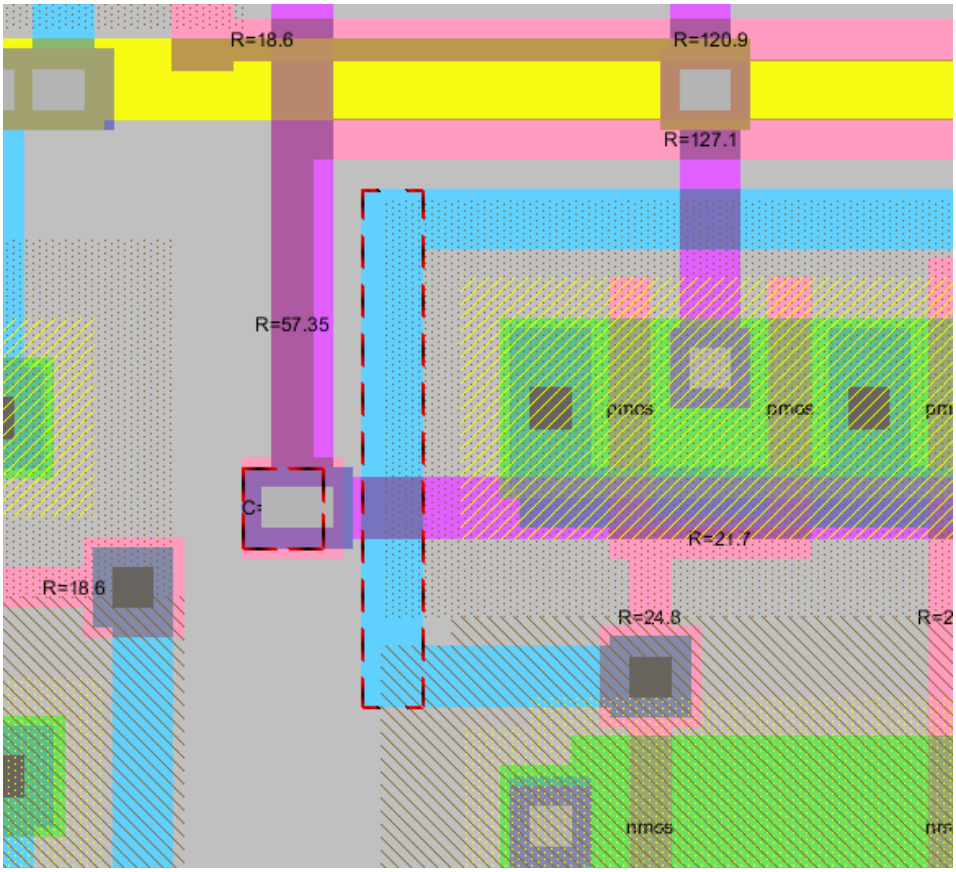

Figure 2.4: Unsolvable error

CHAPTER 3

Calibre LITHOview and DRC

For the earlier part of the project that is for classification of errors and for analyzing different layout, Electric Circuit Simulator was used. But it's extremely difficult and infeasible to do any sort of automation in it. Hence, Calibre LITHOview has been used for all further work, which not only provides much more functionality compared to Electric Circuit Simulator but also provides to use all these functionality using TCL scripting in batch mode.

Graphic Design Stream II (GDSII) was used as the default format for the layout files for this project. After importing the GDS file into CalibreLV the first part was to find out various details associated with the file.

3.1 CalibreLV Environment

Finding out the layouts and cells present in the GDS file

In order to do any sort of operation on the layout, first it is very important to know the layout names present in the file and also the cells that are present in each layout. Basic TCL commands and extensions have been used for this purpose

```
layout all  # returns list of all the layouts present in a gds file

$L cells   # returns list of all the cells that are present in layout 'L'
```

Finding out all the metal and via layers

The metal and via layers are denoted by numbers in most tools like CalibreLV, Cadence etc. For example in CalibreLV, the Metal 1 layer is assigned the number 31, Metal 2 is assigned 32 and so on, while via for metal 1 to metal 2 i.e. via 1 is assigned the number 51, via 2 is assigned 52 and so on. So, once the layout and cell names have been found, the metal and via

layers present in the layout can be found as well using the following command.

```
$L layers  # returns list of all the layer present in the layout
```

Finding the coordinates of metal and via layers

Now that the layer numbers have been found for all the metal and via layers present in the layout, in order to do any operation on these layers the coordinates of these layers have to be found as well. In order to do that the following command has been used.

```
$L iterator { poly | wire | text } cell layer range first last

# Returns a list of objects of the indicated type in the indicated cell and layer
```

The above command returns a list of objects of the indicated type in the indicated cell and layer. In order to get the coordinates of the metal and via layers, it is assumed that all those layers are polygon (poly). In that case that command will return the list of coordinates of all polygons present in the indicated layer.

Moving different layers

Moving the metals or via layers in GUI mode is fairly easy, it's just select and move. But moving the layers using commands is little bit difficult since there are no direct commands to move in CalibreLV. So in order to move let's say a metal 2 layer from (a1, b1 ,a2, b2) where a1 and a2 are X-axis extremities of the polygon layer while b1 and b2 are the Y-axis extremities, to (c1,d1,c2,d2), the layer first needs to be deleted from its original position which is (a1,b1,a2,b2) and then it needs to be created at (c1,d1,c2,d2). The following commands have been used for this purpose.

```
$L delete polygon cellName layer [-prop attr string [G|U]] x1 y1 x2 y2.....x_n y_n
# deletes the polygon described by its layer and coordinates from the specified cell

$L create polygon cellName layer [-prop attr string [G|U]] x1 y1 x2 y2.....x_n y_n
# creates the polygon described by its layer and coordinates from the specified cell
```

The delete polygon command deletes the polygon described by its coordinates, for the example given above, the layer will be 32 and the coordinates will be (a1, b1, a2, b2). Now since the layer needs to be moved to (c1, d1, c2, d2), it needs to recreated there using the create polygon command, in which the coordinates will be (c1, d1, c2, d2) and the layer will be 32 (since it's a metal 2 layer). The coordinates for the polygon layers can be found using the iterator poly command discussed above.

Running the commands in TCL

The above commands discussed can be executed in CalibreLV shell, but it's not possible to run multiple commands at once in the shell. So, all the above commands need to be used in TCL batch mode for CalibreLV. The following code snippet is an example showing the use of CalibreLV commands in batch mode using TCL scripting.

```
set my_layout [layout create layout_name.gds]
# setting a variable my_layout to load the original layout

set all_layouts [layout all]
# all_layouts will contain the list of all layouts present in the gds file

set layout_0 [lindex $all_layouts 0]
# layout_0 will contain the top layout from the list of layouts

set all_cells [$layout_0 cells]
# setting all_cells to contain list of all cells present in layout_0

set cell_0 [lindex $all_cells 0]
# cell_0 will select top cell from the list of all_cells

set lst_Mn [$layout_0 iterator poly $cell_0 $Mn range 0 end]
# lst_Mn stores all the polygons of layer Mn in cell_0 of layout_0

eval $layout_0 delete polygon $cell_0 $Mn -prop attr string 1 $old_cord
# deletes polygon of Mn layer from cell_0 of layout_0 having coordinates old_cord

eval $layout_0 create polygon $cell_0 $Mn -prop attr string 1 $new_cord
# creates polygon of Mn layer from cell_0 of layout_0 having coordinates new_cord
```

In the above code snippet, first the layout handle is being found, then all the layers in the layout is found. And all the cells present in the top layout are found. Then using the top cell, list of all the polygons coordinates of layer Mn are found using the iterator poly command. Then using one of the element of lst_Mn name old_cord, the Mn layer polygon is deleted and then it is being recreated at a new place with coordinate new_cord.

3.2 Calibre DRC

CalibreLV doesn't have an inbuilt design rule checker, so in order to run the design rule checker TSMC 28nm Calibre DRC was used. The DRC deck contains the name of the layout file i.e. the gds file. It also includes the error name for which it will check the design rule, one of the errors for example is M1.S.1 which is an M1 spacing error, similarly for M2, M3 and so on. Any kind of error can be included and the checker will check the violations for that, but for this program only spacing errors are being considered. The error names can vary from technology to technology and foundry to foundry.

Output of Calibre DRC

Calibre DRC outputs the results of the design check into two files, which in this case were named as layout_name.drc.results and layout_name.drc.summary. The results file contains the error polygon data, while the summary file mainly contains the data of the layers causing the DRC violations.

3.3 Error Polygons

Error polygon represents the error marker which basically shows where the error is being created. It can be of various shapes like trapezium, parallelogram, rectangle etc. When the design rule checker is run that is CalibreDRC in this case, it outputs the error polygon data for the selected error into drc.results file.

In order to solve the error, it has to be learned first, meaning the program should learn on itself whether it's an x-axis error, y-axis error or special error using the error polygon data.

Learning the error type

An error polygon can be of various shapes like a trapezium, rectangle or a parallelogram. A few of them are shown below.

Figure 3.1 Rectangular error polygon

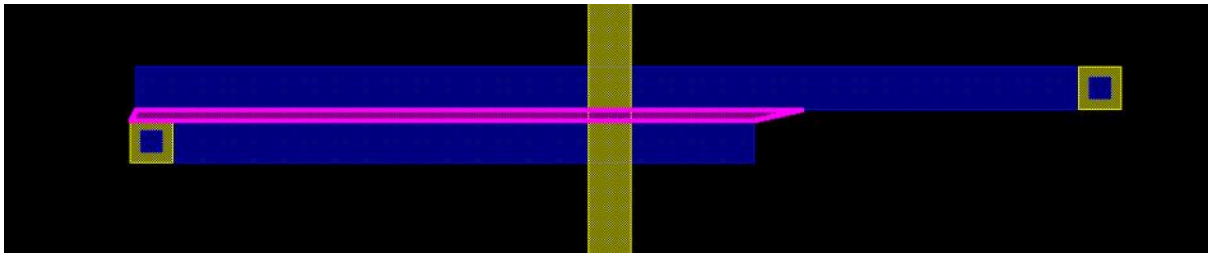

Figure 3.2 Trapezoidal error polygon

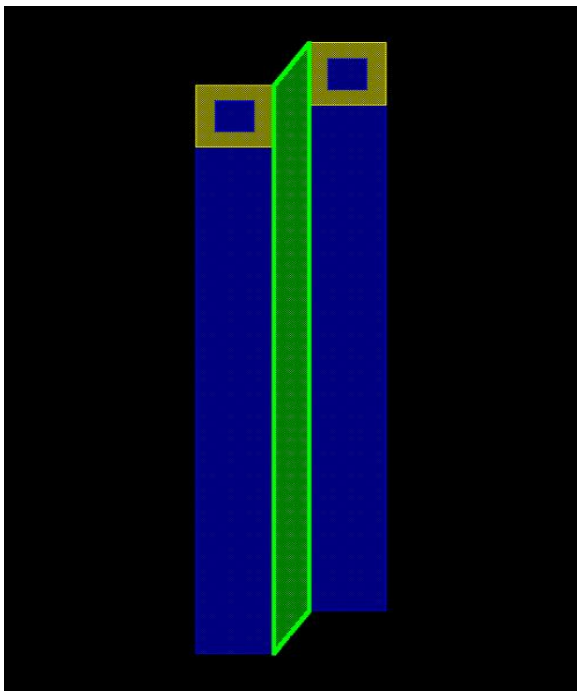

Figure 3.3 Parallelogram shaped error polygon

The type of error can be determined from the error polygon shape. For example, the error polygon in Figure 3.2 is trapezoidal where the two horizontal sides are parallel. Now since the assumption has been made that the metals can only run vertically or horizontally and not in any angle, it can be deduced that, two sides of an error polygon will always be parallel either vertically or horizontally. If the parallel sides are running horizontally i.e. along X-axis then it's a Y-axis error, like in Figure 3.2, but if it runs vertically i.e. along Y-axis then the error will be an X-axis error.

But an error polygon can be rectangular or square as well, like in Figure 3.1. There both the vertical as well horizontal sides are parallel. So in order to solve it, a few test cases needs to be generated. If any one of the two metal layers in Figure 3.1 is moved vertically i.e. either up or down, then the shape of error polygon won't change, two of the coordinates will change but it

will still remain a rectangle. But if any one of the two metal layers is moved horizontally i.e. either left or right, then the error polygon shape will change, the vertical sides won't be vertical anymore. They will become a bit slanted. They might still be parallel but they won't be vertical. So it can deduced that error type in Figure 3.1, is Y-axis type.

So in order to get the type of error from error polygon coordinates, the program will look for parallel vertical or horizontal sides, whichever side is parallel, it will give the error as expected. But if the error polygon is shaped like a rectangle or a square, then the program will select one of metal layer and move it along both the axes i.e. X and Y, by unit distance and check the error polygons. In one of those cases, the error polygon shape will change and the error type can easily be deduced from that.

CHAPTER 4

PROGRAM

4.1 Inputs

The program needs certain input files from the user, which are: a gds file containing the layout, a config file and a list of error files.

The config files contains various details about the layout like the list of metal and via layers present in the layout and their respective numbers. The program will read all these details from the config file and use them accordingly. In order to extract these information the program uses the ‘regexp’ command, using it to look for expressions that matches M1, M2, M3… M9 for the metal layers and via1, via2….. via8 for the via layers and then finds their corresponding layer numbers and stores them.

4.2 Extracting list of errors

Along with the config file, the user has to give another file containing the list of errors. This file should contain only the errors for which the program has to check and solve the DRC violations. The data in this file should be in a particular order, where the term “error_list” should be the heading of the file and one line should contain only one error. Now in order to extract those errors, the program uses the ‘regexp’ command to look for the expression “error_list” and extracts all the data that is below that expression as the list of errors. The name of errors can vary from technology to technology or foundry to foundry, for example one foundry might use the name ‘M1.S.1’ while other might just use ‘xyz1’. The program will work irrespective of whichever error name convention is used.

4.3 Function for running DRC

After getting the list of errors, the function will find the error polygons associated with each of those errors. The only input that function will get is error name and by default the DRC deck will not contain any error name in the error list. The function will put one error at a time into the DRC deck and it'll return the drc.results and drc.summary files from which the error polygons and the metal associated with the error can be found.

4.4 Extracting error polygons and layers causing errors

Also the program doesn't know which metal layer is causing which error. This information can be found from the drc.summary file after running DRC. The function uses the "regexp" command to look for the expressions matching 'M1i', 'M2i',…'M9i' and so on in the drc.summary file.

In a similar fashion the program will extract the error polygons data from drc.results file associated with each error. This way the error name, the error polygons and the layers causing the error can be mapped.

4.5 Solving the errors

After the error is learnt, the following steps are executed to solve the error.

- **Calculating the DRC distance:** The amount by which a metal layer needs to be moved so that it can be solved is called DRC distance. This can be found by taking one of error causing layers and extending it to infinity (practically this is just twice the width of length of layout boundary). Then finding the error polygon corresponding to this error and calculating the longest non-vertical or non-horizontal line. This longest side will give the DRC distance. The metal layer that was extended is reset back to its normal position.

- **Prioritizing the error layers:** One of the error layers has to be selected to move first. This selection is based on the number of vias connected to an error layer. The error

layer having lesser number of via connections has to be moved first. If the error is solved, then this would result in minimum disturbance in the layout, as minimum number of higher/lower level metals will be moved in this case.

- **Finding the new coordinates and moving the 1st layer:** After getting the DRC distance and selecting the first layer, the new coordinates using the coordinates of the layer, the DRC distance and the existing distance between the layers. Similarly the new coordinates can be found for the vias associated with the layer. After getting the new coordinates the layer is moved along with the vias, while making sure that if any other connection of the via breaks it shorted using higher metals to prevent any LVS violations.

- **Running DRC and moving other layer if necessary:** The DRC is run after moving the 1st layer and if it solves the error then the program moves on to the next error. But if the error still remains or it creates any new error then the all the actions are undone i.e. the 1st layer along with all the vias are moved back to their original position. Then step 3 is done for the 2nd layer, i.e. calculating the new coordinates for the layer and vias and moving them accordingly. If it solves the error then we move on, but if it doesn't solve or create some new errors then we mark it as unsolvable error.

CHAPTER 5

RESULTS AND CONCLUSIONS

5.1 Results

The following layout was given as a test case to the program. It was taken from a real 28nm TSMC design.

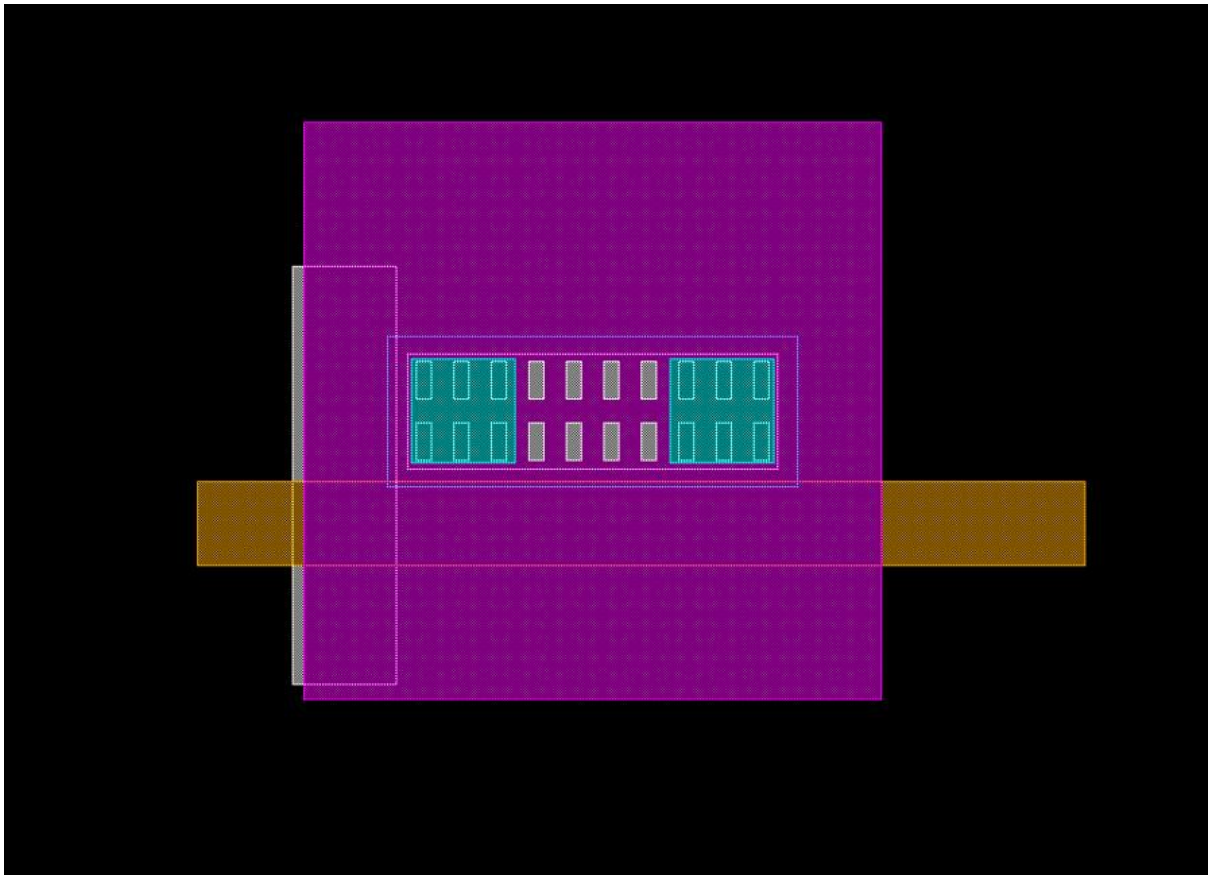

Figure 5.1 Layout used as a test case

The layout has all the metals from M1 till M9 and vias from via1 till via8 in the form of a stacked via connection. There are two DRC violations, one caused by M4 (grey layer) running vertically and the other one by M5 (brown layer) running horizontally. The error polygons are shown in Figure 5.2.

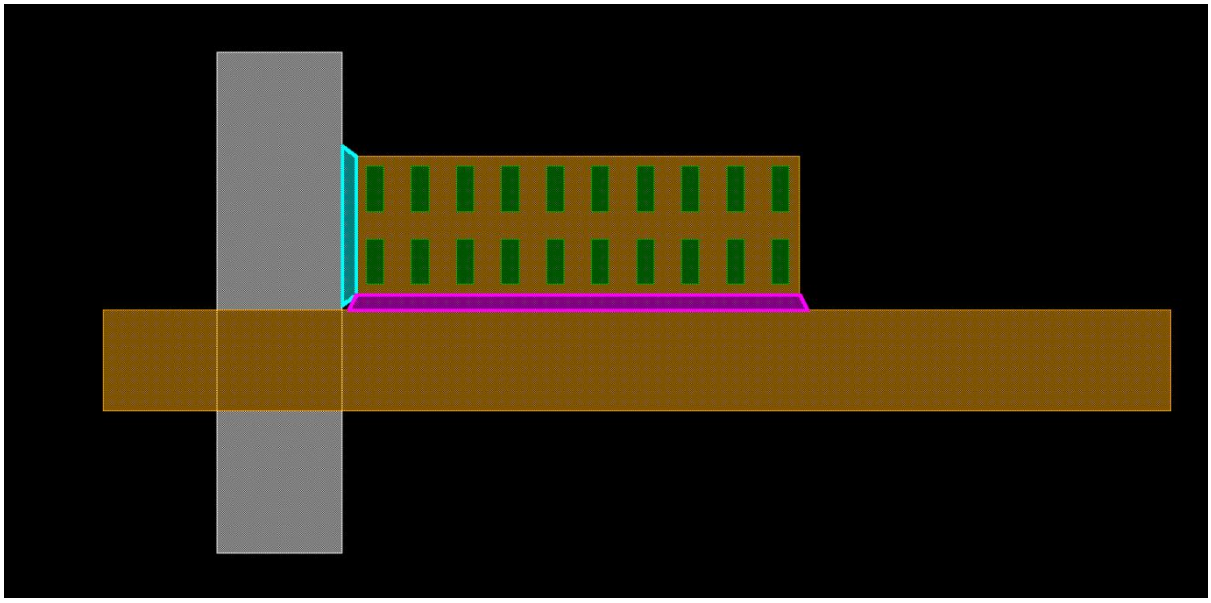

Figure 5.2 Test case layout showing error polygons

The M4 spacing error which is an X-axis error is being shown by the sky-blue marker. The M5 spacing error which is a Y-axis error is being shown by the magenta marker.

When layer with less number of vias is moved

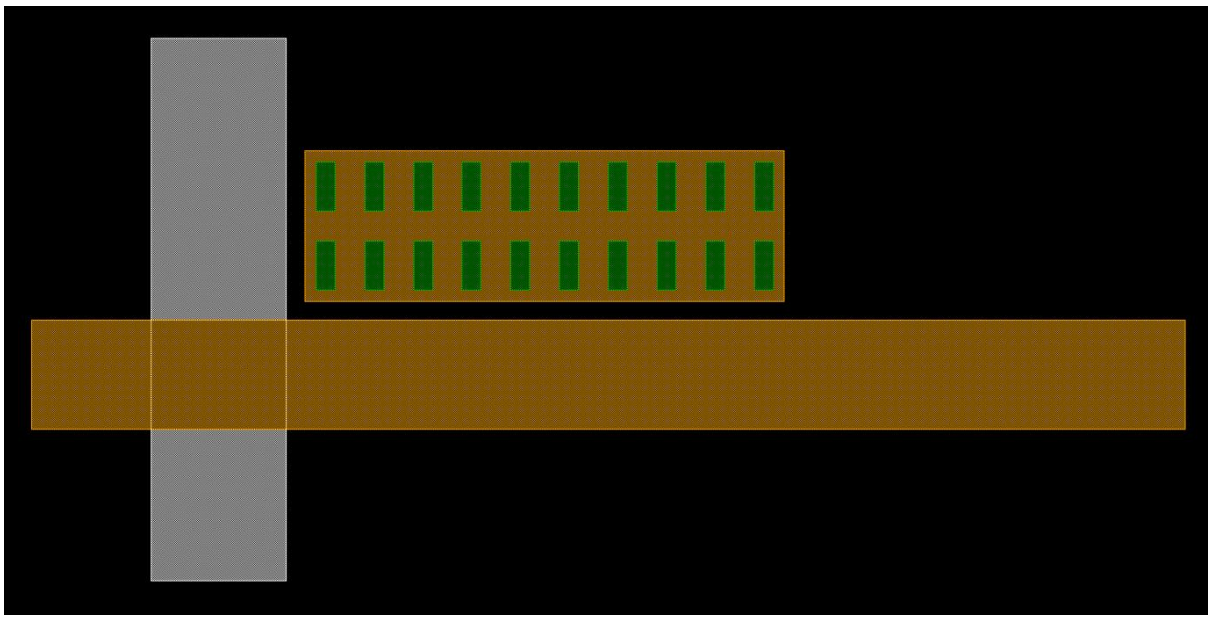

Figure 5.3 Test case layout after running the program

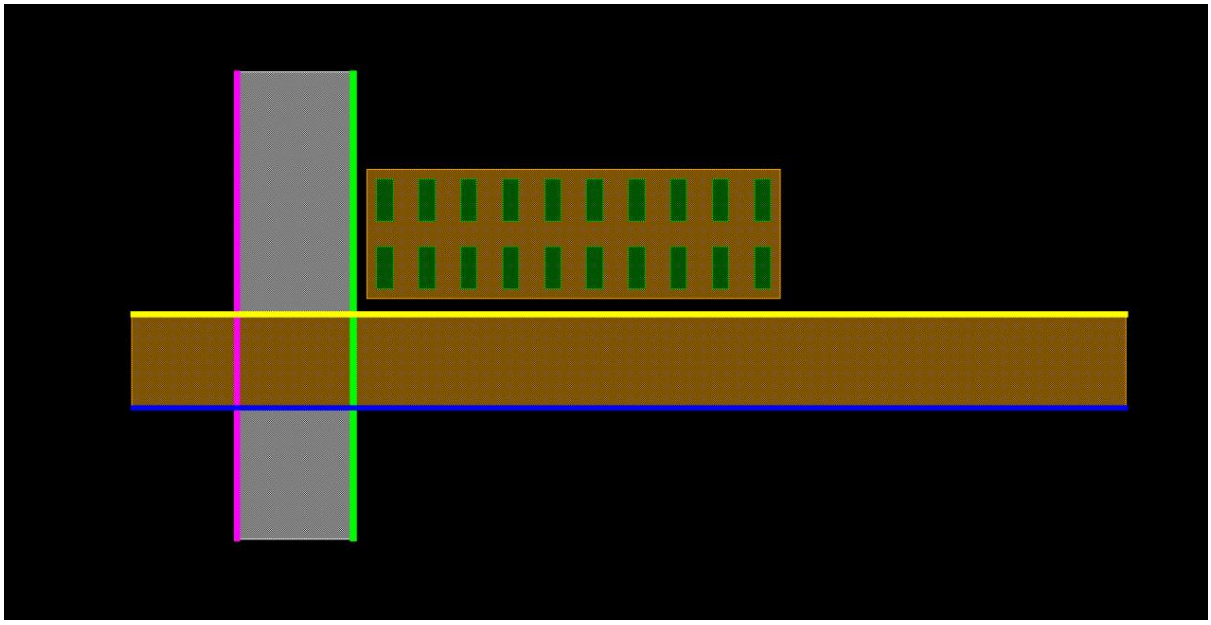

Figure 5.4 Difference between the solved and unsolved test case layouts

In order to solve the M4 spacing error the M4 layer on the left which is running vertically is moved along -ve X-axis to solve the error. In Figure 5.4, the difference in the positions of M4 and M5 layers has been depicted, before and after running the program. The light green marker shows its initial position and the magenta marker shows its final position.

The M5 spacing error is solved by moving the bottom M5 layer which is running horizontally along –ve Y-axis. The yellow marker shows its initial position and the dark blue marker shows its final position.

Here these two layers are moved other than the stacked via because these layers have less number of vias (i.e. 0) compared to layers in the stacked via connection.

When stacked via layer is moved

The algorithm always decides to move the layer with lesser number of vias first. Figure 5.5 shows that the program also works when stacked via layer is moved.

In Figure 5.2 the dark green colored layers are via5 layers. The lower and higher via layers below and above it are hidden. In Figure 5.5, light blue colored layers are via6 layers, dark green colored layers are via5 layers and magenta colored layers are via4 layers.

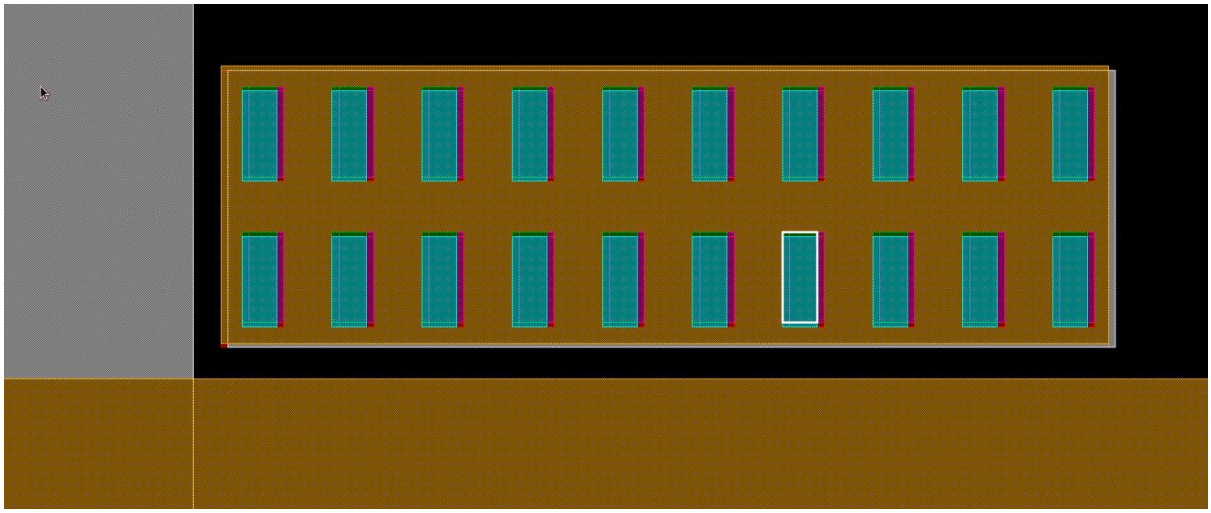

Figure 5.5 Test case layout after stacked via layers are moved

In order to solve the errors the M4 and M5 layers in the stacked via are moved along +ve X-axis and +ve Y-axis respectively. In Figure 5.6 the magenta marker shows that M4 layer is moved to the right and the dark blue marker shows that M5 layer is moved upwards.

Along with the layers it can be seen from the Figure 5.5 and 5.6 that the via4 layers (magenta) are moved to the right and the via5 layers (dark green) are moved upwards. The via6 layers (light blue) are at their original position because the M6 layer has not moved.

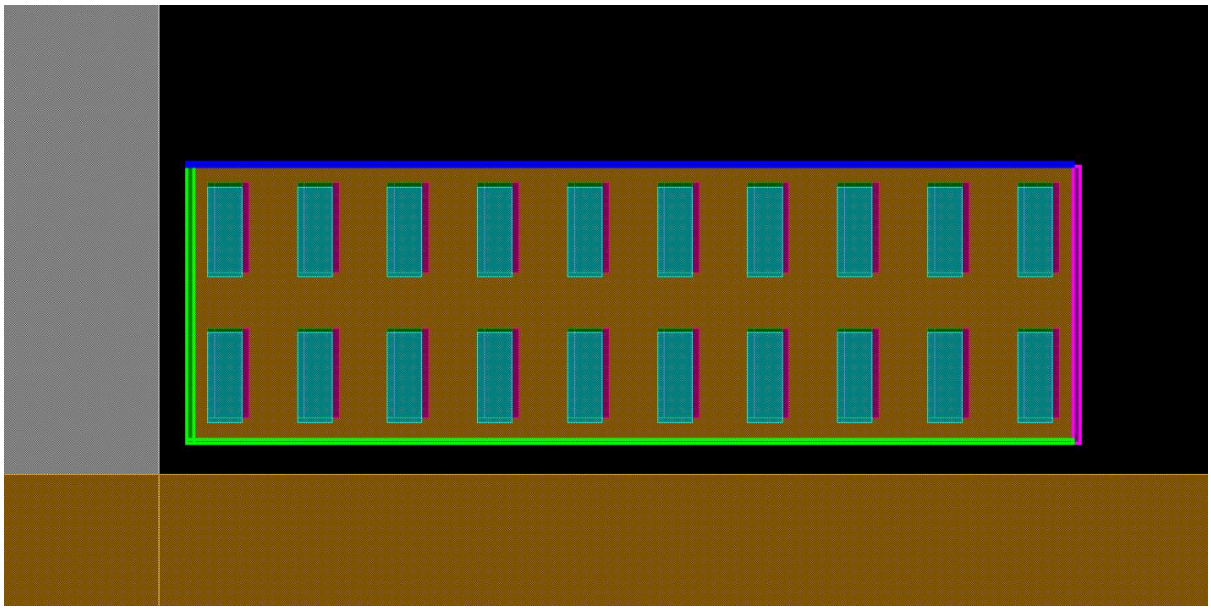

Figure 5.6 Difference between solved and unsolved test case layouts for stacked via

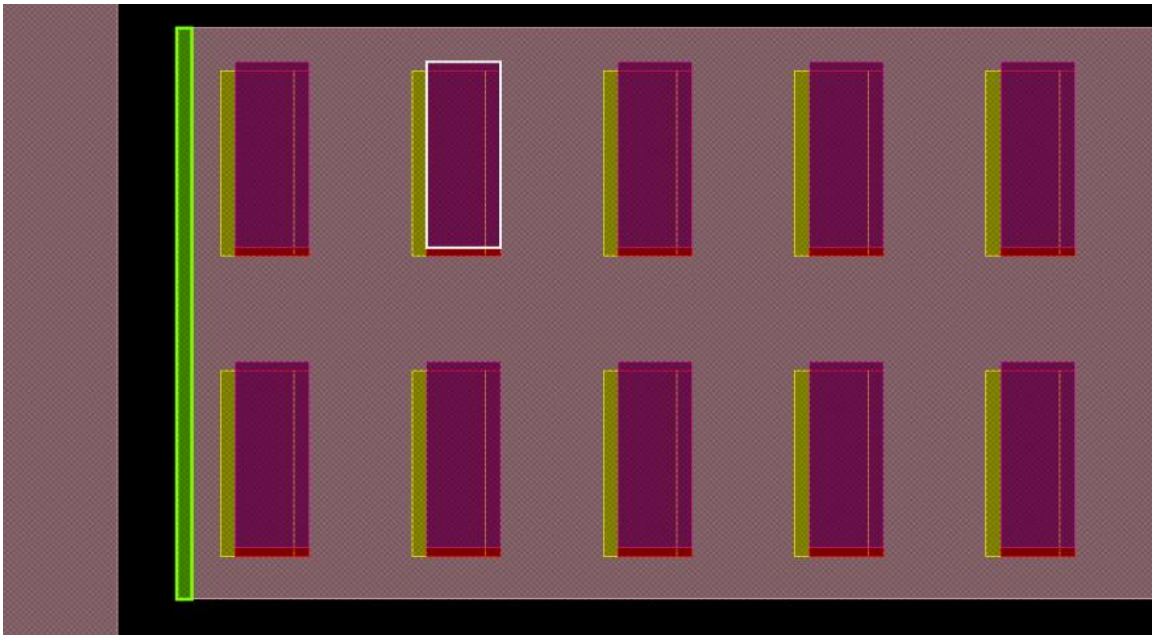

Figure 5.7 Showing M4 layer and the corresponding vias(via4, via3) movement

First, the M4 error is considered by the algorithm and the M4 layer in the stacked via connection is moved to the right. Along with it, the via layers via(n)s and L_vias, i.e. via4 (magenta) and via3 (red) layers are also moved to the right as shown in the Figure 5.7.

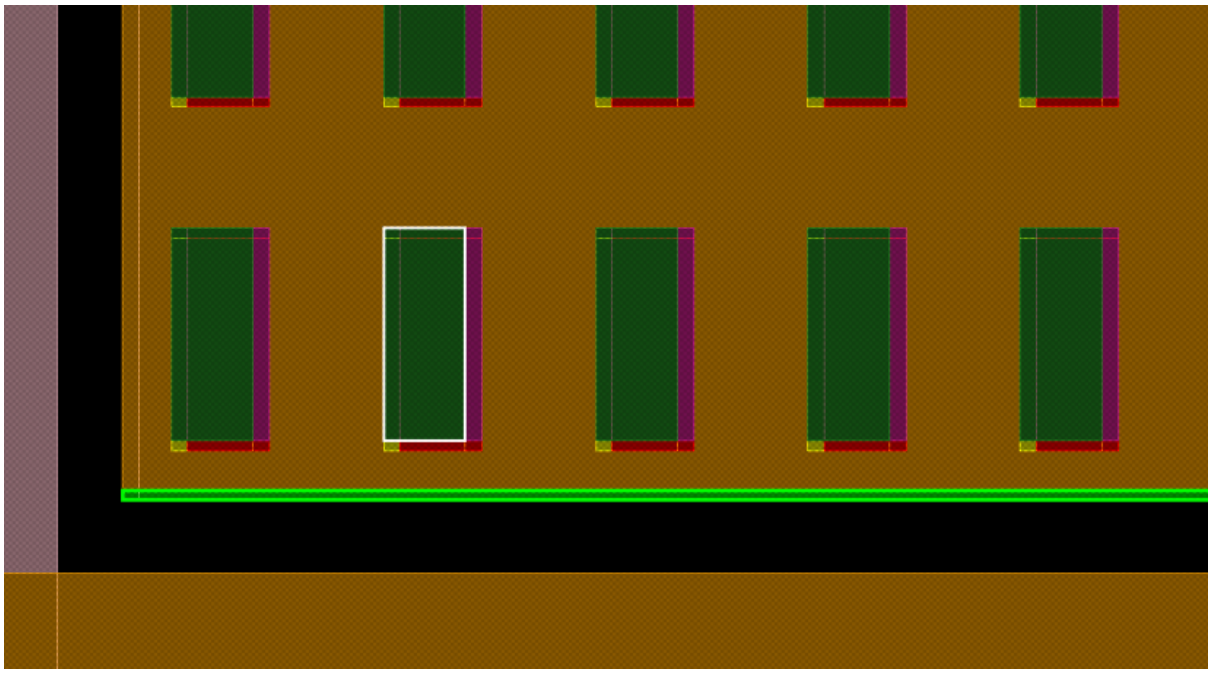

Figure 5.8 Showing M5 layer and the corresponding vias(via5, via4) movement

Next, the M5 error is considered and the M5 layer is moved upwards. Along with it, via(n)s i.e. via5 layers (dark green) and L_vias i.e. via4 layers (magenta) are also moved upwards as shown in the Figure – 5.8. Hence, the via4 layers are actually moved to the right and also to the top while solving both the errors. The via3 layers are moved to the right and via5 layers are moved to the top. All the remaining via layers are in their original position.

5.2 Limitations of the algorithm

- **Unsolvable Errors:** After moving both the layers, if the number of DRC errors are not decreased then the algorithm considers the error to be unsolvable. Rerouting should be done in order to solve these type of errors.
- The algorithm moves layers by deleting and creating them again. Due to this other type of DRC errors such as Enclosure errors (via is not properly enclosed by the layer), minimum and maximum size errors, resolution errors might get created.
- This algorithm only works for higher level metals i.e. metals used for global routing.
- The error is only caused by higher metals i.e. above Metal 3.
- This algorithm assumes that metal layers run in a single direction. If this is not followed, new DRC errors might get created. New LVS violations might also get created.

5.3 Conclusions

An algorithm for automating the DRC cleaning process has been proposed which reduces the time as well as the effort needed to solve the DRC violations manually. First different layouts were inspected to find different types of DRC errors. Then an efficient algorithm was devised to learn and solve these DRC errors. The algorithm solves as many of these DRC errors as possible without creating any new errors. This algorithm works with designs created in any environment and with all kinds of design rules irrespective of the technology scale. This algorithm is efficient enough to maintain LVS cleanliness even after solving all the errors.

REFERENCES

[1] Mentor Graphics, *Batch Commands User's and Reference Manual for CalibreLITHOview*, version 2007.1.

[2] *Electric User's Manual*, version 9.07.