AUTOMATED DRC ERROR SOLVER FOR LVS CLEAN LAYOUTS

A Project Report

Submitted by

MANDA SASHANK

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

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 **Manda Sashank (EE19M076)** to the Indian Institute of Technology Madras, in partial fulfillment for the award of the degree of **Master of Technology,** is a bonafide 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 **Vishwajeet Anand**, who was my fellow associate in doing this project. This project would not have been possible without his contributions and insightful observations.

ABSTRACT

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

Current generation ICs (Integrated Circuits) have millions of transistors and thousands of logic gates. The layouts that are designed for these are large, dense and really complex. Cleaning these layouts of DRC (Design Rule Check) and LVS (Layout v/s Schematic) violations is an important step and this can be a cumbersome process. Especially, making a layout free of DRC violations is complex, repetitive and highly time consuming. Mostly, the metal interconnects which are used to connect several blocks and components inside a layout create these DRC violations.

An efficient algorithm that automates this DRC cleaning process has been developed in this project. This algorithm solves as many of these DRC violations as possible without creating any new violations, thus saving a lot of time and effort that goes into solving these violations manually.

TABLE OF CONTENTS

LIST OF FIGURES

ABBREVIATIONS

IITM	Indian Institute of Technology Madras
VLSI	Very Large Scale Integration
DRC	Design Rule Check
LVS	Layout v/s Schematic
GDSII	Graphic Design System II
IC	Integrated Circuit
SSI	Small Scale Integration
EDA	Electronic Design Automation
Calibre_LV	CalibreLITHOview
TCL	Tool Command Language

CHAPTER 1

INTRODUCTION

1.1 Motivation

An Integrated Circuit (IC) is an assembly of various electronic components (both active and passive), fabricated as a single unit on one small piece of semiconductor material. Integration scale is the number of components fitted into a standard size IC. Back then, SSI phase consisted of less than 100 transistors. Currently, the VLSI phase consists of more than 3,00,000 transistors and about 10,000 gates. The proportion of functioning devices manufactured on a single wafer determines the yield of a semiconductor process. In the current VLSI phase, the die area is significantly reduced. A large number of transistors are manufactured on this smaller area thus increasing the yield. But, reducing the die area makes the designs more and more compact. To ensure reliability of such compact designs, the design rules should be followed.

A design rule is a geometric constraint imposed on an Integrated Circuit (IC) to ensure that the designs function properly, reliably, and can be produced with acceptable yield. 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 information about the error location and the type of error. These DRC errors have to be located by the designer and should be cleaned manually. With the modern technological advancements, the density of the designs has increased rapidly while the die size and transistor gate length has decreased, thus making it further difficult to clean these DRC errors manually.

Consider the layout shown in the below Figure – 1.1. For this LVS clean design, it takes about 15 hours to manually fix all the DRC errors.

- There are around 21,000 DRC errors in this layout.
- The dimensions of the layout are 1330 µm x 7.8 µm (approximately).
- There are about 67,000 transistors in this layout.

Hence, for a real design like this, it is evident that solving all the DRC violations manually is a repetitive and time consuming process. The picture shown below is a small part of a very large layout.

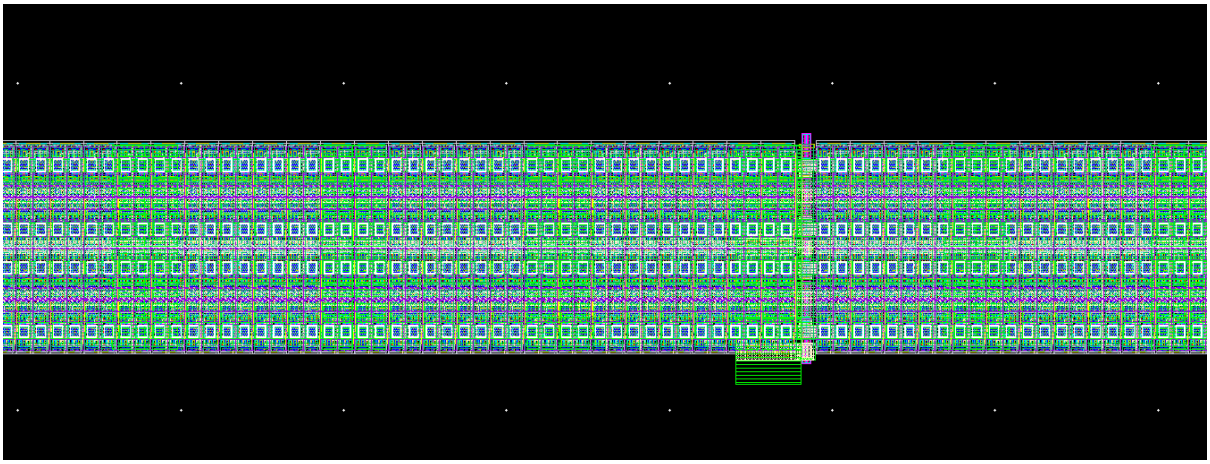

Figure 1.1 – Snippet of a layout with DRC errors

1.2 Objective

This project aims to automate the DRC cleaning process by using an algorithm which solves as many of these DRC errors as possible without causing any new error, thus saving a lot of time and effort that is required for solving these errors manually. The algorithm is expected to meet the following requirements.

- It is expected to work with all kinds of design rules.
- It is expected to work with designs created in any environment irrespective of the technology scale.
- The algorithm is also expected to retain an LVS clean layout after solving the DRC errors.

1.3 Assumptions

- The layout should be LVS clean.
- The DRC errors occur only during the routing step. It means, only the spacing errors corresponding to different metal layers are considered.

- Metal layers having DRC errors should run only in a single direction, either horizontally or vertically.
- The direction in which the metal layers run should be alternative. For example, if M5 errors have to be solved and M5 runs horizontally, then the algorithm assumes that the metals M4, M6 run vertically.

From the above assumptions, it is clear that this algorithm is only suitable for higher level metals (i.e. M3 or higher). Because the lower level metals (i.e. M1 and M2) run in both directions during local routing, the DRC errors caused by these lower level metals are ignored.

1.3 Prerequisites for the algorithm

- **Design Environment:**

Earlier part of the project, such as classifying different types of DRC errors by analyzing different layouts has been done using “Electric Circuit Simulator”. Any sort of automation using this simulator was really difficult.

Hence, Calibre Layout View (Calibre_LV) has been used for the remaining work, which provides a lot more functionality and automation using scripting languages. For this project, Graphic Design Stream II (GDSII) has been used as the default format for the layout file. Different Claibre_LV commands that are used in this algorithm have been elaborated in Appendix A.

- **Scripting Language:**

The scripting language that is used for this project is TCL (Tool Command Language). TCL is a high level, general purpose, dynamic programming language. It is simple yet powerful. It is vastly used in different command shells to control different types of EDA tools.

Different Calibre_LV commands listed in Appendix-A can be executed in the Calibre_LV shell. But, it is not possible to run multiple commands at once in the shell. So, multiple commands are used in batch mode using TCL scripting.

- **DRC function:**

Calibre_LV does not have an inbuilt DRC deck. In order to run DRC, a design rule checker file (_caliber.drc_) has been used for this project. The GDS file name has to be given to this DRC file.

a. When an error name is given as an argument for the DRC function, the DRC file executes DRC and generates two files, namely the summary file and the results file.
b. The summary file contains information about the metal layer(s) which has (have) DRC errors.
c. The results file contains information about the error polygons (elaborated in Chapter-2).

- A “config-file” is given to the program which contains the information about metal layer names and numbers, and also the list of DRC errors. Inside the program, the given GDS file is opened and the top level layout and cell are marked.
- Each error from the config-file is given as argument to the DRC function one after the other. The summary file is then checked to know about the metal layer corresponding to the DRC error and the results file is used to generate the list of error polygons.

1.4 Algorithm

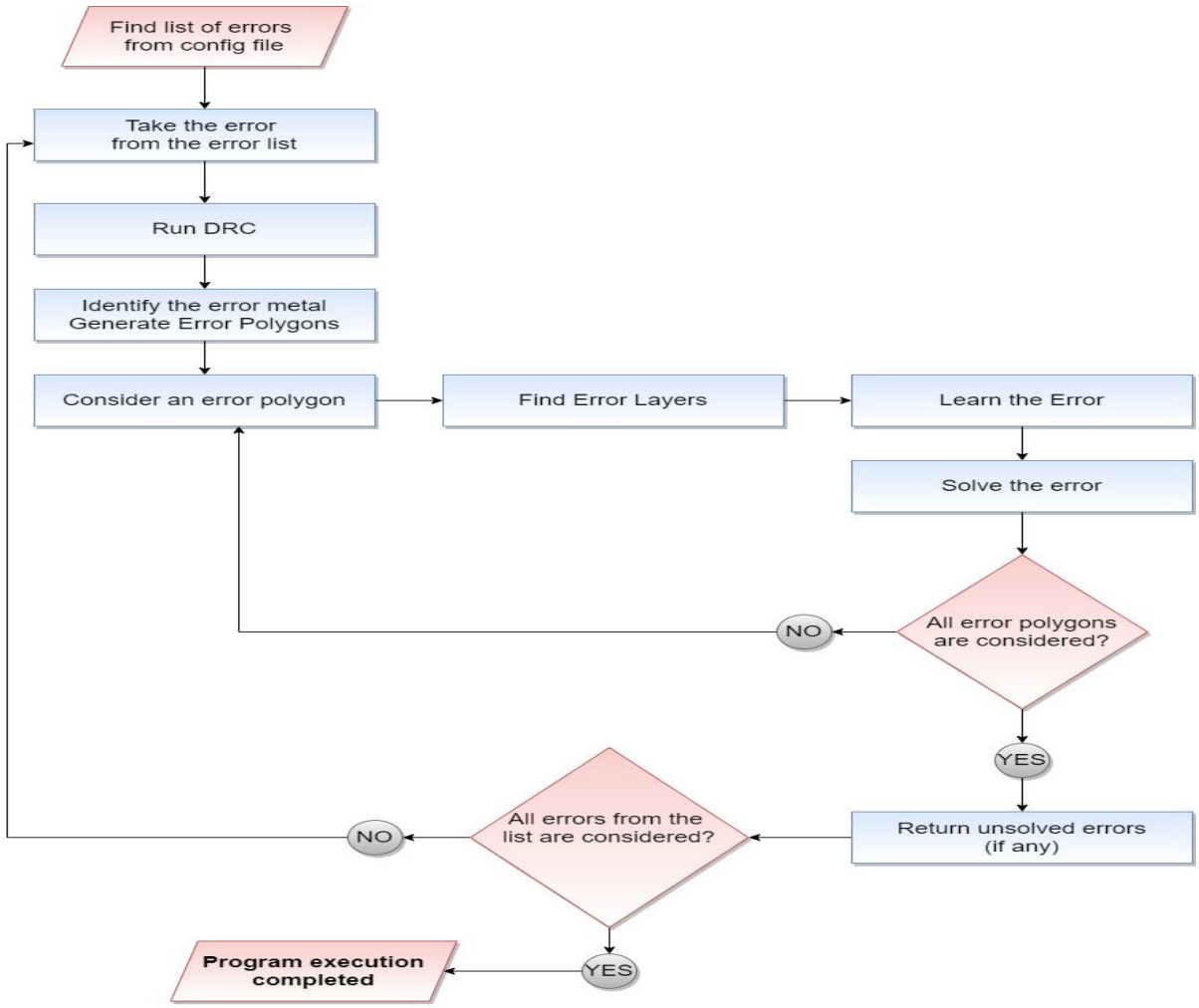

Figure 1.2 - Flowchart showing an outline of the algorithm

CHAPTER 2

FINDING ERROR LAYERS

A spacing error is caused by two metal layers. This error can be identified with the help of an “Error Polygon”. Error Polygon is nothing but a marker that highlights the area where the error is being created. When the DRC function is called, it executes the DRC command for a particular metal layer and generates all the error polygons corresponding to that particular metal layer.

2.1 Types of Errors and Error Polygons

Errors are classified into two types, X-axis errors and Y-axis errors. An X-axis error is caused by two layers running vertically and can be solved by moving the metals layers along X-axis. A Y-axis error is caused by horizontally running layers and can be solved by moving layers along Y-axis.

The error polygons that are generated by the DRC deck are in general trapezoid shaped (i.e. they have four coordinates and have at least one set of opposite sides parallel). In some special cases, the error polygons are also parallelogram and rectangle shaped.

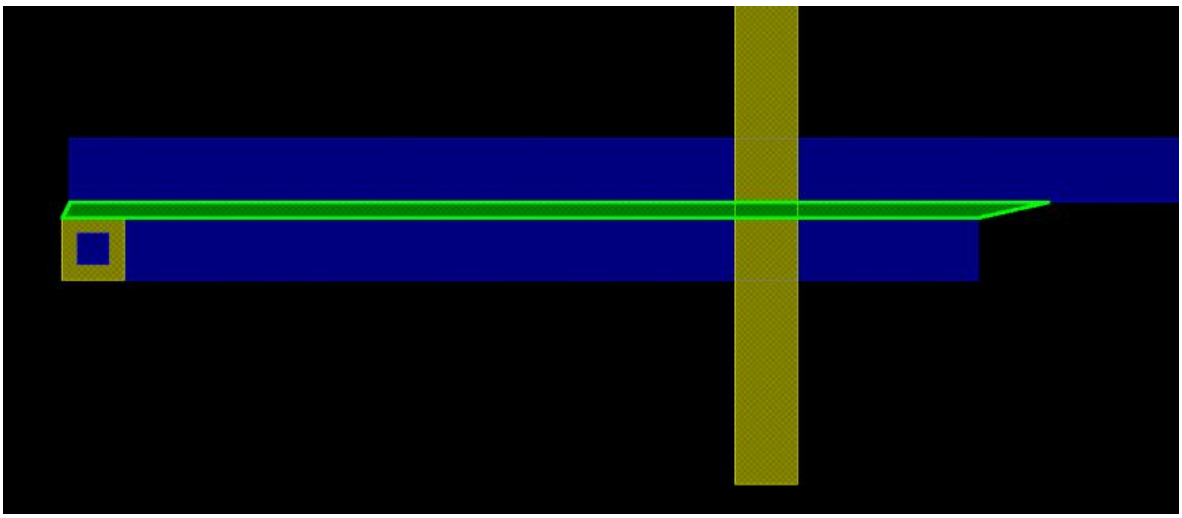

Figure 2.1 - Trapezoidal shaped Error Polygon

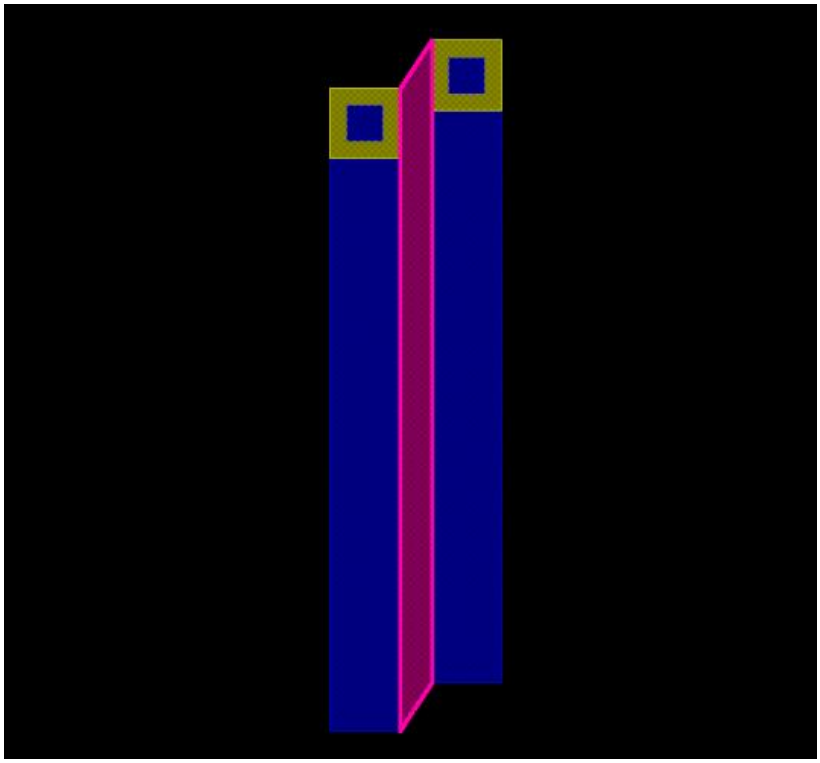

Figure 2.2 - Parallelogram shaped error polygon

Figure 2.3 - Rectangle shaped error polygon

The errors shown in figures 2.1, 2.3 are Y-axis errors and the error shown in figure 2.2 is an X-axis error.

2.2 Function for finding layer extremities

- The “layout iterator” command from Calibre_LV (Appendix A) outputs all the layers for a particular metal in the form of a TCL list. Each element of the list contains four points corresponding to that particular metal layer (x1 y1 x2 y2 x3 y3 x4 y4). Of these four points, the extreme two points i.e. (P0-the lower coordinate, Pn-the upper coordinate i.e. the diagonally opposite one) are needed for the program.
- The function first separates the x and y coordinates into two lists {x1 x2 x3 x4}, {y1 y2 y3 y4}. These two lists are sorted and the extremities of each list are taken and are formed into a separate list {x0 y0 xn yn}. Here P0 = (x0 y0) and Pn = (xn yn).
- Hence, this function takes the list {x1 y1 x2 y2 x3 y3 x4 y4} as an argument and returns the list {x0 y0 xn yn} that contains the layer extremities.

2.3 Function for finding error layers

Each error polygon has four coordinates (xpoly_i, ypoly_i where i = 1, 2 3, 4). The metal layer list contains all the rectangles corresponding to that particular metal. A metal layer is considered to be an error layer if “exactly two of the four polygon points lie inside or on the boundary of that layer”.

Mathematically, x0 <= xpoly_i <= xn and y0 <= ypoly_i <= yn (this should be true for exactly two values of i).

So, the function for finding the error layers takes metal layer list and the error polygon coordinates as arguments and returns the metal layers which are causing that DRC error.

The DRC deck for this particular algorithm generates error polygons whose coordinates lie on the boundary of the metal layer (i.e. ‘=’ condition applies here). This error finding function can be modified according to the DRC deck used.

CHAPTER 3
LEARNING THE ERROR

After the error layers are found for a specific error polygon, the error layer(s) should be moved in a particular direction and by a particular distance to solve the error. In order to know these, it is important to learn the error. Learning the error comprises the following steps:

- Identifying the type of error and finding the distance between the error layers.
- Finding the direction in which the error layers should be moved.
- Finding out the distance by which the error layer(s) should be moved in order to solve the DRC error.

To carry out the above mentioned steps, different functions are defined and are called accordingly.

3.1 Function for finding 'vias' connected to a layer

A metal layer is connected to its higher or lower level metal using a 'Via' layer. By using the 'layout iterator' command, the list of all the vias corresponding to a particular metal layer is found. For example, for an M4 layer – all the via4 (i.e. a via which connects M4 and M5) layers should be found. From the list of these via layers, the vias which are connected the particular error layer should be found.

If V1 (p1 q1), V2 (p2 q2) are the extremities of a via layer, and M0 (x0 y0), Mn (xn yn) are the extremities of an error layer, then the required conditions are:

x0 <= p1 <= xn and y0 <= q1 <= yn, x0 <= p2 <= xn and y0 <= q2 <= yn, i.e. both the points V1, V2 should lie inside or on the boundary of the error layer.

Hence, this function takes the error layer extremities and list of vias as arguments and returns a list which contains vias that are connected to the error layer. Here, the '=' condition implies that the via is on the boundary of the metal layer (i.e. touching the metal layer from the inside). Though this '=' condition is considered here, this type of condition

is almost rare because, a minimum spacing condition is required to be met when a metal envelopes a via.

3.2 Function for finding a contact layer connected to a via

A metal to metal contact is nothing but a connection between two consecutive metal layers (i.e. M(n), M(n+1) or M(n-1), M(n)). A contact can be placed in the following ways:

a. Two metals are running, one on top of another and simply a via is placed between them forming a contact between the two metals. For example, a via5 is placed between M5 and M6 layers forming an M5-M6 contact.
b. Along with the via, small layers of both the metals are also placed. E.g. For an M5-M6 contact, along with a via5, small layers of M5 and M6 enveloping the via5 are also placed in between the running M5 and M6 layers.

For the case-b type contact placement, when an error layer is moved, the metal layers that are created while placing the contact should also be moved so that additional DRC errors are not created.

For a particular error via (i.e. one of the vias connected to the error layer), the metal layer connected to that via should be found. The conditions required are:

x0 <= p1 <= xn and y0 <= q1 <= yn, x0 <= p2 <= xn and y0 <= q2 <= yn, where V1 (p1 q1), V2 (p2 q2) are the extremities of the error via, and M0 (x0 y0), Mn (xn yn) are the extremities of a metal layer.

Error via, error layer and the metal layer list are given as arguments for this function and the metal layer that is connected to the error via is returned (i.e. the contact layer). For a given via, error layer and a contact layer are connected to it. Error layer extremities are given as an argument here in order to differentiate the contact layer from it. For case-a type condition, an empty list is returned by the function in order to inform the program, that no contact layers are present.

3.3 Function for test case generation

- **Arguments:** Error layers, metal layers list, via layers list, error polygon index
- **Returning value(s):** Test case error polygons appended into a single TCL list

Since the layers of a particular metal only run either horizontally or vertically, the type of errors that occur are X axis errors and Y axis errors. For a given error polygon, there are two error metal layers. Test case error polygons are generated by moving each error layer in +ve X, -ve X, +ve Y, -ve Y directions. So, a total of 8 test case error polygons are generated. These error polygons are used to determine the type of error and also the distance between the error layers.

To move any layer, “layout delete” and “layout create” commands of Calibre_LV (Appendix A) are used. Original layer coordinates (where the layer is currently) and new layer coordinates (where the layer should be moved) are given to both these commands respectively in order to move the layer.

Test case generating function executes the following steps:

a. An error layer is moved in a particular direction (for example along +ve X axis) by a distance of 1 data-base unit (For Calibre_LV, 1 db unit = 0.001 μm, the smallest possible measurement).
b. Function-1 is called and the returned error vias list is saved into a variable.
c. For every error via, the contact layer is found by calling Function-2.
d. The contact layer is also moved in the same direction by 1 db unit (this step is skipped if Function-2 returns an empty list).
e. The DRC function is called and the DRC is executed, generating the error polygon list.
f. The test case error polygon for this DRC error is identified using the index which is given as an argument to this function. This index indicates the location of the error polygon corresponding to this particular DRC error.
g. Steps a-f are repeated for the remaining directions for error layer 1 and also for all the directions for error layer 2, using a loop and some defined variables.

Thus, the test case generating function, when called returns a list which contains all the test case error polygons. Here, after generating each test case, the entire process is reversed in order to get back the old layout. So, the layout is undisturbed after the test case generation process.

3.4 Function for learning the error

- **Arguments:** List of error polygons, error polygon index, test case error polygons
- **Returning value(s):** Flags determining the type of error, distance between the error layers.

There are two types of errors, an X-axis error and a Y-axis error as depicted in the figures in Chapter 2.1.

a. An X-axis error is a spacing error between two metals running along Y-axis. Hence, to solve the error, one of the error layers should be moved along X-axis.
b. A Y-axis error is an error between two metal layers running along X-axis. The metals should be moved along Y-axis in order to solve the error.

The error polygons are generally trapezoid shaped and only the error polygon coordinates are sufficient to determine the type of error for this general case. A special case occurs, where the error polygon is rectangle shaped, when the lengths of the metal layers are equal. For this special case, the test case error polygons are needed.

Determining the type of error:

a. General case:

An error polygon contains 4 points {x1 y1 x2 y2 x3 y3 x4 y4}. The x and y-coordinates are separated into different lists, X {x1 x2 x3 x4} and Y {y1 y2 y3 y4}. Now, these two lists are sorted in such a way that the duplicate elements are eliminated. The length (i.e. the number of unique coordinates) of each list is calculated (x_len and y_len).

For metals running along X-axis, the error polygon has the horizontal sides parallel. So, the number of unique y-coordinates will only be 2 where as the number of unique x-coordinates will be greater than 2.

Similarly for metals running along Y-axis the number of unique x-coordinates will be lesser than the number of unique y-coordinates.

- **Case-1:** x_len > y_len → Y-axis error
- **Case-2:** x_len < y_len → X-axis error
- **Case-3:** x_len = y_len → Special error, the error polygon is a rectangle. Hence test case error polygons are needed to determine the type of error.

b. Special case:

Here, x_len = y_len. So, the test case error polygons for +ve X and +ve Y (corresponding to one of the error layers) are taken, and are sorted in the same manner to get the number of unique X and Y coordinates (x_lenX, y_lenX for +ve X test case and x_lenY, y_lenY for +ve Y test case).

- **Case-1:** x_lenX > y_lenX → Y-axis error
  When one of the layers is moved, the error polygon is no longer a rectangle and hence the number of unique x-coordinates became four.
- **Case-2:** x_lenX = y_lenX → insufficient information (+ve Y test case should be considered)
  x_lenY < y_lenY → X-axis error

Here, it is certain that one of the two cases occurs and hence the type of error can be determined.

Determining the direction of movement and the distance between error layers:

a. Case-1: Y-axis error

- **Notation:**
  L1 → layer 1, L2 → layer 2
  ep → error polygon, +Y → test case for +ve Y-axis, -Y → test case for -ve Y-axis
  y_diff → difference between the y-coordinates

The DRC deck generates error polygons on the boundary of the error layers. So, the difference between the y-coordinates of the error polygon is nothing but the distance between the error layers.

- y_diff_ep → distance between the error layers

y_diff_+Y and y_diff_-Y are also calculated for both the layers. For example, if y_diff_+Y for layer 1 is greater than y_diff_ep, it means that the distance between the layers has increased. So, layer 1 should be move along +ve Y axis in order to solve the DRC error.

- y_diff_L1+Y > y_diff_ep → move layer 1 along +ve Y axis
- y_diff_L1-Y > y_diff_ep → move layer 1 along -ve Y axis
- The same conditions also apply for layer 2. To solve the error, if layer 1 has to be moved along +ve Y axis, it implies that layer 2 has to be moved along -ve Y axis. There is no need to check the above conditions for layer 2 again.

b. Case-2: X-axis error

- **Notation:**
  L1 → layer 1, L2 → layer 2
  ep → error polygon, +X → test case for +ve X-axis, -X → test case for -ve X-axis
  x_diff → difference between the x-coordinates
- x_diff_ep → distance between the error layers
- x_diff_L1+X > x_diff_ep → move layer 1 along +ve X axis, layer 2 along –ve X axis
- x_diff_L1-X > x_diff_ep → move layer 1 along -ve X axis, layer 2 along +ve X axis.

In order to return the information about the movement of the error layers, flags are set for both error layers.

- f_L1 → for layer 1 and f_L2 for layer 2.
- The flag values are (1 -1 2 -2) for (+Y –Y +X -X) respectively.

Together with the flags, the distance between the error layers should also be returned. Hence, f_L1, f_L2 and distance are appended into a single TCL list and then returned.

For example, if layer 1 has to be moved along –ve Y axis and layer 2 along +ve Y axis, and the distance between the error layers is 17 db units, then, the list {-1 1 17} is returned.

So, the learn error function, when called takes the original error polygon and the test case error polygons as arguments, and returns a list containing flags (which inform the main program about the error layer movement) and also the distance between the error layers.

3.5 Finding the DRC distance

The final step in learning the error is to know the distance by which an error layer should be moved in order to solve the DRC error. To determine this, the minimum spacing distance (the minimum distance between two metal layers which do not cause a DRC error) between two metal layers should be known first.

Figure 3.1 - Trapezoid shaped error polygon (QR = 50)

PQRS is an error polygon shown in the above figure 3.1. PQ and RS are the set of parallel sides of the trapezoid. According to the DRC rule, the minimum spacing distance for the above case is 50 db units. But, the algorithm has to determine this DRC distance using the error polygon and the DRC function. The DRC deck generates the error polygon in such a way that the lengths of the non parallel sides do not exceed this minimum spacing distance (i.e. 50 db units for this case). So, the lengths of the sides PS and QR have to be calculated and then, the maximum of those two lengths has to be considered. QR = 50 db units (rounded off) for the above case.

The above condition holds true only when there is a considerable difference in between the lengths of the metal layers (or when the metals are placed away from each other by a considerable amount). The lengths of the longest sides (non parallel sides) for the cases shown in the below figures 3.2, 3.3 are 37 and 10 respectively.

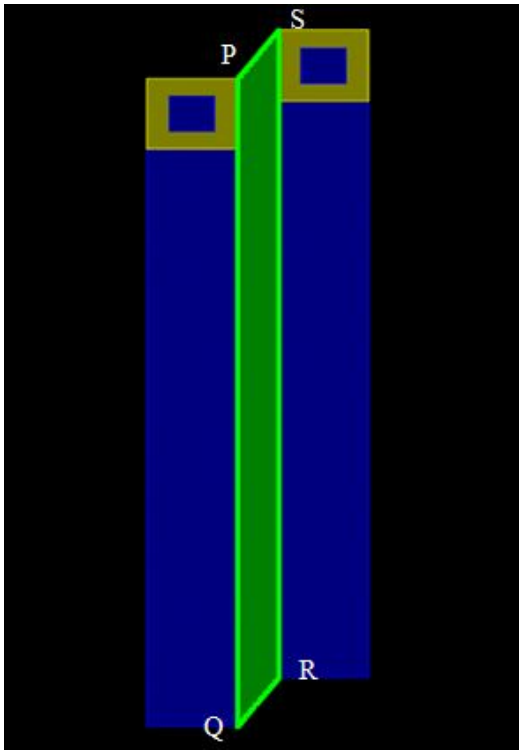

Figure 3.2 - Parallelogram shaped error polygon (QR = 37)

Figure 3.3 - Rectangle shaped error polygon (QR = 10)

So, in order to tackle the above two scenarios, one error layer has to be extended till infinity so that at least one of the lengths (of the non parallel sides of the error polygon) can be equal to the DRC distance (50 db units for this case).

- **Algorithm to find the DRC distance:**

  Practically, a metal layer cannot be extended till infinity. So, it has to be extended up to a point which can be used to output the correct DRC distance.

  The Calibre_LV command “layout bbox” outputs the boundary coordinates of the layout.

  (a0 b0 an bn) → coordinates of the bounding box

  (x0 y0 xn yn) → coordinates of the error layer.

If the metal is running along X-axis, then it is extended till ‘2 * an’ and, if it is running along Y-axis, then it is extended up to ‘2 * bn’. Now, the DRC function is called and the DRC distance can be found from the coordinates of the generated error polygon.

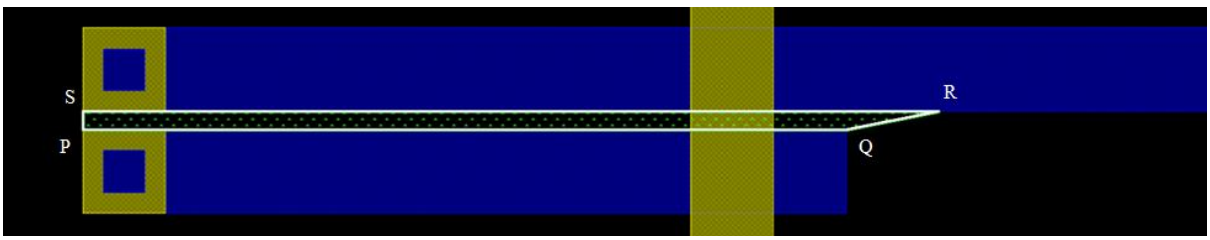

Figure 3.4 - Error Polygon for finding DRC distance

Metal layer 1 corresponding to the rectangular error polygon in figure 3.3 has been extended till ‘2 * an’. After running the DRC again, the error polygon has become trapezoid shaped now making QR = 50 db units which is the nothing but the actual DRC distance.

In order to solve the error, an error layer has to be moved by

a. ‘DRC distance - y_diff_ep’ (for a Y-axis error)
b. ‘DRC distance - x_diff_ep’ (for an X-axis error)

CHAPTER 4
SOLVING THE ERROR

After learning everything about the error, the final part of the algorithm is moving the error layers to solve the DRC error. Together with the error layers, many other layers (such as vias, lower level vias, lower level metals, higher level metals etc.) have to be moved (or extended) in order to solve the error still keeping the layout LVS clean.

- **Notation:**

  M(n) → Metal layer for which the DRC errors have to be solved

  M(n+1) → Higher level metal

  M(n-1) → Lower level metal

  via(n) → Via that connects metals M(n), M(n+1)

  H_via → Higher level via (connects metals M(n+1), M(n+2))

  L_via → Lower level via (connects metals M(n-1), M(n))

  For example, if the DRC errors corresponding to the metal layer M5 have to be solved, then, M(n) = M5, M(n+1) = M6, M(n-1) = M4, via(n) = via5 (connects metals M5, M6), H_via = via6 (connects metals M6, M7), L_via = via4 (connects metals M4, M5)

In a layout, a metal layer M(n) will have connections with lower level metals (M(n-1)) or/and higher level metals (M(n+1)). When this metal layer M(n) is moved to solve a DRC error, it has to be ensured that its connection with other metal layers does not break in order to retain an LVS clean layout. New layers of other metals can also be created in order to ensure that an LVS clean layout is retained (in some cases).

4.1 Different types of connections

1. **via(n)**

   via(n) connects the metals M(n) and M(n+1). So, when the metal layer M(n) is moved, all the via(n)s connected to it have to be moved. For every via(n), higher level metal layer M(n+1) running on top of it has to be found. This metal layer M(n+1) has to be extended (if required) so that the metal envelopes the via(n) after its new placement.

2. **L_via**

   Metal layers M(n-1) and M(n) are connected by an L_via. Together with M(n), all the L_vias connected to it have to be moved. For every L_via, lower level metal layer M(n-1) running below has to be found. It has to be ensured that this metal layer M(n-1) envelopes the L_via after its new placement, by extending the metal layer (if required).

3. **Stacked via connection**

   A stacked via connection occurs when the metal layer M(n) is connected to metals M(n+2) or higher. In this scenario, an H_via is present. H_via connected the metal layers M(n+1) and M(n+2).

   For example, if an M5 layer is connected to M8, then, the metal layers M6, M7 and the via layers via5, via6, via7 are present in between these two layers. Now, if the M5 layer and via5 are moved, then the connection between M5 and M6 (present in the stack) breaks, which results in an LVS failure. A new M6 layer has to be created in between via5 and via6 so that the connection does not break.

   Hence, for a stacked via connection, a new M(n+1) metal layer has to be created between via(n) and H_via to retain an LVS clean layout.

When an error layer is moved, the above 3 scenarios have to be taken care of to ensure that an LVS clean layout is retained.

4.2 Functions used in solving the error

1. **Function to find vias → F(find via):**

   This function takes the error layer extremities and the list of vias as arguments and returns the list that contains vias connected to the error layer. This function has been described clearly in the previous chapter (i.e. Chapter 3).

2. **Function to find a contact layer connected to an error via → F(find contact layer for via):**

   This function takes the coordinates of the error via, the list of metal layers and the error layer extremities as arguments and returns the coordinates of the contact layer. If there is no contact layer, then an empty list is returned. This function has been described clearly in the previous chapter (i.e. Chapter 3).

3. **Function to find higher/lower layer connected to an error via → F(find high/low layer for via):**

   When errors are being solved for a metal layer M(n), via(n)s and L_vias are also moved together with the error layer. It has to be ensured that, the connections between layers M(n), M(n+1) and M(n), M(n-1), do not break. So, these higher/lower level layers connected to via(n)/L_via have to be found and extended (if required).

   a. To find a higher level layer, arguments given → via(n), list of M(n+1) layers.
   b. To find a lower level layer, arguments given → L_via, list of M(n-1) layers.
   c. In both the cases, new error layer extremities and the flag for the error layer (contains the information about the type of error) are also given as arguments.
   d. Error layer new extremities → {a1 b1 a2 b2}
   e. Via extremities → {p1 q1 p2 q2}
   f. Higher/lower layer (called H/L_layer) extremities → {x0 y0 xn yn}
   g. Flag values → (1 -1 2 -2) for (+Y –Y +X -X)

To find whether the H/L_layer is connected to a via or not, the following conditions should be satisfied: x0 <= p1 <= xn and y0 <= q1 <= yn, x0 <= p2 <= xn and y0 <= q2 <= yn.

After finding the H/L_layer, it has to be extended (if required) according to the error flag.

a. Case-1: Flag = 1 (i.e. error layer moved along +ve Y-axis)

i. If yn >= b2 → no need to extend the H/L_layer
   H/L_layer old extremities = H/L_layer new extremities

ii. If yn < b2 → extend H/L_layer till b2
   H/L_layer old extremities = { x0 y0 xn yn }
   H/L_layer new extremities = { x0 y0 xn b2 }

b. Case-2: Flag = -1 (i.e. error layer moved along -ve Y-axis)

i. If y0 <= b1 → no need to extend the H/L_layer
   H/L_layer old extremities = H/L_layer new extremities

ii. If y0 > b1 → extend H/L_layer till b1
   H/L_layer old extremities = { x0 y0 xn yn }
   H/L_layer new extremities = { x0 b1 xn yn }

c. Case-3: Flag = 2 (i.e. error layer moved along +ve X-axis)

i. If xn >= a2 → no need to extend the H/L_layer
   H/L_layer old extremities = H/L_layer new extremities

ii. If xn < a2 → extend H/L_layer till a2
   H/L_layer old extremities = { x0 y0 xn yn }
   H/L_layer new extremities = { x0 y0 a2 yn }

d. Case-4: Flag = -2 (i.e. error layer moved along -ve X-axis)

i. If x0 <= a1 → no need to extend the H/L_layer
   H/L_layer old extremities = H/L_layer new extremities

ii. If x0 > a1 → extend H/L_layer till a1
   H/L_layer old extremities = { x0 y0 xn yn }
   H/L_layer new extremities = { a1 y0 xn yn }

The old and new extremities of the H/L_layer are appended into a single list and then this list is returned.

Hence, the function takes the coordinates of the via, the new extremities of the error layer, flag corresponding to the error layer and the list of higher/lower metal layers, and returns a list containing the old and new coordinates of the H/L_layer connected to the via.

Here, it is assumed that, if the layer M(n) is running horizontally/vertically, then the layers M(n-1), M(n+1) are running vertically/horizontally.

4. Function to find a stacked via connection → F(find H_via) :

In a stacked via connection, metal M(n) is connected to metal M(n+2) or higher. To determine whether this connection is there or not, there should be a H_via for a given via(n).

This function takes coordinates of via(n), list of H_vias as arguments and returns a flag (H_flag) which tells the program about a stacked via connection.

via(n) → {a1 b1 an bn}, H_via → {p1 q1 pn qn}

a. Case-1: via(n) is completely inside or on the boundary of H_via (via(n) size <= H_via size). The conditions required are:
a1 >= p1 and an <= pn, b1 >= q1 and bn <= qn

b. Case-2: via(n) size <= H_via size, but the vias are not exactly overlapping. Required conditions are:
(p1 <= a1 <= pn and q1 <= b1 <= qn) or (p1 <= an <= pn and q1 <= bn <= qn)

If any one of the above two cases are satisfied, then H_flag is set to ‘1’ and returned, otherwise a ‘0’ is returned.

4.3 Steps involved in solving the error

The DRC function when called, executes DRC and generates the error polygons corresponding to a metal layer M(n). For each error polygon, error layers are found and then test cases error polygons are generated and error is learnt by using them. Here, the program assumes that the error is not solved during test case generation and hence none of the test case error polygons are empty lists. After the error is learnt, the following steps are executed to solve the error.

1. **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.

   a. Function F(find via) is called giving error layer extremities and list of via(n) as arguments. Number of connected via(n)s are calculated.
   b. Function F(find via) is called again giving error layer extremities and list of L_via as arguments. Number of connected L_vias are calculated.

   The above two steps are repeated for both the error layers and the number of connected vias (via(n)s + L_vias) are calculated. The error layer having lesser number vias is selected to move first.

2. **Finding the new coordinates:**

   The learn error function gives the distance between the error layers. The DRC distance is also found out by the procedure elaborated in Chapter-3.

   a. Moving distance = (DRC distance) – (distance between the error layers)
   b. Old error layer coordinates → {x0 y0 xn yn}
   c. Based on the flag retuned by the learn error function, the new coordinates should be calculated.
      i. If flag = 1 or -1 → Y-axis error → only y-coordinates should be changed.
         y0_new = y0 + (flag * Moving distance)
         yn_new = yn + (flag * Moving distance)

New error layer coordinates → {x0 y0_new xn yn_new}

ii. If flag = 2 or -2 → X-axis error → only x-coordinates should be changed.

x0_new = x0 + ((flag/2) * Moving distance)

xn_new = xn + ((flag/2) * Moving distance)

New error layer coordinates → {x0_new y0 xn_new yn}

By using the above logic, new coordinates for all other layers (i.e. via(n), L-via, M(n-1), M(n+1)) can also be found. Old coordinates are given to the "layout delete" command and new coordinates to the "layout create" command.

3. Error layer selected in step-1 is moved first.

4. Function F(find via) is called by giving error layer coordinates, list of via(n) layers as arguments. A list of error via(n)s is returned by this function. For every error via(n):
   a. Error via(n) is moved.
   b. Function F(find contact layer for via) is called by giving error via(n) coordinates and list of M(n) layers as arguments. If the function does not return an empty list → contact layer is present. The returned contact layer is moved.
   c. Function F(find high/low layer for via) is called by giving error via(n) coordinates, list of M(n+1) layers, error layer extremities, flag for error layer as arguments. It returns a list containing old and new M(n+1) layer coordinates. If both coordinates are not same, then the old layer is deleted and new layer is created (layer is being extended).
   d. Function F(find H_via) is called by giving via(n) coordinates and list of H_vias as arguments. It returns H_flag. If H_flag is '1', then, a new M(n+1) layer is created between new via(n) and the existing H_via. This new M(n+1) layer can be found using the error layer flag, old and new coordinates of error layer and via(n).

5. Function F(find via) is called by giving error layer coordinates, list of L_via layers as arguments. A list of error L_vias is returned by this function. For every error L_via:
   a. Error L_via is moved.

b. Function F(find high/low layer for via) is called by giving error L_via coordinates, list of M(n-1) layers, error layer extremities, flag for error layer as arguments. It returns a list containing old and new M(n-1) layer coordinates. If both coordinates are not same, then the old layer is deleted and new layer is created.

6. The layout is saved and DRC function is called. Error polygons are generated.
    a. If the number of new error polygons < number of old error polygons, then the error is solved.
    b. If new error polygon >= number of old error polygons, then, error is not solved. It means that moving layer 1 is not the right choice. So, the original layout is again recreated and saved, by interchanging the 'layout delete', 'layout create' commands in steps 3-5 and again executing them.

7. If the error is not solved by moving layer 1, layer 2 is moved and steps 4,5 are executed for layer 2.
8. Again DRC is executed and error polygons are generated.
    a. If the number of error polygons is lesser than the original case, then the error is solved.
    b. If they are greater or equal in number, then the error is not solved. The original layout is recreated again and saved.

9. If the error is not solved by moving layer 2 also, then the error is considered unsolvable and the corresponding error polygon is appended to a list which contains these unsolvable errors.

Thus, the error solving function when called for an error corresponding to a particular metal layer M(n), returns a list containing the unsolved error polygons. If the list is empty, then it means all M(n) errors have been solved. This function is called for all metal layers that have DRC errors in sequence so that the corresponding errors can be solved.

4.4 Flags for the error solving function

The error solving function elaborated above solves errors for each metal layer M(n). But, some of the steps have to be skipped for metals M1, M8, M9. This information has to be passed to the function using flags.

1. **M1_flag:** Condition for M1 layer is being considered here, though the algorithm is for higher level metals which run in a single direction. For M1 layer, L_vias are not present.

   If M1_flag = 1, then, step 5 should be skipped.

2. **M9_flag:**
   For M9, via(n) and H_via are not present.
   If M9_flag = 1, then step 4 has to be skipped.

3. **M8_flag:**
   For M8, H_via is not present.
   If M8_flag = 1, then step 4(d) has to be skipped.

Also, for step 1, only L_via number has to be considered for M9 and only via(n) number has to be considered for M1.

Apart from the arguments that were listed in all the above functions, the layout handle, top level layout name and top level cell name should also be given as arguments to some functions. Different functions and the arguments corresponding to them have been listed in Appendix B.

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 TSMC 28 nm layout.

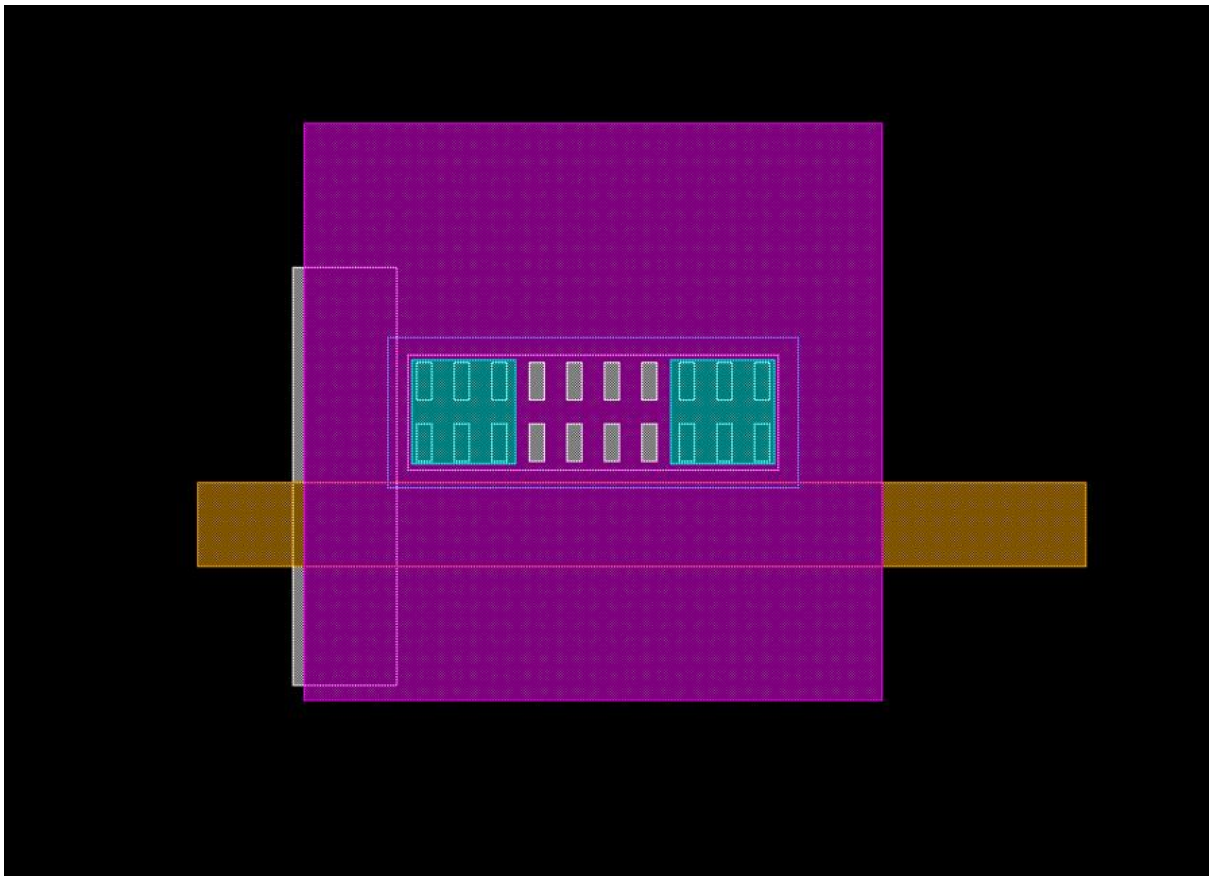

Figure 5.1 - Layout used as a test case

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.

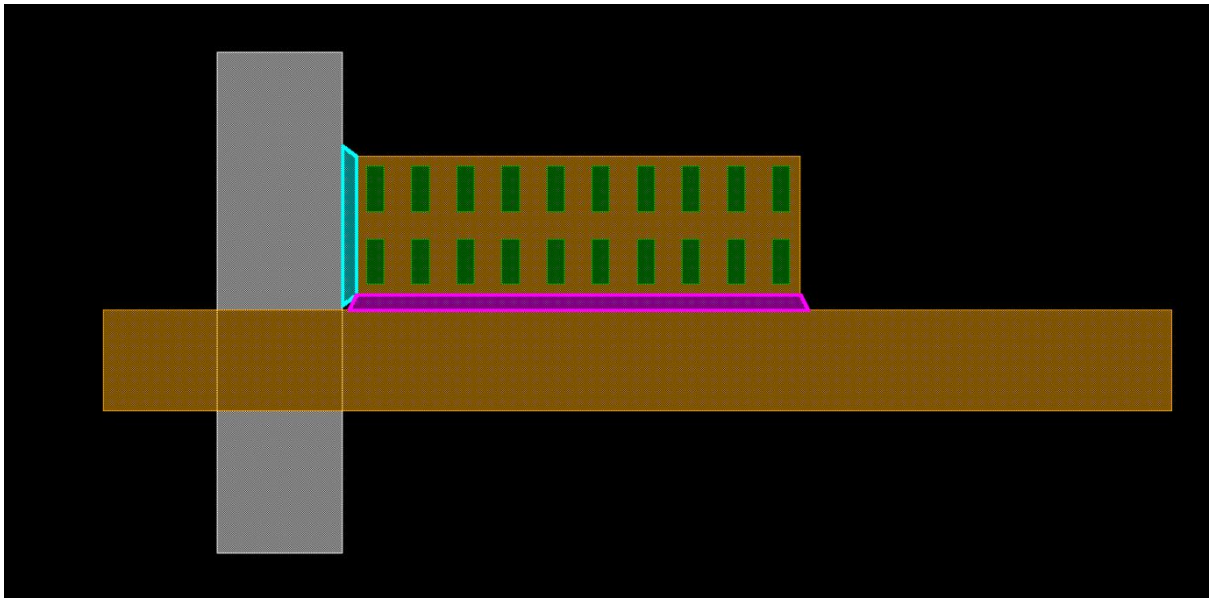

Figure 5.2- Test case layout showing error polygons

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.

a. When layer with less number of vias is moved

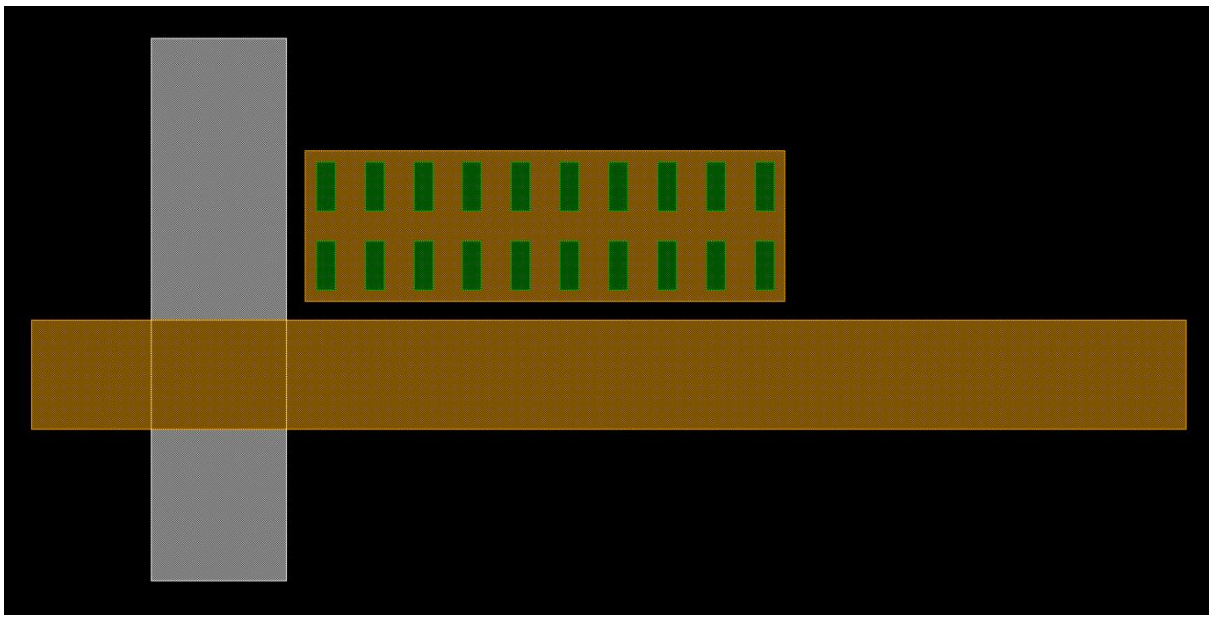

Figure 5.3- Test case layout after running the program

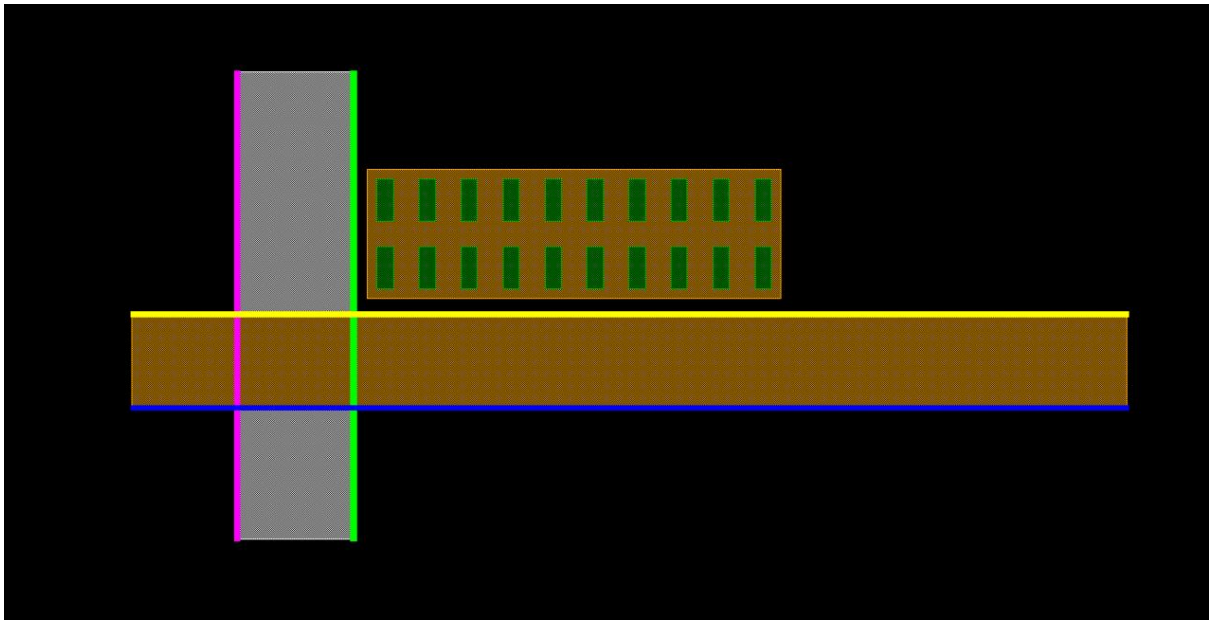

Figure 5.4 - Difference between the solved and unsolved test case layouts

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 layershas 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.

b. 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.

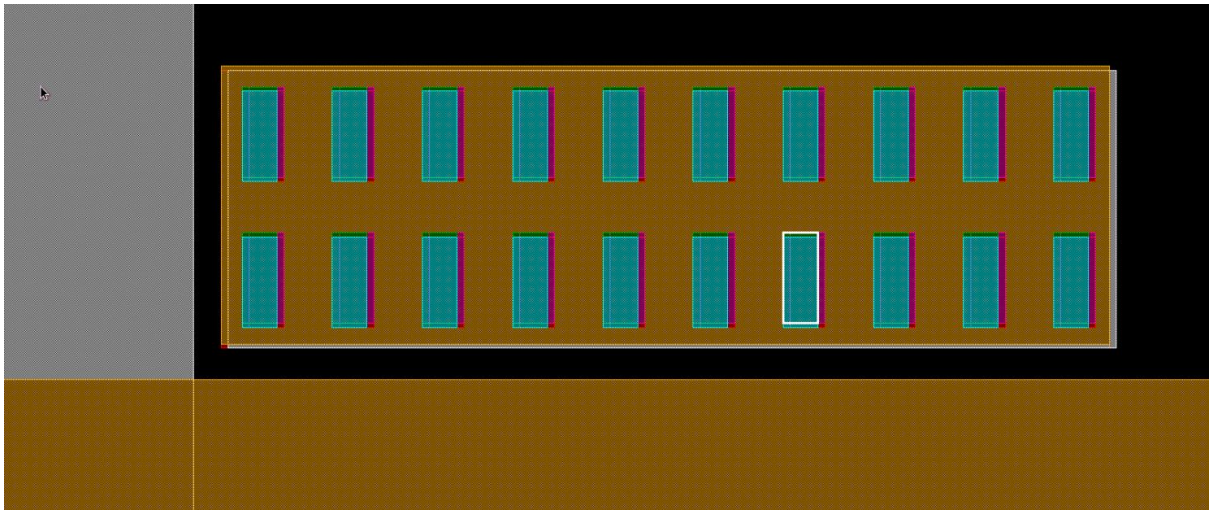

Figure 5.5- Test case layout after stacked via layers are moved

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.

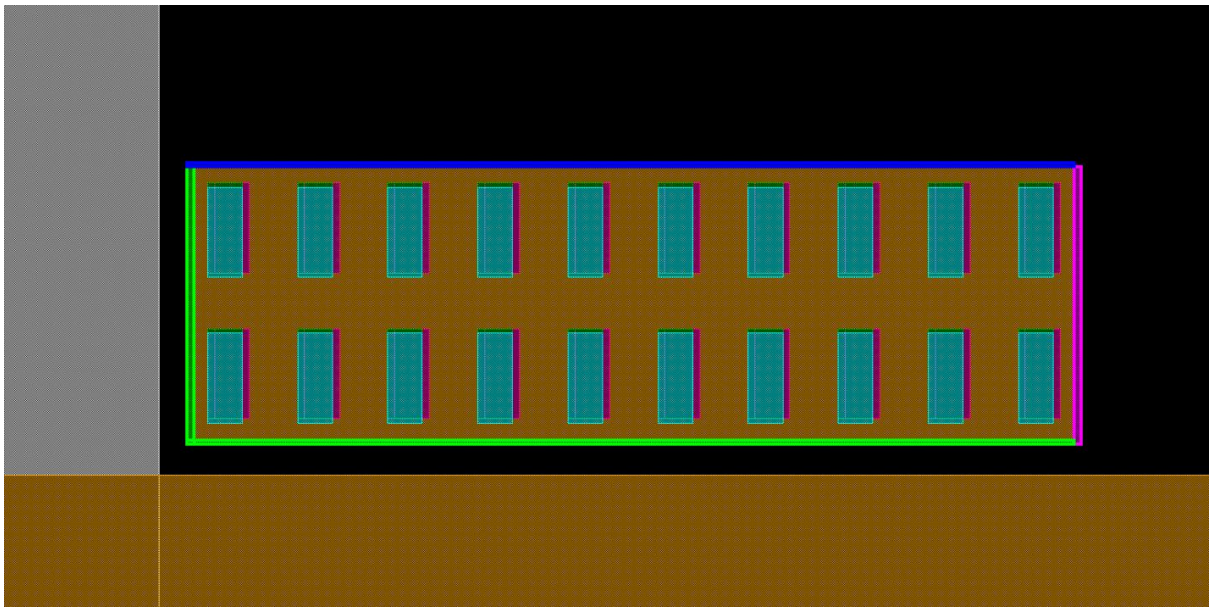

Figure 5.6- Difference between solved and unsolved test case layouts for stacked via

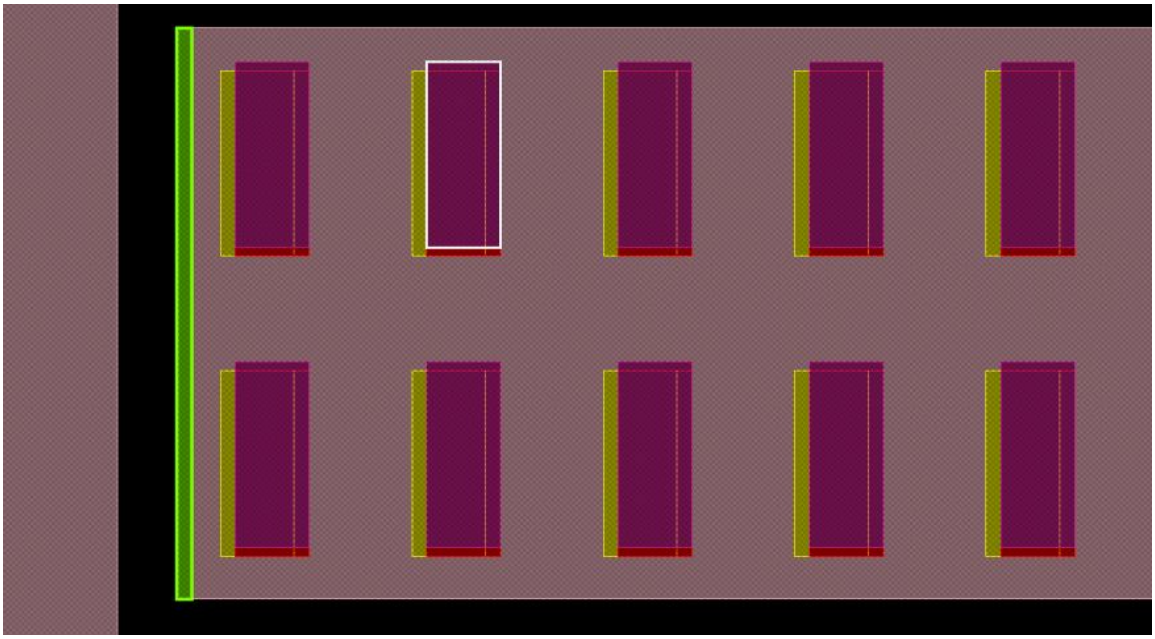

Figure 5.7- Showing M4 layer and the corresponding vias(via4, via3) movement

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.

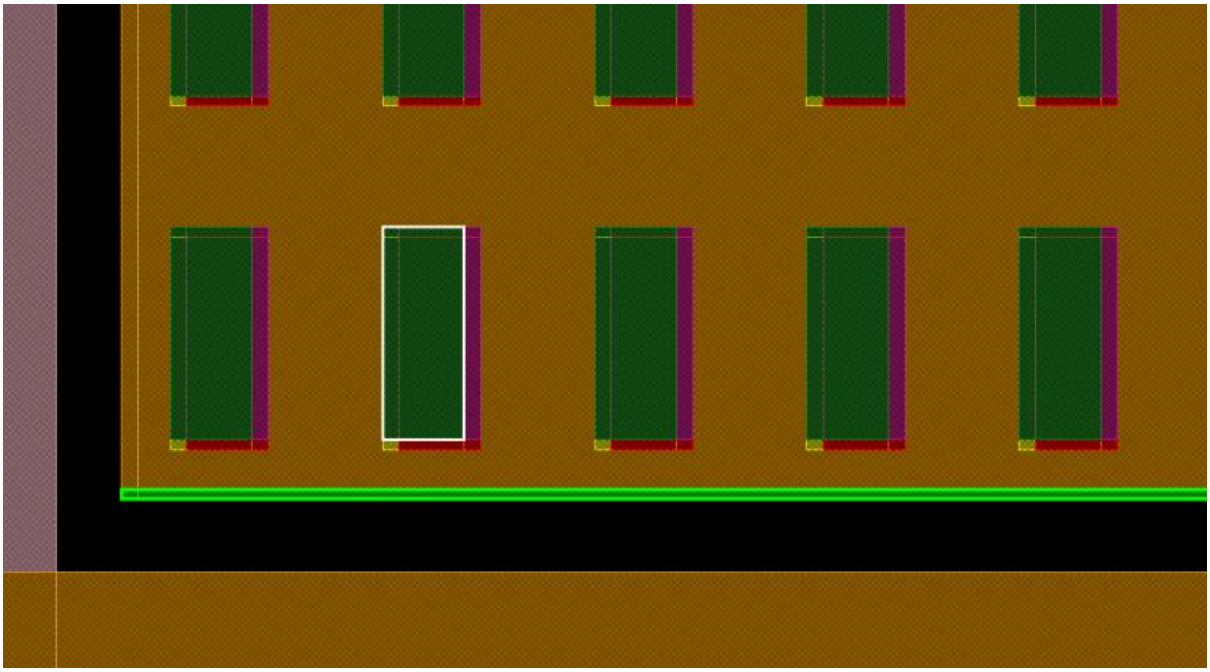

Figure 5.8 - Showing M5 layer and the corresponding vias(via5, via4) movement

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.

APPENDIX A – Calibre_LV Commands

Different Calibre_Lv commands that are used in the program are listed below.

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'

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

$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

$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

APPENDIX B – Functions used in the Algorithm

Functions in TCL are called procedures (proc → procedure). Different TCL functions that are developed in the algorithm have been listed below, together with the notation for variable names that are passed as arguments to the function.

I. Notation:

- error_name → name of the DRC error
- layout_name → layout handle for the gds file opened inside the program
- layout → top level layout name
- cell → top level cell name
- Mn → layer number of the metal (for which errors have to be solved)
- Hn → layer number for the higher level metal (M(n+1))
- Ln → layer number for the lower level metal (M(n-1))
- vian → layer number for the via connecting metals M(n) and M(n+1)
- Hvia → layer number for the via connecting metals M(n+1) and M(n+2)
- Lvia → layer number for the via connecting metals M(n-1) and M(n)
- lst_* → list containing all the * layers [Here * corresponds to a metal or via layer]
- eps → error polygon
- index → index for the given list of error polygons

II. Functions developed

- proc runDRC { error_name } → DRC function
- proc get_eps { gds_name } → Function to find error polygons
- proc find_extrs { layer } → Function to find extremities of a given layer (metal or via)
- proc find_errlayers { eps index lst_Mn } → Function to find error layers
- proc find_error_vias { lay_exts lst_vian } → Function for finding vias connected to a layer

- proc find_connected_contact { via lay_exts lst_Mn } → Function for finding a contact layer connected to a via
- proc tc_gen { error_name gds_name gds_file errexts_Mn lst_Mn Mn lst_vian vian index layout cell layout_name } → Function for finding test case error polygon
- proc learn_err { eps indx tc_eps } → Function for learning the error
- proc find_higher_via { via lst_Hvias } → Function for finding a higher via
- proc findvia_layers { via via_new lst_Mns errlay_new flag } → Function for finding higher/lower layers connect to a via
- proc error_solving {error_name gds_name gds_file layout_name layout cell lst_Mn lst_Hn lst_Ln lst_vian lst_Hvia lst_Lvia Mn Hn Ln vian Hvia Lvia M1_flag M8_flag M9_flag} → Function for solving the error (Here different flags are also given as arguments to find the metal layers)
- Arguments given while running the program script → <gds file name> <config file name> <list of errors file name>
  [should be given in the same order]

REFERENCES

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

[2] Electric User's Manual, version 9.07 (retrieved:). URL https://www. staticfreesoft.com/jmanual/index.html.

[3] TCL language → URL https://www.tcl.tk/