Improving Pattern Matching Accuracy with Duplex Pattern Masking

Yongfu Li, Valerio Perez, I-Lun Tseng, Vikas Tripathi, Zhao Chuan Lee, Jason Khaw, Yoong Seang Jonathan Ong

GLOBALFOUNDRIES
Singapore

https://www.globalfoundries.com

ABSTRACT

Lithography simulation has been the sign-off physical verification tool for checking lithography weak-points, but at the cost of extremely high computational complexity and runtime. To achieve fast turn-around time for our customers, pattern matching based physical verification has been generally adopted in place of lithography simulations. This technology has helped to overcome the limitations of DRC in dealing with multi-dimensional constraints and precisely identifies more localized problematic areas in the layout. The accuracy and runtime heavily depend on the bounding region of a weak-point. A smaller bounding region is adopted in practice because any escapes of real weak-points can be devastating during manufacturing. However, as a result, having to fix false violations can increase development time and degrade timing performance at critical paths. In this paper, we propose a duplex (two-level) pattern matching methodology to improve the accuracy of each violation markers. We have characterized a list of “central”, “false violation” and “Known weak-point” peripheral patterns using lithography simulation. The new pattern matching classification will be based on the overlapping boolean condition of both original and peripheral patterns error markers. When we perform the pattern matching verification through Synopsys ICV, the duplex pattern matching derives the final violation markers based on multiple information from pattern matching result, thus providing more critical information for design fixing or violation waiver.
Table of Contents

1 Introduction .......................................................... 3
  1.1 Problem Statement ........................................... 3
  1.2 Contributions .................................................. 5

2 Architecture .................................................................. 6

3 Implementation .......................................................... 8
  3.1 Pattern Learning .................................................. 8
  3.2 Pattern Matching .................................................. 9

4 Results and Discussion .................................................. 11
  4.1 User Case I: Unit Testing ......................................... 11
  4.2 Pattern Matching Verification ................................. 12
    4.2.1 Pattern Matching using Synopsys ICV command-line Option ........... 12
  4.3 Pattern Matching using Synopsys ICV VUE in IC WorkBench .............. 13
  4.4 User Case II: Improving Pattern Matching Accuracy for 55nm Technology 16

5 Conclusions .................................................................. 17

Appendices ........................................................................ 17

Appendix A Pattern Learning C shell Script ......................... 17
Appendix B Synopsys ICV runset for Pattern Learning .......... 17
Appendix C Synopsys ICV runset for Duplex Pattern Matching ........ 19

References ........................................................................ 22

Table of Figures

Figure 1. Layout images for standard cell A and B with Pattern #1. .......... 3
Figure 2. Our proposed pattern matching workflow with PXL functions. ......... 6
Figure 3. Classification of new and existing pattern matching data. ............. 7
Figure 4. (a) Sample layout for unit testing (b) “central” pattern in the library (c)&(d) hit and false “peripheral” patterns identified through lithography simulation. 12
Figure 5. (a) ICV layout error report (b) Location of error markers in the unit test-case. 14
Figure 6. (a) Graphical user interface of the IC WorkBench (b) ICV error result in ICV VUE interface. .............................................. 15
1 Introduction

After many years of continuous scaling on the CMOS technology without the use of extreme ultraviolet lithography tool, the limits of the 193i optical lithography tool has caused the printing of smaller geometrical sizes to suffer serious degradations in resolution. The classical rule-based Design-Rule-Check (DRC) approach is no longer sufficient to guarantee 100% pattern printability. Design-for-Manufacturability (DFM) compliance checking is required to identify manufacturing weak-points and to prevent catastrophic errors such as open (necking) and shorts (bridging) issues.

Every foundry uses their in-house proprietary model-based lithography simulation tool to obtain accurate printed chip’s images that detect any pattern-related defects (lithography weakpoints). With this great accuracy, it comes at a cost of extremely high computational complexity and runtime. Therefore, chip-level lithography simulations are usually performed only during the tapeout phase. As a result, designers are given only a short period of time to modify the design to eliminate any production failures.

GLOBALFOUNDRIES offers a quick-lithography verification solution, (DRC+) and layout optimization solutions, which is based on pattern matching technology [1, 2, 3, 4, 5, 6]. The pattern library is built from the fab-learned critical lithography weakpoint patterns. Then, the entire design layout is scanned, and locations with layout topologies identical to one of the pattern in the library patterns are identified. Thus, the tool detects lithographic weakpoints very effectively and it is scalable to large chip verifications.

Pattern matching is categorized into exact pattern matching and fuzzy pattern matching techniques. In the exact pattern matching, the weakpoint is reported if and only if there is an exact same pattern in the library. Hence, the exact pattern matching is similar to string searching algorithm, which has been highly-optimized for performance. However, the exact pattern matching is weak in recognizing previously unseen weak-point patterns. To overcome part of the problem, fuzzy pattern matching technique is adopted to identify similar weakpoint pattern in the layout.

1.1 Problem Statement

Figure 1: Layout images for standard cell A and B with Pattern #1.
The window size of the pattern affects the accuracy of weakpoint detection and computation runtime. In our previous work, we proposed a methodology to optimize the window size of each pattern [7]. However, the technique is not suitable in our advanced technology due to an increasing adoption of fuzzy pattern matching and smaller window size to identify unseen lithography weakpoints.

For example, as shown in Figure 1, “Pattern #1” appears in both standard cells A and B with differing pattern topologies at the periphery of the clipping window. “Pattern #1” in standard cell A is a lithography weakpoint but the “Pattern #1” in standard cell B is a “false violation”. In this paper, we propose a duplex (two-level) pattern classification methodology to improve the classification of our fuzzy patterns.
1.2 Contributions

Our contributions are briefly summarized as follows:

1. We propose a robust and efficient duplex (two-level) weak-point pattern matching tool utilizing both central and peripheral pattern information.

2. We demonstrate our implemented prototype system in Section 2.

3. We provide the relevant scripts for our users to adopt our methodology.

The remainder of this paper is structured as follows. We present our proposed duplex (two-level) weak-point pattern classification software and we demonstrated our proposed pattern matching methodology using the Synopsys Programmable, Extensible Language (PXL) in Section 2 and 3, respectively. In Section 4, we provide our verification results on a design testcase through Synopsys IC Validator (ICV) command-line option and Synopsys IC WorkBench. We conclude our work in Section 5.
2 Architecture

As shown in Figure 2, the concept of our duplex (two-level) pattern matching software can be summarized into the following steps:

1. We use our existing pattern library to search for matching patterns in a given input design.
2. We extract the topography pattern with a larger window size\(^1\).
3. We use the GLOBALFOUNDRIES lithography simulation tool to verify each peripheral pattern. If the lithography error marker falls within the peripheral pattern, we classify it as “hit peripheral” pattern. Otherwise, it is classified as “false peripheral” pattern.
4. The central and peripheral patterns are built into the duplex pattern matching library.

\(^1\)We define the original patterns and the patterns with extended window size as central (Level 1) and peripheral (Level 2) patterns.
5. Using the duplex pattern matching library for physical verification, any overlapping of the original pattern and “hit peripheral” patterns is termed as “known weak-point”. User is required to modify the layout to avoid the repeated occurrence.

6. Similarly, any overlapping of the original and “false peripheral” patterns is termed as “false violation”. User can ignore the error markers.

7. For the non-overlapping of original and peripheral patterns, we termed it as “new violation”. User is required to repeat steps (2)-(4).

As illustrated in Figure 3, all the pattern matching error markers identified in our existing design databases can be classified either into “known-lithography weak-point” or “false violation”. The duplex pattern matching classification is based on the overlapping boolean condition of both original and peripheral patterns error markers. When we use the duplex pattern matching software on the new design database, there is likelihood to identify error markers that does not overlap with the peripheral patterns. Thus, lithography simulation is required to determine the classification of the weak-points and the result can be further classified back into the corresponding category of peripheral pattern. With the proposed duplex pattern matching software, we can continuously improve the accuracy of the lithography weak-point pattern library over the time. In addition, our proposed duplex pattern matching software enables user to prioritize and examine more critical error markers and reduce the time to fix all the error markers.

![Figure 3: Classification of new and existing pattern matching data.](image-url)
3 Implementation

The implementation of our duplex pattern matching software uses several C shell scripts, Tcl scripts and Synopsys IC Validator (ICV) runsets, where the ICV runsets are written with the Synopsys Programmable, Extensible Language (PXL) [8, 9]. In this paper, we will focus our discussion on the pattern learning and pattern matching functions.

3.1 Pattern Learning

To translate the peripheral patterns into “hit peripheral” and “false peripheral” library, we use Synopsys ICV to compile these layouts into our ICV pattern library. An example C shell script algorithm for the pattern learning is shown in Appendix A. We use the Synopsys utility, pdb_utility.pl to convert the pattern library into layout for review. An example of the Synopsys ICV runset is described in Appendix B. Typically, the ICV runset comprises of eight sections, which are summarized as follows:

1. User-defined environment variables
2. Call library() to define the design library used
3. Call error_options()
4. Call pattern_options() to define the pattern library path
5. Layer Assignment
6. Define the library information
7. Pattern Learning
8. Output layout database

where the pattern_options() function defines the path for the pattern library and the pattern_learn() function creates a new pattern library or updates an existing pattern library with source patterns.

All the patterns are partitioned into three different libraries based on the lithography simulation results. We propose to use different pattern library names to maintain the different category of our peripheral patterns. Hence, we use the shell environment variables PM_LIB_NAME and PM_LIB_PATH and the ICV function pattern_library() to define the new library location. Code snippet is illustrated in Algorithm 1. Our original patterns are compiled into the “CENTRAL” library. The “known weak-point” and “false violation” patterns are compiled into “PERIPHERAL_HIT” and “PERIPHERAL_FALSE” libraries, respectively.

```
// Step 6: Define the library information
library = pattern_library {
  library_name = $PM_LIB_NAME,
  library_path = $PM_LIB_PATH);
```

Algorithm 1: Code snippet of ICV runset for pattern_library()
3.2 Pattern Matching

The duplex pattern matching verification software is implemented using our proposed Synopsys ICV runset. An example of the Synopsys ICV runset is described in Appendix C. The general concept of duplex pattern matching is comprised of eight sections, which are summarized as follows:

1. User-defined environment variables
2. Call library() to define the design library used in the pattern matching check
3. Call error_options() for error filtering
4. Call pattern_options() to define the pattern library path
5. Layer Assignment
6. Call pattern_match() to perform pattern matching
7. Perform the boolean between central marker and peripheral marker
8. Output layout database

In the duplex pattern matching flow, we invoke pattern_match() function (Appendix C - Step 6) multiple times within the same runset to identify error markers on a design that match the central and peripheral patterns from our pattern library. Then, we perform layer boolean operations on these error markers for duplex pattern matching classification. A code snippet of the ICV runset is illustrated in Algorithm 2. The algorithm uses pattern_match() and interacting() functions to identify the “known weak-point” error marker.
Algorithm 2: Code snippet of ICV runset to identify “known weak-point”
To properly translate the different type of error markers into Synopsys graphical user interface software (ICV VUE), we propose to use error_options() function to classify the error comments from “known weakpoint”, “false violation” and “new violation” into “Fixed”, “Ignore” and “Watch”, respectively. A code snippet of the ICV runset is illustrated in Algorithm 3.

```
// Step 3: Call error_option() for error filtering
error_options(
  error_limit_per_check = ERROR_LIMIT_UNLIMITED,
  classify_by_layer = {
    
    violation_comments = {"*NEW*"},
    classification = "Watch",

    violation_comments = {"*FALSE*"},
    classification = "Ignore",

    violation_comments = {"*KNOWN*"},
    classification = "Fixed"
  }
);
```

Algorithm 3: Code snippet of ICV runset for error_options()

For physical inspection of the error makers, we invoke gds_library() and write_gds() functions to generate an overlay layout with all these marker shapes.

4 Results and Discussion

The experiments were carried out in two parts. In the first experiment, we verify our prototype pattern matching system on our sample design. We also present two methods of executing Synopsys ICV runsets, namely (1) Synopsys ICV command-line option and (2) Synopsys ICV VUE [9]. In the second experiment, we perform pattern matching verifications on several chip-level large designs to help us understand the performance trade-offs for our new prototype system.

4.1 User Case I: Unit Testing

Figures 4(a) and (b) illustrate the layout view of our sample design and one of our lithography weakpoint pattern in the library, respectively. In this experiment, we label the lithography weakpoint pattern as “central” pattern. As shown in Figure 4(a), there are a total of three “central” markers highlighted in the layout. Two highlighted layout topologies have previously observed in our past lithography simulations. We have characterized these layout topologies as hit and false “peripheral” patterns, as shown in Figures 4(c) and (d). The “peripheral” patterns are built into our prototype pattern matching software for verification.
4.2 Pattern Matching Verification

4.2.1 Pattern Matching using Synopsys ICV command-line Option

Our C shell script used for executing the Synopsys ICV command-line option is shown in Algorithm 4. We used shell environment variables to define the three following pattern library, namely (1) “CENTRAL”, (2) “PERIPHERAL_HIT” and (3) “PERIPHERAL_FALSE”. The result is stored in the run_detail directory.

```
#!/bin/csh -f

setenv LAYOUT_PATH "./testcase.gds" ;
setenv FORMAT GDSII;
setenv PM_LIB_PATH "./pattern_library";
setenv PM_CENTRAL_LIB_NAME "CENTRAL";
setenv PM_PERIPHERAL_FALSE_LIB_NAME "PERIPHERAL_FALSE";
setenv PM_PERIPHERAL_HIT_LIB_NAME "PERIPHERAL_HIT";
setenv MATCHING_RUNSET "./pattern_matching.runset";

icv \
  -f $FORMAT \n  -i $LAYOUT_PATH \n  -turbo \n  -verbose \n  -64 \n  $MATCHING_RUNSET
```
4.3 Pattern Matching using Synopsys ICV VUE in IC WorkBench

Synopsys pattern matching can be executed through Synopsys graphical user interface softwares such as IC WorkBench, IC compiler, IC compiler II and custom compiler [10]. In this paper, we demonstrate the pattern matching verification in the IC WorkBench. We need to define the shell environment variable “ICWBEV_USER=IC64” before starting the software. The pattern matching runset is loaded into the ICV VUE and the result can be viewed through the interface as shown in Figures 5. From Figure 5(b), we can clearly see that our proposed pattern matching system is able to classify the error into three categories, namely (1) “Error”, (2) “Ignore” and (3) “Watch”. Figure 6(a) illustrates the graphical user interface of the Synopsys IC WorkBench with out unit test-case. Using the ICV VUE interface, we are able to annotate the errors back to the layout viewer, as shown in Figure 6(b).
Figure 5: (a) ICV layout error report (b) Location of error markers in the unit test-case.
Figure 6: (a) Graphical user interface of the IC WorkBench (b) ICV error result in ICV VUE interface.
4.4 User Case II: Improving Pattern Matching Accuracy for 55nm Technology

To improve the accuracy of our pattern matching library in the 55nm technology, we adopted the proposed duplex pattern matching technique. The experiment is conducted using our Linux workstation with Intel 2.7-GHz 8 Core Duo CPU and 128-GB of memory for consistency in comparison. We have identified four 55nm designs as our testing design databases (layout) while the remaining layouts are used to collect our peripheral patterns. The results are shown in Table 1. As shown in this table, the total area of our four designs ranged from 8.00-mm$^2$ to 30-mm$^2$ and the maximum file size is 6.4-GB. From our experimental result, our proposed duplex pattern matching software has achieved 100% accuracy and each error marker has been visually verified. As expected, there is an increased in the normalized CPU runtime per mm$^2$ from 27.14 to 76.82 seconds/mm$^2$. The real runtime is within the acceptable range (within 5 minutes) compared to the other physical verification checks.

Table 1: Benchmark result on four 55nm designs.

<table>
<thead>
<tr>
<th>Design</th>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>Total Area (mm$^2$)</td>
<td>8.50</td>
<td>6.54</td>
<td>29.91</td>
<td>7.99</td>
</tr>
<tr>
<td>File Size (GB)</td>
<td>2.3</td>
<td>2.6</td>
<td>6.4</td>
<td>0.21$^1$</td>
</tr>
<tr>
<td>Original</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Time (Sec)</td>
<td>808</td>
<td>444</td>
<td>1231</td>
<td>568</td>
</tr>
<tr>
<td>Real Time (Sec)</td>
<td>129</td>
<td>73</td>
<td>200</td>
<td>93</td>
</tr>
<tr>
<td>Violation Layer</td>
<td>N.A.</td>
<td>Metal 1</td>
<td>Poly</td>
<td>Metal 1</td>
</tr>
<tr>
<td>Violation Count</td>
<td>0</td>
<td>4</td>
<td>15</td>
<td>15</td>
</tr>
<tr>
<td>Proposed</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Time (Sec)</td>
<td>1461</td>
<td>758</td>
<td>2043</td>
<td>999</td>
</tr>
<tr>
<td>Real Time (Sec)</td>
<td>247</td>
<td>129</td>
<td>328</td>
<td>160</td>
</tr>
<tr>
<td>Violation Layer</td>
<td>N.A.</td>
<td>Metal 1</td>
<td>N.A.</td>
<td>Metal 1</td>
</tr>
<tr>
<td>Violation Count</td>
<td>0</td>
<td>4</td>
<td>0</td>
<td>15</td>
</tr>
<tr>
<td>Difference</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>CPU Time (Sec)</td>
<td>653</td>
<td>314</td>
<td>812</td>
<td>431</td>
</tr>
<tr>
<td>Real Time (Sec)</td>
<td>118</td>
<td>56</td>
<td>128</td>
<td>67</td>
</tr>
<tr>
<td>Normalized CPU Time (Sec/mm$^2$)$^2$</td>
<td>76.82</td>
<td>48.01</td>
<td>27.14</td>
<td>53.94</td>
</tr>
<tr>
<td>Percentage CPU Time</td>
<td>1.81$\times$</td>
<td>1.71$\times$</td>
<td>1.66$\times$</td>
<td>1.76$\times$</td>
</tr>
<tr>
<td>Violation Count</td>
<td>0</td>
<td>0</td>
<td>15</td>
<td>0</td>
</tr>
</tbody>
</table>

$^1$ Evaluated with a zip file.
$^2$ Normalized CPU time with the total area
5 Conclusions

In this work, we adopt an innovative approach to optimize the pattern matching tool through our proposed two-level hotspot pattern classification flow utilizing both central and peripheral pattern information. We have demonstrated a working model and provided the relevant software code to enhance our users learning. Since Synopsys ICV supports command-line option, we are able to automate the entire process and the system is able to continuously identify new unique patterns. Therefore, the enhanced pattern matching system is able to improve its accuracy over time.

Appendix A Pattern Learning C shell Script

#!/bin/csh -f
setenv PATTERN_LIBRARY "./pattern.gds";
setenv FORMAT GDSII;
setenv PM_LIB_PATH "./pattern_library";
setenv PM_LIB_NAME "PERIPHERAL_HIT";
setenv LEARNING_RUNSET "./pattern_learning.runset";
icv -f $FORMAT \
  -i $PATTERN_LIBRARY \n  -c "*" \n  -turbo \n  -verbose \n  -64 \n$LEARNING_RUNSET
$ICV_HOME_DIR/contrib/pdb_utility.pl \
$PM_LIB_PATH \n$PM_LIB_NAME \n-read ${PM_LIB_PATH}/${PM_LIB_NAME}/central.gds

Algorithm 5: Synopsys ICV command-line option for pattern learning

Appendix B Synopsys ICV runset for Pattern Learning

#include <icv.rh>

// Step 1: User-defined environment variables
#pragma envvar default PM_LIB_PATH "./pattern_library"
#pragma envvar default PM_LIB_NAME "MET"

// Step 2: Call library() to define the design library used
library(
  format = GDSII,
  library_name = "TOP.gds",
  cell = "*"
// Step 3: Call error option
error_options (    
    error_limit_per_check = ERROR_LIMIT_UNLIMITED
);

// Step 4: Pattern Option
pattern_options (    
    pattern_library_path = $PM_LIB_PATH
);

// Step 5: Layer Assignments
metal : polygon_layer = assign( {{1,0}} );
    pattern_marker : polygon_layer = assign( {{2,0}} );
    pattern_extent : polygon_layer = assign( {{3,0}} );
edge_tolerance_layers : polygon_layer = assign( {{4,0}} );
ignore_region_layers : polygon_layer = assign( {{5,0}} );
pattern_text_id : text_layer = assign_text( {{6,0}}, use_exploded_text={{cells ="*"}, text = {"*"}});

error_list : list of write_error_map_s = {};

// Step 6: Define the library information
library = pattern_library (    
    library_name = $PM.Lib_NAME,
    library_path = $PM_LIB_PATH);

// Step 7: Pattern Learning
violation @= {
    pattern_learn (        
        pattern_library_handle = library,
        pattern_layers = {metal},
        pattern_marker = pattern_marker,
        pattern_text_id = pattern_text_id,
        pattern_extent = pattern_extent,
        ignore_region_layers = ignore_region_layers,
        edge_tolerance_layers = edge_tolerance_layers,
        match_ambit = {0.2,0.2,0.2,0.2},
        ambit_mode = PM_MARKER_CENTER,
        pattern_fuzziness = PM_EDGE_UNIFORM,
        uniform_fuzzy_size = 0.004,
        edge_jog_size = 0,
        pattern_reflect = true,
        pattern_rotate = true,
        ignore_extra_polygons = false,
        output_pattern_xml = false,
        anchor = FIRST_PATTERN_LAYER,
        pattern_type = PM_TWO_DIMENSIONAL,
        pattern_naming_mode = PM_NAMESEQUENTIAL,
        pattern_anchor_optimization = PM_ALL
    );
}
Algorithm 6: An example of Synopsys ICV runset for pattern learning

Appendix C  Synopsys ICV runset for Duplex Pattern Matching

```c
#include <icv.rh>

// Step 1: User-defined environment variables
#pragma envvar default PM_LIB_PATH "./pattern_library"
#pragma envvar default PM_TOP_MET "10"

// Step 2: Call library() to defines the design library
// used in the pattern matching check.
library(
    format = GDSII,
    library_name = "TOP.gds",
    cell = "*"
);  

// Step 3: Call error_option() for error filtering
error_options(
    error_limit_per_check = ERROR_LIMIT_UNLIMITED,
    classify_by_layer = {
        { violation_comments = {"*NEW*"},
          classification = "Watch"
        },
        { violation_comments = {"*FALSE*"},
          classification = "Ignore"
        },
        { violation_comments = {"*KNOWN*"},
          classification = "Fixed"
        }
    });  

// Step 4: Call pattern_option() to define the pattern library path
pattern_options(
    pattern_library_path = $PM_LIB_PATH
);  
```
// Step 5: Layer Assignment
MET1 : polygon_layer = assign ( {{15,0}} );
MET2 : polygon_layer = assign ( {{17,0}} );
metal_layer_list : list of polygon_layer = {MET1, MET2};
metal_error_list : list of write_error_map_s = {};

// Step 6: Call pattern_match() to perform pattern matching
for(i = 0 to strtoi($PM_TOP_MET)-1) {
  layer_index : integer = i + 1;
  central_error = pattern_match(
    pattern_library_name = $PM_CENTRAL_LIB_NAME,
    pattern_layers = {metal_layer_list[i]},
    report_orientations = ALL);
  central_layer = marker_merge( layer1 = central_error);
  false_error = pattern_match(
    pattern_library_name = $PM_PERIPHERAL_FALSE_LIB_NAME,
    pattern_layers = {metal_layer_list[i]},
    report_orientations = ALL);
  false_layer = marker_merge( layer1 = false_error);
  hit_error = pattern_match(
    pattern_library_name = $PM_PERIPHERAL_HIT_LIB_NAME,
    pattern_layers = {metal_layer_list[i]},
    report_orientations = ALL);
  hit_layer = marker_merge( layer1 = hit_error);
  pm_hit_layer = interacting(
    layer1 = central_layer,
    layer2 = hit_layer,
    include_touch = ALL,
    count_parity = ALL,
    count_by = SHAPE,
    name = "known weakpoints");
  pm_false_layer = interacting(
    layer1 = central_layer,
    layer2 = false_layer,
    include_touch = ALL,
    count_parity = ALL,
    count_by = SHAPE,
    name = "false violations");
  peripheral_layer = or(
    layer1 = hit_layer,
    layer2 = false_layer);
  new_violation_layer = not_interacting(
    layer1 = central_layer,
    layer2 = peripheral_layer,
    include_touch = ALL,
    count_parity = ALL,
    count_by = SHAPE,
    name = "new weakpoints");
}

// Step 7: Perform the boolean between central marker and peripheral marker
pm_hit_violation @= {
@"METAL"+layer_index+" KNOWN PM RESULT: ";
copy(
    layer1 = pm_hit_layer,
    name = "known weakpoints");
};

pm_false_violation @= {
    @"METAL"+layer_index+" FALSE PM RESULT: ";
copy(
    layer1 = pm_false_layer,
    name = "false weakpoints");
};

pm_new_violation @= {
    @"METAL"+layer_index+" NEW PM RESULT: ";
copy(
    layer1 = new_violation_layer,
    name = "new weakpoints");
};

out_layer_index = layer_index;
metal_error_list.push_back({pm_hit_violation,{out_layer_index,0}});

// Step 8: Output layout database
output_lib = gds_library("out.gds");
write_gds(
    output_library = output_lib,
    errors = metal_error_list);

Algorithm 7: An example of Synopsys ICV runset for pattern matching
References


