EasyMRC: Efficient Mask Rule Checking via Representative Edge Sampling

JIAHAO XU, The Chinese University of Hong Kong, Hong Kong, Hong Kong
ZHUOLUN HE, The Chinese University of Hong Kong, Hong Kong, Hong Kong
SHUO YIN, The Chinese University of Hong Kong, Hong Kong, Hong Kong
YUAN PU, The Chinese University of Hong Kong, Hong Kong, Hong Kong
WENJIAN YU, Dept. Computer Science & Tech., BNRist, Tsinghua University, Beijing, China
BEI YU, CSE, The Chinese University of Hong Kong, Hong Kong, Hong Kong

The photolithography process is getting more sophisticated with technology node scaling down and VLSI designs becoming complex. As photomask patterns get finer, mask rule checks are inevitable to avoid discrepancies in the layout and to ensure manufacturability. This article introduces an efficient mask rule checking approach that utilizes a representative edge sampling scheme. The representative edge sampling scheme selects a subset of edges and points of each polygon that capture its contour, meanwhile greatly reducing the number of edges involved in actual checking. Experimental results demonstrate that the proposed approach achieves significant speedup compared with the state-of-the-art academic tool.

CCS Concepts: • **Hardware** → **Design rule checking;**

Additional Key Words and Phrases: Mask rule checking, sampling, sweepline

ACM Reference Format:
Jiahao Xu, Zhuolun He, Shuo Yin, Yuan Pu, Wenjian Yu, and Bei Yu. 2025. EasyMRC: Efficient Mask Rule Checking via Representative Edge Sampling. *ACM Trans. Des. Autom. Electron. Syst.* 30, 3, Article 47 (April 2025), 19 pages. https://doi.org/10.1145/3723044

1 Introduction

With VLSI designs becoming more complex and technology node scaling down, the photolithography process is also getting sophisticated for wafer production. A photomask is the physical representation of the design, typically generated after resolution enhancement techniques like **optical proximity correction (OPC)**. As photomask patterns get finer, the figures composing the patterns inevitably have the tendency to get smaller due to the high density of the pattern and introduction of various corrections [1]. The requirement of manufacturability of such smaller features

This work was partially supported by the Research Grants Council of Hong Kong SAR (no. RFS2425-4S02 and no. CUHK14211824) and the Natural Science Foundation of Beijing, China (no. Z230002).
Authors' Contact Information: Jiahao Xu, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: jhxu24@cse.cuhk.edu.hk; Zhuolun He, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: zleonhe@gmail.com; Shuo Yin, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: syin22@cse.cuhk.edu.hk; Yuan Pu, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: pyxch11081210@gmail.com; Wenjian Yu, Dept. Computer Science & Tech., BNRist, Tsinghua University, Beijing, Beijing, China; e-mail: yu-wj@tsinghua.edu.cn; Bei Yu (Corresponding author), CSE, The Chinese University of Hong Kong, Hong Kong, Hong Kong; e-mail: byu@cse.cuhk.edu.hk.

ACM 1084-4309/2025/04-ART47
https://doi.org/10.1145/3723044

has enforced a strict set of **mask rule checks (MRCs)** before mask fabrication. Any violations or discrepancies in the design, such as overlapping of components, inadequate spacing between features, incorrect width or length of conductive paths, or other violations of rule constraints, are flagged for further review and correction. Keck et al. [2] have introduced the flow from **design rule checking (DRC)**-clean layout data to mask manufacturing, where MRC appears as a critical stage between reduced verification of OPC'ed data and mask inspection. Moore et al. [3] have pointed out that MRC must provide the basic functionalities including verification, filtering, and reporting. Mason et al. [4] have proposed to standardize mask design rules using a common grammar set, illustrated in *Bacus-Naur* form, where a rule consists of <layer>, <action>, and <value> terms. Kato et al. [5, 6] have developed SmartMRC, which supports typical MRC functions like *space* check, *width* check, and *single point vertex* check. Buck et al. [7] have reviewed the position of MRC in design for manufacturing, where MRC is run to inspect the data for mask manufacturing rule violations prior to mask inspection, and "Do Not Inspect Regions"are created for non-critical unresolvable data regions.

It has been pointed out that the DRC capabilities of EDA tools can be used to form the basis of an MRC system for photomasks [8]. Both DRC and MRC are to ensure a design layout conforming to a deck of geometric rules imposed by the manufacturer. From an algorithmic perspective, the processes basically involve running computational geometry algorithms to analyze the geometric relationship between objects and to identify violation patterns (e.g., points that are within a certain distance threshold) from the layout plane. During the advancement of DRC in the past decades, many fundamental data structures, algorithms, and methodologies have been developed. A series of sweepline-based algorithms are proposed, including rectangle intersection report [9] and Boolean mask operations [10]. Spatial data structures like quad tree [11], KD-tree [12], R-tree [13], and corner-stitching [14] are introduced to efficiently handle layout data. Binning [15] and other layout partition methodologies [16] are widely adopted to improve range search efficiency. To utilize multi-processing hardware platforms to accelerate DRC, many parallel algorithms are also developed [17–21]. Readers may refer to the work of He and Yu [22] for a detailed review of various DRC techniques.

Despite the great similarity between DRC and MRC in terms of task, algorithm, and methodologies, a few critical differences remain. First, academic OPC tools often produce a pixel-based image as their solution, whereas the original layout data are usually in standard industrial format like LEF/DEF or GDSII. Second, since the OPC algorithm tends to compensate for the diffraction and proximity effects in the lithography process [23], mask pattern shapes are distorted to minimize edge placement error, process-variation band, and so on, resulting in much more irregular geometries. Figure 1 illustrates the difference between DRC and MRC. Essentially, in MRC, polygon edges are no longer rectilinear: they can appear in any angle. Moreover, many short segments exist in the artificial zig-zag shape contours. Therefore, typical DRC approaches alone are inadequate to handle mask rule checking.

This article proposes EasyMRC, an efficient mask rule checking methodology. EasyMRC takes a post-OPC layout as input and reports all mask rule violations in the layout. EasyMRC utilizes a representative edge sampling technique, which samples a subset of edges and points of each polygon that best captures its contour while eliminating most redundant checks involved. A sweepline-based scheme is accordingly designed to correctly identify all violations without missing any. Our contributions are summarized as follows:

— We propose a representative edge sampling technique tailored for MRC that greatly improves its efficiency.
— We design a sweepline-based algorithm used together with the representative edge sampling scheme, so that all violations will be completely reported.

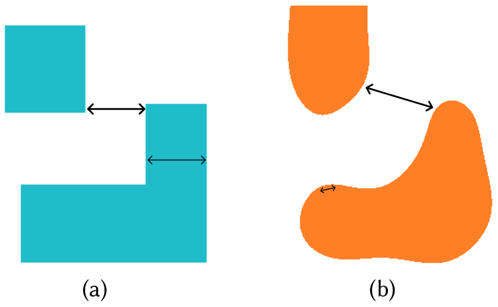
Fig. 1. Comparison between DRC (a) and mask rule checking (MRC) (b). Distance in all directions has to be considered in MRC.

— We achieve a significant speedup on MRC, compared with the state-of-the-art academic tool.

The rest of the article is organized as follows. Section 2 introduces related preliminaries. Section 3 gives problem formulation. Section 4 describes EasyMRC algorithms in detail. Section 5 demonstrates experimental results. Section 6 concludes the article.

2 Preliminaries

2.1 Optical Proximity Correction

In the field of semiconductor manufacturing, small feature sizes on chips leads to the effects of optical diffraction and light scattering during the lithography process. These effects result in distortions in the printed patterns. Serving as an important resolution enhancement technique in the lithography process, OPC analyzes the optical characteristics of the lithographic system and the wafer process, and determines the optimal adjustments to the mask shapes. These adjustments aim to counteract the optical diffraction, lens aberrations, and other phenomena that can introduce distortions to the printed pattern. Essentially, OPC adjusts the shapes on the photomask to ensure that after the light exposure and development stages, the shapes on the silicon wafer match the intended design as closely as possible.

2.2 Mask Rule Checking

The mask designs usually have to be verified for a set of predefined manufacturing rules after being sent to the mask shops. These rules encompass various aspects of the mask layout, such as minimum feature size, spacing, width-to-space ratios, and other geometrical constraints. Violating these rules can lead to yield loss and reliability problems. Today, many of the advanced OPC techniques generate curvilinear mask shapes rather than pure rectangular shapes. Therefore, the type of rules changes at the same time, and efficient techniques for checks are required [24]. This enables designers to make necessary adjustments to the layout, ensuring compliance with the manufacturing rules and optimizing the yield, and overall performance of the IC. Regarding spacing violations between shapes, which is the focus of this work, one key difference from rectangular shapes is that the minimum distance violating the spacing rule is not necessarily perpendicular to the edges. In earlier rule definitions, the minimum width (or spacing) typically considered only horizontal and vertical distances, checks in other directions referred to as internal (external) "corner-to-corner" measurements. However, this does not apply to curved geometries. In this work, the spacing check and width check consider the distance between shapes in all directions and identify any violations of the spacing rule. It should be noted that current OPC tools output pixel-level masks rather than analytical expressions of the curvilinear contours. Therefore, we still have to format the mask into a segment-wise GDSII file with horizontal and vertical edges before further processing.

2.3 Sweepline

The sweepline algorithm is a powerful technique used in various fields. It involves sweeping a line across a set of objects or events to efficiently process and solve geometry-related problems. The algorithm maintains a data structure, often a binary search tree, to keep track of the objects intersecting or being encountered by the sweepline as it moves. By carefully choosing the order of processing events and efficiently updating the data structure, the sweepline algorithm can solve problems such as segment intersection, polygonal decomposition, and closest pair of points. The sweepline algorithm has been used in various tasks of DRC, including the spacing violation between objects. Due to the $O(\log(n))$ complexity for each operation of updating the binary search tree or extracting information from it, the sweepline algorithm in these tasks can usually achieve an expected time upper bound of $O(n\log(n))$. Many techniques have been proposed to accelerate these processes. In this article, we will customize the sweepline algorithm based on the characteristics of MRC tasks to achieve acceleration.

3 Problem Formulation

Problem 1 (Format Conversion). *Given a pixel-based graphic generated by OPC tool, the task is to convert it to a segment-based formatted file.*

Problem 2 (Spacing Rule Checking). *Given a formatted post-OPC layout, the task is to report all spacing rule violations. The spacing rule is defined based on the geometric relationship between objects. In this work, it requires that the minimum distance between any two polygons is not less than a threshold, and we report the locations where the distances violate this threshold. The spacing rule is a most common and fundamental rule in mask optimization.*

Problem 3 (Width Rule Checking). *Given a formatted post-OPC layout, the task is to report all width rule violations. In this work, width rule checking requires the minimum distance between any two opposite edges of one polygon is not less than a threshold, and we report the locations where the distances violate this threshold. The width rule is a most common and fundamental rule in mask optimization.*

4 Algorithm

Figure 2 illustrates the overall flow of EasyMRC. Given an OPC'ed mask image, we first convert it to a standard GDSII format file, where the boundaries of all polygons are saved in the form of segments, facilitating subsequent processing and being imported into the state-of-the-art tool, which is introduced in Section 4.1. Next, the sampling algorithm introduced in Section 4.3 will filter out representative points and edges and record some necessary information for use in our custom sweepline algorithm later on. The preceding procedures are considered *preprocessing* of mask rule checking. Then, if the task involves space checking, pairs of polygon candidates will be generated by a standard sweepline algorithm introduced in Section 4.2. Subsequently, the custom sweepline algorithm described in Section 4.4 will be executed for each candidate pair. If the task involves width checking, the sweepline algorithm introduced in Section 4.5 will be directly executed within each polygon. Finally, we provide a complexity analysis of the algorithms in Section 4.6.

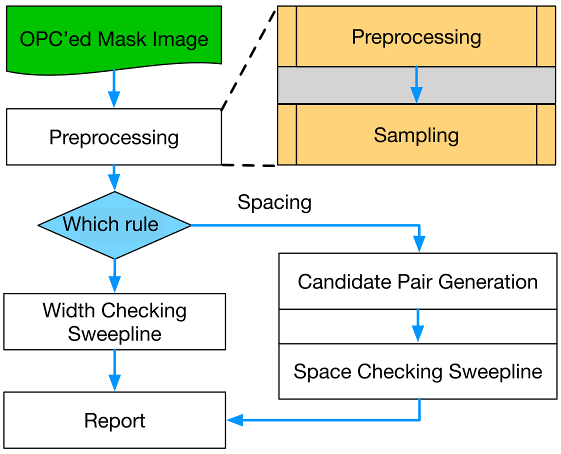

Fig. 2. Overall flow of mask rule checking.

4.1 Format Conversion

To accurately convert a pixel-wise image generated by the OPC process into a segment-wise GDSII file, we apply a Format Conversion algorithm.

Due to the use of pixel-based graphics methods in many advanced OPC algorithms, fitting a series of short edges with low-order functions becomes challenging. Since this work focuses solely on spacing and width violations within a single layer, we assume that the input mask consists of a binary image in one layer. Therefore, we directly record the horizontal and vertical edges of each polygon with precision.

After importing the PNG file, we scan from the bottom-left corner to the top-right corner. If we encounter a mask pixel on the edge that has not been visited, we start to construct a new polygon. Figure 3 shows how the process works. The black arrows represent the pixels being searched, and the thick boundary lines represent the segments being generated. We search clockwise along the edge of the polygon and change the direction whenever encountering a corner, and record the segment in a list. Note that pixels have area, so at corners, it is necessary to judge which endpoints to connect based on the situation. When a pixel is searched, it is recorded to ensure that all the polygons will only be built once. Therefore, all the pixels will only be visited or searched once, consuming a linear time complexity. After the process is over, we output the GDSII file since all the edges for each polygon have been saved in a list.

4.2 Candidate Pair Generation

This process is only necessary for spacing checks. For ease of explanation, we introduce it before discussing the sampling method. To report spacing violations effectively among polygons, it is essential to first identify potential rule-violating pairs which are spatially close. This step avoids the complexity of discerning if points in the segment tree belong to the same polygon, when applying the sweepline algorithm. To facilitate this, bounding boxes are employed to find pairs of polygons which are spatially close. The process to detect bounding box overlaps using the sweepline algorithm includes four steps. Initially, a bounding box whose width and height are determined by the distance of violation is generated for each polygon, and two *events* are then generated on the left and right edge for the bounding box. Subsequently, these events are *sorted* by the x-coordinate of the edges, with lower edges given precedence in cases of equal x-coordinates. Next, an empty *segment tree* is set up to track active bounding boxes as the sweepline moves. The final step involves

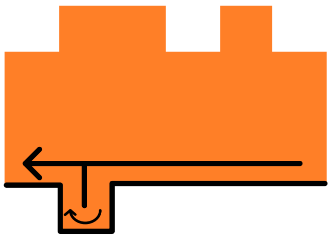

Fig. 3. Extracting segments from an image.

sweeping all events from left to right. Each event is processed in turn: for a left-edge event, overlaps of intervals within the segment tree are checked, and the overlap of each pair of polygons is recorded. The interval is then inserted into the segment tree. Conversely, a right-edge event involves removing the interval from the tree.

By following this procedure, the sweepline algorithm efficiently identifies and records the overlaps between bounding boxes. In space checking, we apply sweepline to all these pairs respectively.

4.3 Sampling

As previously introduced, mask patterns are irregular geometries with many short segments in the zig-zag contours. Our motivation is to pick a subset of these segments so that the shapes are still effectively captured, violations are correctly reported, and the number of involved checks are reduced. Therefore, we have designed a *sampling* method to select representative edges and points. The idea is straightforward: in a smaller circular area, only the central vertices participate in the complex search processes. The other points and edges within the range are checked using faster methods, thus reducing the overall algorithm runtime.

The Sampling Method. We pick representative points and edges in this process. A sampling radius r is first defined. Points and edges within r from a representative point will be *shielded* by it. To dynamically adapt to masks of different sizes, r is chosen as a multiple of l, where l represents the average length of all edges. According to our experiments, $r = 4l$ serves as a good setting. For each polygon, we scan the vertices along the polygon boundary starting from a starting point. After selecting the first vertex, we choose the next representative point that is as far away as possible, ensuring that all points between the two are shielded. Edges with lengths greater than the sampling radius are sampled. Only these representative points and edges will directly interact with the binary search tree during the sweepline phase. Figure 4 illustrates the sampling by an example, where the red circles indicate representative points, and the gray circles demonstrate the sampling radius as well as the shielded points inside.

Guarantee of Correctness. The sampling approach should ensure that any existing violation will be correctly identified. To show that this is indeed the case, let us start the discussion without applying sampling method. Since a violation is equivalent to the minimum distance between two edges, the violations fall into two categories.

THEOREM 1. *There are and only are two possibilities of a violation between two edges (Figure 5):*

(1) *it is between endpoints of two edges, or*
(2) *it is between an endpoint of one edge and a point on another.*

PROOF. The first situation occurs when two edges have no intersecting projection, and the second situation occurs when two edges have non-empty projection. Here the projection is checked

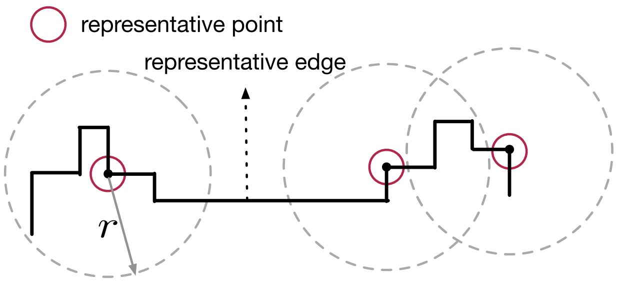

Fig. 4. An example for representative points and edges.

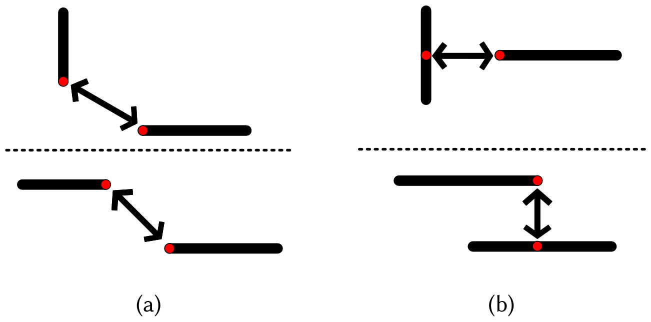

Fig. 5. Two types of violation before sampling.

on the coordinate axis parallel to the edges when they are parallel, whereas it is checked on both axes when the edges are perpendicular. □

Obviously, a sweepline algorithm reports all the violations, including both categories, as long as we do not sample.

Our customized sweepline algorithm after sampling only insert representative points into the segment tree or query on the segment tree using them as central points. To avoid missing violations between points and edges within their range, we set an "extended rule"

$$R' = R + 2 * r, \tag{1}$$

where R is the original rule distance that generate violations between points and edges. Representative point pairs within R' will be directly found in the segment tree, thereby preventing the omission of violations between the points and edges they shield. Furthermore, we categorize the sources of violation into two types. These two types of violations, illustrated in Figure 6, require different treatments later in the sweepline process.

Theorem 2. *Let $\mathcal{SP}(p)$ denote all the points or edges within r of a representative point p. There are and only are two possibilities of a violation after sampling:*

(a) *It is the distance between $\mathcal{SP}(p_1)$ and $\mathcal{SP}(p_2)$, including type (1) and type (2) distance mentioned previously. p_1 and p_2 are two representative points.*
(b) *It is the distance between $\mathcal{SP}(p_1)$ and a representative edge, with only type (2) distance being applicable. Here, p_1 is a representative point.*

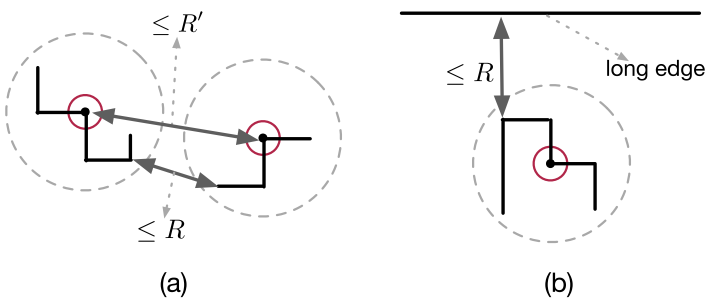

Fig. 6. Two types of violation after sampling.

Proof. In Theorem 1, we defined two categories of violations, without any omissions. The following demonstrates that identifying all violations of types (a) and (b) is equivalent to identifying all violations of types (1) and (2).

If two points described in (1) violate rule R, their corresponding representative points will also violate R'. Therefore, violations in (1) are encompassed within violations in (a). If the two edges described in (2) are not representative edges, they are shielded by representative points, and as such, they are similarly included in violations of type (a). If one of the edges is a representative edge, this situation specifically falls into type (b), and the corresponding representative point must violate the extended rule $R'' = R + r$ in conjunction with the representative edge. Finally, if both edges are representative edges, their endpoints must have been sampled. This implies that at least one violation will be reported between an endpoint of one edge and the other edge, categorized as type (b). □

Therefore, to ensure that all violations are identified without omission, we must query all pairs of representative points that violate the extended rule within the segment tree, and subsequently check all corresponding point-edge pairs within their sampling radius r. The same procedure should be applied to pairs consisting of a representative point and a representative edge.

4.4 The Sweepline Algorithm for Space Checking

We have come up with a customized sweepline algorithm to accommodate the sampling scheme. In particular, we have to run the sweepline several times, as there are slight differences in handling the two kinds of violations previously defined. Because space checking is more complex than width checking, let us first discuss the implementation method for space checking. Afterward, it can naturally be extended to width checking.

First, we describe the algorithm for handling type (a) violations. Recall that the extended spacing rule threshold is defined as $R' = R + 2r$. For each candidate pair of polygons, we select the sampled points within the extended bounding box (i.e., the bounding box inflated by R' in all four directions) of the other polygon. Next, we sort these representative points by their x-coordinates. Points with the same x-coordinate are sorted by their y-coordinates to resolve ties. We then construct two *segment trees* for the two polygons, each maintaining the points along the y-coordinates. Now, sweep through the points from left to right. For each point p encountered,

(1) Remove from the segment trees any points whose x-coordinates are smaller than p by at least R'.
(2) Search in the segment tree of the other polygon for points whose y-distance from p is smaller than R'. If any such point q is found, check whether any of the distances between $\mathcal{SP}(p)$ and $\mathcal{SP}(q)$ is smaller than R. These are the violations that need to be reported.
(3) Insert p into the segment tree.

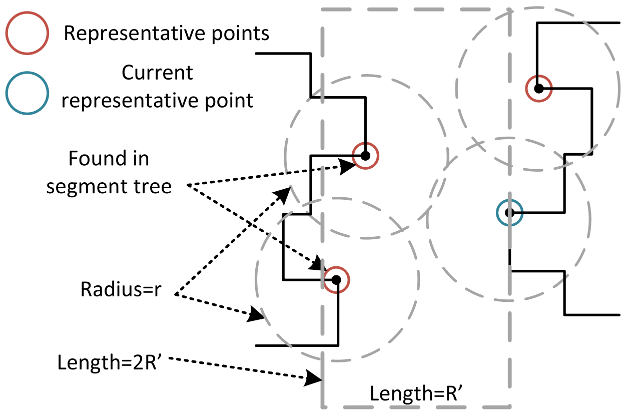

Fig. 7. Search range in data structure during sweepline for type (a) violations.

Note that for (1), the points can be safely removed because they will no longer yield violation pairs with the points encountered in the future due to the total order between all points.

From a geometric perspective, we found all representative points in the left half of a square with a side length of $2 * R'$ centered at each representative point, which is shown in Figure 7. Therefore, we actually checked all the points and edges (whether sampled or not) of the other polygon in the left half square centered at the current representative point, with the points and edges within the sampling radius of the current point. After this process, we identified all violations of type (a) and some violations of type (b), specifically those occurring between $\mathcal{SP}(p)$ and $\mathcal{SP}(q)$ for any p and q. Note that there is no need to apply the algorithm from right to left or from bottom to top, as a violation must involve one representative point on the left and one on the right, with the right one playing the role of the current point. Algorithm 1 summarizes the overall flow of space checking sweepline algorithm.

To find the remaining type (b) violations, we only need to find the violations between points and representative edges, so the point must be covered in the projection of the edge; otherwise, the case is already handled in type (a). Now set the extended rule distance to $R'' = R + r$. The vertical representative edges are split into two events. Suppose its x-coordinate is x_0, then the two events are set at x_0 and $x_0 + R''$. For the horizontal representative edges, suppose its right endpoint has x-coordinate x_0 and the event is set at $x_0 + r$. Sort the representative points and the events in ascending x-coordinates. Similarly, initialize two segment trees for the two polygons to maintain points along the y-coordinate axis. Sweep through the points and edges from left to right. For each point or event encountered:

(1) Remove from the segment trees those points whose x-coordinates are smaller than p by at least $R' = R + 2r$.
(2) • If it is a point, insert it into the segment tree.
  • If it is an event of a vertical edge, search in the segment tree for the points that are covered by the projection of the edge. Due to the sampling radius, the actual search range should be expanded by r. If a representative point p is found, check $\mathcal{SP}(p)$ with the current edge. Report any found violation. The events at x_0 and $x_0 + R''$ will identify the violations on the left and right side of this edge, respectively.
  • If it is an event of a horizontal edge, suppose its y-coordinate is y_0, then search in the segment tree for range $[y_0 - R'', y_0 + R'']$. If a representative point p is found, check $\mathcal{SP}(p)$ with the current edge. Report any found violation.

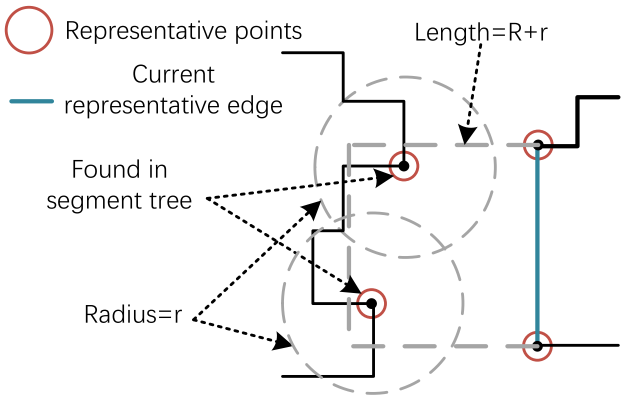

Fig. 8. Search range in data structure for the remaining type (b) violations.

ALGORITHM 1: Space Checking Overall Flow

```
Input:  Post-OPC layout, minimal spacing $R$, sampling radius $r$
Output: Violation pair list $\mathcal{S}$
 1: Convert layout image to GDSII;                                      ▷ Section 4.1
 2: Find candidate polygon pairs $\mathcal{C}$;                          ▷ Section 4.2
 3: Sample representative points $\mathcal{P}$, edges $\mathcal{E}$;     ▷ Section 4.3
 4: for polygon pair $(p_1, p_2)$ in $\mathcal{C}$ do
 5:     Sort $\mathcal{P}(p_1) + \mathcal{P}(p_2)$ in ascending $x$-coordinates;
 6:     Initialize two segment trees for $p_1$ and $p_2$, respectively;
 7:     for each point $v$ in the sorted representative points do
 8:         Remove the points whose $x$-coordinates are smaller than $v$ by $R'$ from the trees;
 9:         Insert $v$ into the tree of its polygon at its $y$-coordinate;
10:         Query in the other tree for $[y - R', y + R']$;
11:         for each representative point $v_i$ found in the range do
12:             Check the distances between $\mathcal{SP}(v_i)$ and $\mathcal{SP}(v)$;
13:             Record the violations;
14:         end for
15:     end for
16:     Identify type(b) violations in direction $d$;                   ▷ Algorithm 2
17: end for
18: return violations list $\mathcal{S}$;
```

The query range of the left event of a vertical edge is illustrated in Figure 8. The right event of a vertical edge will query the mirrored range on the right side. As for a horizontal edge, the search range is the corresponding upper and lower sides. All the cases are correctly handled in this way. The program flow for type (b) violations is as shown in Algorithm 2. It should be noticed that the flows of maintaining representative points in Algorithm 1 and Algorithm 2 are the same. The only difference is to handle the events of representative edges together. Therefore, we can directly merge the two flows, thus completing the whole algorithm in one go.

4.5 Width Checking

Width checking differs slightly from space checking. First, width checking occurs within all polygons, so there is no need for preprocessing of candidate pairs. Second, we aim to ensure that the

ALGORITHM 2: Identify the remaining type (b) violations

```
Input:  Representative points $\mathcal{P}$, edges $\mathcal{E}$
Output: Type (b) violation pair list $\mathcal{S}'$
 1: Create events set $E$ for representative edges.
 2: Sort $\mathcal{P}(p_1) + \mathcal{P}(p_2) + E$ in ascending $x$-coordinates;
 3: Initialize two segment trees for $p_1$ and $p_2$, respectively;
 4: for each point $v$ or event $e$ in the sorted representative points do
 5:     Remove the points whose $x$-coordinates are smaller than $v$ by $R'$ from the trees;
 6:     Insert $v$ into the tree of its polygon at its $y$-coordinate;
 7:     Query in the other tree for the corresponding range of $e$;
 8:     for each representative point $v_i$ found in the range do
 9:         Check the distances between $\mathcal{SP}(v_i)$ and the corresponding long edge of $e$;
10:         Record the violations;
11:     end for
12: end for
```

minimum internal diameter in all directions does not fall below the rule. However, two adjacent edges that are perpendicular to each other should obviously not be considered. Therefore, we stipulate that only edges facing in opposite directions and too close to each other will be reported.

In summary, we can retain most of the process from the spacing checking algorithm, as it accurately identifies all pairs of edges with distances below the threshold. However, only those pairs of edges that are facing in opposite directions will be recorded as a violation.

4.6 Complexity Analysis

We analyze the runtime complexity of previously described procedures.

Format Conversion. All pixels will only be visited once, so the complexity is $O(m)$, where m is the number of pixels in the image.

Candidate Pair Generation. This process is to find the polygon pairs that have overlapping bounding boxes. The boxes are rectangles, so the time complexity is $O(p \log p)$ by the sweepline framework, where p is the total number of boxes (polygons). Note that if p is not too large, we can simply maintain an active set with a linear data structure to cut down the constant coefficient.

Sampling. The time complexity is $O(N)$, where N is the total number of endpoints, because sampling is done in one loop for each polygon to select the representative points from the endpoints.

Sweepline. Since we have to consider the *corner to corner* cases, one can easily construct examples reaching the worst time complexity of $O(n^2)$ for each pair of polygons found in candidate pair generation. Here, n is the number of endpoints of the two polygons (assuming they have the same scale). But we can make some assumptions without losing generality: let R, r, l represent the distance of violation, the sampling radius, and the average length of the shielded edges, respectively. Then let us assume that in a half square described in Section 4.4, the number of endpoints is at the scale of $O(\frac{R+2r}{l})$, which visually indicates that the shape of a polygon does not "fold repeatedly."

For the algorithm addressing violations of type (a), we can estimate that the worst-case time complexity of searching for violations in the segment tree, for each representative point acting as the "current point," is given by $O(\frac{(R+2r)\log L}{lr} + \frac{R+2r}{l} * \frac{r}{l})$. Here the first part $\frac{(R+2r)}{lr} * \log L$ comes from querying all the representative points in the segment tree. $\frac{(R+2r)}{lr}*$ stands for the scale of representative points in the search range. All points within the sampling radius of the identified points are then checked individually against the points within the sampling radius of the current point. This process contributes to the second term $\frac{R+2r}{l} * \frac{r}{l}$, where $\frac{r}{l}$ represents the scale of shielded points and edges associated with a single representative point. L is the length of the interval in the segment tree, which can be discretized to the same scale as n. Thus, the worst-case total time complexity is

$$\begin{aligned} O\Bigg(\frac{n}{r/l} * \Bigg(\frac{(R+2r)\log(L)}{lr} + \frac{R+2r}{l} * \frac{r}{l}\Bigg) \quad & \triangleright \text{ query \& check} \\ + \frac{n}{r/l} * \log(L) \quad & \triangleright \text{ insertion} \\ + T_1\Bigg) \quad & \triangleright \text{ \#violations,} \end{aligned} \tag{2}$$

where $\frac{n}{r/l} * \log(L)$ accounts for inserting or removing representative points into or from the segment tree, and $\frac{n}{r/l}$ represents the scale of representative points. T_1 denotes the number of violations detected during this process. Typically, we only need to determine whether a violation exists within the range of a pair of representative points. Therefore, the point-to-point check process can terminate and report as soon as the first violation is found. And thus the cost of finding a violation is $O(\log(L) + \frac{R+2r}{l} * \frac{r}{l})$.

The sweepline to find violations of type (b) similarly has a time complexity of

$$O\left(k * \left(\frac{(R+2r)\log(L)}{lr} + \frac{(R+2r)}{l}\right) + \frac{n}{r/l} * \log(L) + T_2\right), \tag{3}$$

where k is the scale of the representative edges.

In width check, the preceding processes take place in each polygon rather than between the candidate pairs. But it has no direct impact on complexity analysis.

Typically, R/l and r/l are considered to be relatively small constants, so the total complexity remains of the same order as when sampling is not applied. In fact, the sampling strategy is effective because it reduces the number of data structure operations with large constants, at the cost of increasing the number of checks between point pairs. These checks, however, are fast and remain acceptable in terms of performance. In Section 5.4, we experimentally verified the effectiveness of the sampling strategy.

Note that if we use a balance tree rather than a segment tree, we will not need to cost $\log L$ time for each representative point found in the search operation, but find them as a continuous interval in the balance tree. Then the worst time complexity becomes

$$O\left(\frac{n}{r/l} * \left(\log\left(\frac{n}{r/l}\right) + \frac{R+2r}{l} * \frac{r}{l}\right) + T_1\right). \tag{4}$$

The reduction of log factors will achieve further optimization when the polygons have quite long run length, where the scale of representative points needed to be checked is larger for each search.

In summary, the scheme proposed in this article can convert OPC'ed mask images into GDSII files in input complexity. Given a GDSII OPC'ed layer, the algorithm reports all the spacing violations or width violations including the corner-to-corner case in $O(N * polylog(n))$ time. We will see in the experiments that our scheme is effective in practice, which greatly reduces the actual time consumption.

4.7 Multithread

To benchmark against the multithreading mode of the baseline in our experiments, we also implemented multithreading parallelization. In spacing checks, since the processing of each candidate pair does not interfere with others, these pairs can be directly distributed as tasks to all processes. In width checks, since all inspections occur within each polygon, all polygons can be distributed as tasks to the processes. These strategies do not increase the overall work complexity and achieve good load balance.

5 Experiments

5.1 Setup

We implemented our approach in C++. Our benchmarks comprise optimized patterns from the ICCAD 2013 contest [25], generated by OpenILT [26, 27], and four large masks (active layer, metal layer, poly layer and via layer) of the GCD (which show different pattern distributions on different layers) and two full-chip size masks (CPU0, CPU1) optimized by FuILT [28].

We used the DRC function of the widely used academic tool KLayout [29] as the baseline, since there is no specialized academic tool for MRC. We also conducted experiments with the commercial tool Calibre as reference. The experiments of KLayout and our method were all conducted on a Debian server with an Intel Xeon Silver 4310 CPU. The experiments of Calibre were conducted on a Red Hat virtual machine with an Intel i7-12700H CPU (due to a license problem).

The input images were converted to a GDSII file first with our Format Conversion tool before being imported into our program and KLayout. Figure 10 visualizes a post-OPC mask as well as the corresponding GDSII file with spacing violations reported.

5.2 Overall Comparison

To examine the overall performance of the proposed approach, we compare EasyMRC with the state-of-the-art academic tool KLayout and industrial tool Calibre. Note that Calibre runs on a different platform with only one thread available due to license problem, so the runtime of it can not be directly compared. The distance of violation is set to 50 nm (the following distance units are all in nanometers), slightly greater than the average spacing between polygons to generate a certain amount of violations, thereby more comprehensively testing the program's performance. And we choose the sampling radius as $4l$, where l is the average length of all the polygons in EasyMRC. KLayout and our method use eight threads.

The results of space rule checking and width rule checking are listed in Tables 1 and 2, respectively. #Poly and #edges denote the number of polygons and edges in each case, respectively. #VioP is the number of polygon pairs that have violations (in width check, it denotes the number of polygons that have internal violation), whereas #VioE denotes the number of edge pairs violating the rule generated by different tools. The column *Time* lists the runtime of the tools. All the time units are in milliseconds. *Correct* denotes that our method indicated all the violations without omissions under the following standard.

It should be noted that in mask rule checking, there are often a large number of conflicting edge pairs concentrated in one area. It is unrealistic to ensure that all the violation pairs we generate are exactly the same as the standard answer generated by KLayout or Calibre. (Actually, the specific numbers of violation edge pairs generated by KLayout and Calibre have a large gap.)

In Section 4, we explained why our method may generate fewer violations than KLayout in some circumstances (e.g., only one violation will be recorded within the range of a pair of representative points). In addition, Figure 9 illustrates a possibility that our method generates more violations than KLayout. In KLayout, only the red edge pairs are recorded. However, in our method, they are handled as type (b) violations, and the green violations are handled as type (a) violations. These examples demonstrate why it is hard to align the violation number between different tools.

Therefore, we check the correctness of our programm in two directions. First, for any conflicting point pair in the standard answer generated by KLayout, as long as there is a pair of points in our

Table 1. Runtime Comparisons of Space Rule Checking

Case	#Poly	#Edges	#VioP	Calibre*		KLayout [29]		Ours		
				#VioE	Time	#VioE	Time	#VioE	Time	Correct
Mask1	33	6096	15	1030	77	1129	154.8	346	2.081	√
Mask2	22	5442	17	1180	87	1428	138.6	352	1.984	√
Mask3	32	6676	29	2704	112	3028	218.0	1052	2.645	√
Mask4	33	5206	15	1264	97	1292	109.1	460	2.031	√
Mask5	24	5402	16	1371	83	1430	179.5	603	2.422	√
Mask6	24	5782	12	752	88	872	166.3	253	2.546	√
Mask7	28	4864	10	1392	83	1413	200.2	372	1.814	√
Mask8	28	5660	13	1067	67	1152	169.5	515	1.731	√
Mask9	27	6076	16	728	82	749	223.1	363	1.419	√
Mask10	29	4010	0	0	80	0	21.5	0	1.246	√
Average							211.7		1.992	
Active	700	54109	1690	169271	801	152909	4665	209614	17.11	√
Metal	1481	99374	8904	571795	2052	255165	36331	220531	70.46	√
Poly	1162	45604	5372	154641	724	88209	6601	527390	30.09	√
Via	5411	59205	38052	346581	994	277819	17975	208414	30.51	√
Average							16393		37.04	
CPU0	6858	460652	13872	1387605	5372	1244749	26290	1711950	105.6	√
CPU1	14904	853932	72018	4740751	16346	2163630	199093	1838427	452.1	√
Average							112691		278.9	

*Calibre runtime is only for reference since it runs on a different platform as stated in Section 5.1.
Runtimes are in milliseconds.

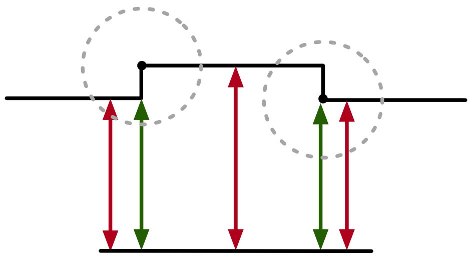

Fig. 9. Different identification of violations.

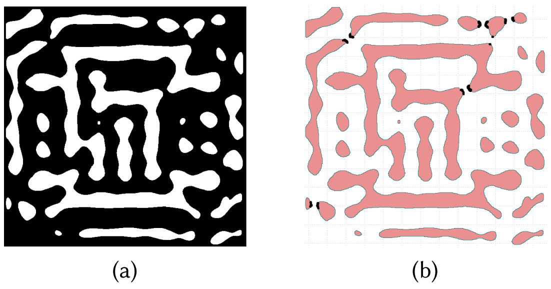

Fig. 10. An OPC'ed mask (a) and the converted GDSII file with violations marked with bold black dots (b).

Table 2. Runtime Comparisons of Width Rule Checking

Case	#VioP	Calibre*		KLayout [29]		Ours		
		#VioE	Time	#VioE	Time	#VioE	Time	Correct
Mask1	33	11643	75	12791	157.5	13493	3.554	√
Mask2	22	11143	65	12270	184.7	11333	3.145	√
Mask3	32	11872	51	12809	247.3	15869	3.296	√
Mask4	33	11327	66	12323	122.8	16303	5.004	√
Mask5	24	10140	68	11161	207.3	15135	4.675	√
Mask6	24	9832	47	10625	250.1	16179	4.697	√
Mask7	28	9846	55	10541	138.2	13387	3.368	√
Mask8	28	15458	65	16397	245.1	9632	4.537	√
Mask9	27	11439	52	11428	214.0	7674	4.242	√
Mask10	29	8256	44	8992	80.8	5112	3.013	√
Average					184.8		3.953	
Active	700	305230	595	283503	1175	250057	12.19	√
Metal	1481	324226	812	240380	4040	105774	27.93	√
Poly	1162	106612	363	85826	415	177746	12.27	√
Via	5411	73924	291	74391	89	42384	7.14	√
Average					1429		14.89	
CPU0	6858	2581226	6177	2398084	7813	2118073	74.1	√
CPU1	14904	2777249	9318	2077771	34307	927754	163.5	√
Average					21060		118.8	

*Calibre runtime is only for reference since it runs on a different platform as stated in Section 5.1.

answer within $2r$ ($2r$ for sampling radius) from it, the result is considered correct (i.e., no omissions). Second, each pair of violations in our results is generated only when the real distance is less than R, which makes each violation pair in our results indeed a violation (i.e., no wrong answers). Since the sampling radius is typically set much smaller than the violation distance, this stipulation ensures that we accurately indicate the locations of all violations for engineers to check.

It can be seen that, for all the cases, EasyMRC achieves dozens of times speedup than Klayout, with an average of 106 times for basic examples, 442 times for large examples, 404 times for full-chip size examples in spacing check and 46 times for basic examples, 96 times for large examples, and 177.2 times for full-chip size examples in width check. Note that the runtime of Calibre cannot be directly compared. It is evident that the proposed sampling-enabled approach is efficient and effective. Larger masks are generally tiled to this scale; even without such processing, EasyMRC's performance will not be inferior to KLayout.

5.3 Runtime Breakdown

We are curious about the time consumption at different stages. Figure 11 shows the distribution of processing time for the six large examples. Each horizontal bar is for one test case, where the green portion is for candidate pair generation, yellow for sampling, and red for sweepline, respectively. The experiment settings are the same as in Section 5.2. The experimental results demonstrate that the sampling strategy reduces the time consumption of the whole process through only a small certain cost of preprocessing, which is precisely the outcome we aimed to achieve.

5.4 Ablation Study

To eliminate the constant influence in algorithm implementation as much as possible and accurately evaluate the effectiveness of the sampling strategy, we conducted a series of additional experiments. The results are listed in Figure 12.

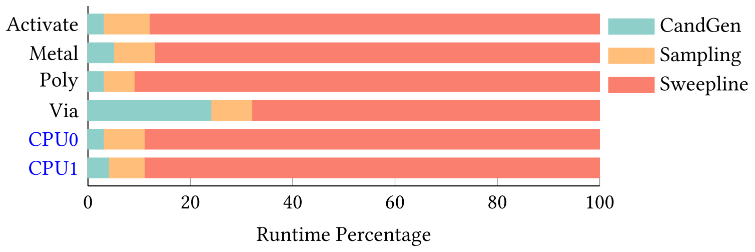

Fig. 11. Runtime breakdown of space check.

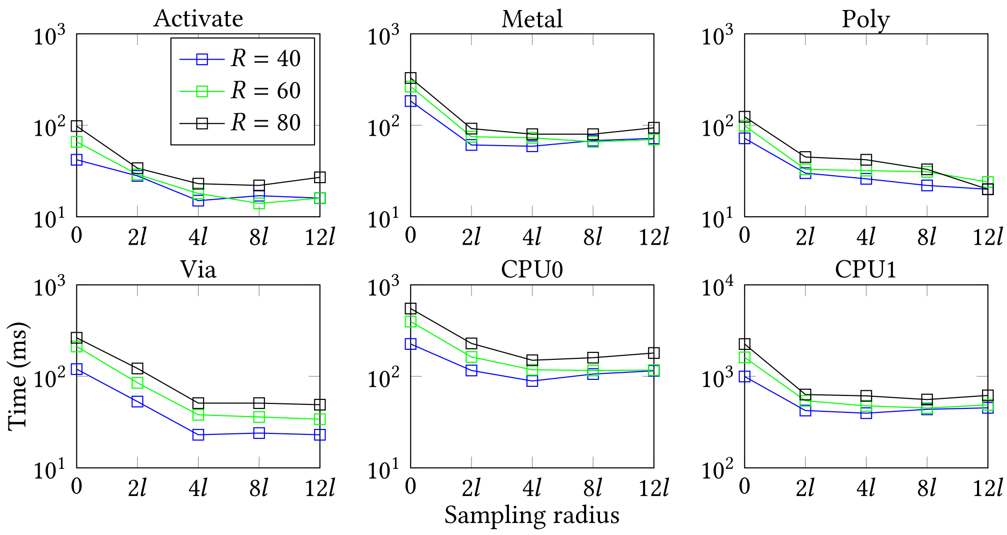

Fig. 12. Time consumption versus sampling radii.

The violation distance is denoted by R, and the sampling radius r is chosen as a multiple of l, where l denotes the average length of all edges. This choice allows us to examine the impact of different sampling radii on time consumption. When r is set to 0, the program no longer performs sampling. Instead, it directly includes all points and edges in the sweepline algorithm. Specifically, it inserts and searches for all points whose Chebyshev distances from the current point are smaller than R in the segment tree, and subsequently verifies whether they violate the rule.

The experimental results show that the optimization effect reaches its maximum when the sampling radius is set to $4l$. As r continues to increase to unreasonable levels, the time consumption does not vary significantly, which aligns with the complexity analysis. While the time consumption for querying in the segment tree indeed has a significant constant factor, this does not imply that larger r are always beneficial. This is because each pair of representative points identified by the segment tree reports only one violation within the range. When r is small, this approach does not affect the accuracy of the report (for reference, R are greater than 40 in our benchmarks, and $r = 4l$ are typically about 10). However, when r is large, it may become difficult for users to identify all violations at once. Therefore, the value of r should remain relatively small compared to R.

The results of ablation experiments show that the sampling strategy can achieve significant speedup without affecting the reported accuracy.

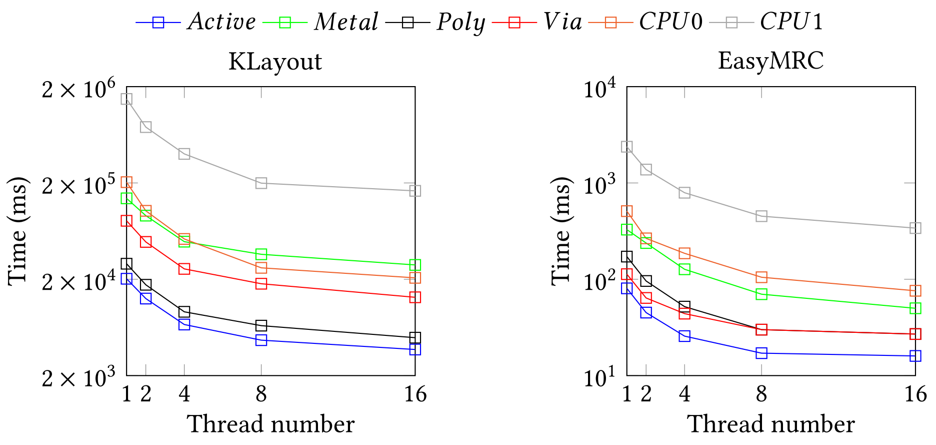

Fig. 13. Time consumption versus thread number.

5.5 Multithread Efficiency

We compared the optimization effects of multithreading parallelization between KLayout and EasyMRC when processing the six large examples. Figure 13 shows the change rate of the total runtime with the number of threads. Other parameter settings are the same as in Section 5.2.

On the four GCD benchmarks, our program has slightly higher parallelism. At 2, 4, 8, and 16 threads, the average total runtime of our program is 1.48, 2.88, 4.68, and 4.88 times that of single-threaded execution, whereas KLayout is 1.51, 2.81, 3.81, and 4.91 times. As the number of threads continues to increase, the speedup ratio gradually becomes flat. As for the two full-chip size examples, the average total runtime of our program is 1.92, 3.77, 6.79, and 7.49 times that of single-threaded execution, whereas KLayout is 1.95, 3.71, 7.48, and 8.99 times. This may indicate that our method spends a larger proportion of time in preprocessing (sampling). Overall, the experiments indicate that the approach of splitting tasks based on candidate pairs can achieve good load balancing.

6 Conclusion

Modern VLSI designs are becoming more and more complex, and technology nodes continue to shrink. In the meantime, the photolithography process is getting more sophisticated, and photomask patterns are getting finer. There is no doubt that mask rule checking is essential to ensure the validity of the photomask. Unlike the layout data in DRC, mask data are more irregular with short edges in zig-zag shapes. Therefore, this article proposed an efficient mask rule checking approach tailored to such characteristics. A representative edge sampling scheme was utilized to sample a subset of edges and points of each polygon, which best captures its contour, while greatly reducing the number of edges involved in actual checking. Experimental results were conducted to demonstrate the effectiveness of such an approach: compared with the state-of-the-art academic tool, significant speedup was achieved.

References

[1] Wakahiko Sakata, Kiyoshi Yamasaki, Shogo Narukawa, and Naoya Hayashi. 2005. Mask rule check for inspection of leading-edge photomask. In *Proceedings of the BACUS Symposium on Photomask Technology*, Vol. 5992. 1283–1291.

[2] Martin C. Keck, Wolfram Ziegler, Roman Liebe, Torsten Franke, Gerd Ballhorn, Matthias Koefferlein, and Joerg Thiele. 2001. Mask manufacturing rule check: How to save money in your mask shop. In *Proceedings of the 20th Annual BACUS Symposium on Photomask Technology*, Vol. 4186. 114–118.

[3] Bill Moore, Tanya Do, and Ray E. Morgan. 2006. Advanced non-disruptive manufacturing rule checks (MRC). In *Photomask Technology 2006*. Vol. 6349. SPIE, 317–326.

[4] Mark Mason, Christopher J. Progler, Patrick Martin, Young-Mog Ham, Brian Dillon, Robert Pack, Mitch Heins, John Gookassian, John Garcia, and Victor Boksha. 2005. Mask design rules (45 nm): Time for standardization. In *Proceedings of the 25th Annual BACUS Symposium on Photomask Technology*, Vol. 5992. 98–107.

[5] Kokoro Kato, Kuninori Nishizawa, Tadao Inoue, Koki Kuriyama, Toshio Suzuki, Shogo Narukawa, and Naoya Hayashi. 2006. Advanced mask rule check (MRC) tool. In *Photomask and Next-Generation Lithography Mask Technology XIII*. Vol. 6283. SPIE, 153–163.

[6] Kokoro Kato, Yoshiyuki Taniguchi, Kuninori Nishizawa, Masakazu Endo, Tadao Inoue, Ryouji Hagiwara, and Anto Yasaka. 2007. Mask rule check using priority information of mask patterns. In *Photomask Technology 2007*. Vol. 6730. SPIE, 1420–1429.

[7] Peter Buck, Richard Gladhill, and Joseph Straub. 2007. Mask manufacturing rules checking (MRC) as a DFM strategy. In *Design for Manufacturability through Design-Process Integration*. Vol. 6521. SPIE, 575–579.

[8] R. Gladhill, D. Aguilar, P. D. Buck, D. Dawkins, S. Nolke, J. Riddick, and J. A. Straub. 2005. Advanced manufacturing rules check (MRC) for fully-automated assessment of complex reticle designs. In *Proceedings of the 25th Annual BACUS Symposium on Photomask Technology*, Vol. 5992. 69–77.

[9] Jon Louis Bentley and Derick Wood. 1980. An optimal worst case algorithm for reporting intersections of rectangles. *IEEE Transactions on Computers* 29, 07 (1980), 571–577.

[10] Ulrich Lauther. 1981. An $O(N \log N)$ algorithm for Boolean mask operations. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC '81)*. 555–562.

[11] Raphael A. Finkel and Jon Louis Bentley. 1974. Quad trees a data structure for retrieval on composite keys. *Acta Informatica* 4, 1 (1974), 1–9.

[12] Jon Louis Bentley. 1975. Multidimensional binary search trees used for associative searching. *Communications of the ACM* 18, 9 (1975), 509–517.

[13] Antonin Guttman. 1984. R-trees: A dynamic index structure for spatial searching. In *Proceedings of the ACM Conference on Management of Data (SIGMOD '84)*. 47–57.

[14] John K. Ousterhout. 1984. Corner stitching: A data-structuring technique for VLSI layout tools. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 3, 1 (1984), 87–100.

[15] Jon Louis Bentley and Jerome H. Friedman. 1979. Data structures for range searching. *ACM Computing Surveys* 11, 4 (1979), 397–409.

[16] George E. Bier and Andrew R. Pleszkun. 1985. An algorithm for design rule checking on a multiprocessor. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC '85)*. 299–304.

[17] F. Gregoretti and Z. Segall. 1984. Analysis and evaluation of VLSI design rule checking implementation in a multiprocessor. 7–14.

[18] Erik C. Carlson and Rob A. Rutenbar. 1988. Mask verification on the connection machine. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC '88)*. 134–140.

[19] Kai-Ti Hsu, Subarna Sinha, Yu-Chuan Pi, Charles Chiang, and Tsung-Yi Ho. 2011. A distributed algorithm for layout geometry operations. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC '11)*. 182–187.

[20] Zhuolun He, Yuzhe Ma, and Bei Yu. 2022. X-Check: GPU-accelerated design rule checking via parallel sweepline algorithms. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '22)*. 1–9.

[21] Zhuolun He, Yihang Zuo, Jiaxi Jiang, Haisheng Zheng, Yuzhe Ma, and Bei Yu. 2023. OpenDRC: An efficient open-source design rule checking engine with hierarchical GPU acceleration. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC '23)*. 1–6.

[22] Zhuolun He and Bei Yu. 2023. Heterogenous acceleration for design rule checking. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '23)*. 1–8.

[23] Yu-Hsuan Su, Yu-Chen Huang, Liang-Chun Tsai, Yao-Wen Chang, and Shayak Banerjee. 2016. Fast lithographic mask optimization considering process variation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 35, 8 (2016), 1345–1357.

[24] Ingo Bork, Alexander Tritchkov, Shumay Shang, Evgueni Levine, and Mary Zuo. 2020. MRC for curvilinear mask shapes. In *Photomask Technology 2020*, Vol. 11518. SPIE, 115180R.

[25] Shayak Banerjee, Zhuo Li, and Sani R. Nassif. 2013. ICCAD-2013 CAD contest in mask optimization and benchmark suite. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD '23)*. 271–274.

[26] Su Zheng, Bei Yu, and Martin Wong. 2023. OpenILT: An open source inverse lithography technique framework. In *Proceedings of the IEEE International Conference on ASIC (ASICON '23)*.

[27] Su Zheng, Yuzhe Ma, Binwu Zhu, Guojin Chen, Wenqian Zhao, Shuo Yin, Ziyang Yu, and Bei Yu. 2023. OpenILT: An Open-Source Platform for Inverse Lithography Technique Research. Retrieved March 19, 2025 from https://github.com/OpenOPC/OpenILT/

[28] Shuo Yin, Wenqian Zhao, Li Xie, Hong Chen, Yuzhe Ma, Tsung-Yi Ho, and Bei Yu. 2024. FuILT: Full chip ILT system with boundary healing. In *Proceedings of the ACM International Symposium on Physical Design (ISPD '24)*. 13–20.

[29] KLayout. n.d. Home Page. Retrieved March 19, 2025 from https://klayout.de/

Received 4 October 2024; revised 12 January 2025; accepted 25 February 2025