ForgePattern: Flexible Layout Pattern Generation through Reinforcement Learning

YU FENG, Fudan University, Shanghai, China
ZIYANG YU, The Chinese University of Hong Kong, Hong Kong, Hong Kong
HAOYU WANG, Fudan University, Shanghai, China
YIMING XU, Fudan University, Shanghai, China
JIAQI LIU, Zhangjiang Laboratory, Shanghai, China
BEI YU, The Chinese University of Hong Kong, Hong Kong, China
SHISHENG XIONG, Fudan University, Shanghai, China and Zhangjiang Laboratory, Shanghai, China

Layout pattern generation provides dataset support to numerous Design for Manufacturability (DFM) studies, with different research objectives requiring distinct styles of layout patterns. However, existing layout pattern generation methods only support random generation or limited conditional generation based on image-text pairs annotated for specific tasks, which fails to meet the diverse requirements of downstream tasks. To address this limitation, we propose ForgePattern, a flexible layout pattern generation framework that accommodates diverse task-specific requirements through adaptive selection among customizable reward functions. ForgePattern leverages a customized diffusion model as the generative backbone and incorporates reinforcement learning to directly optimize the diffusion model. By maximizing the expected reward of the reinforcement learning objective, ForgePattern aligns the generated layout patterns with specific requirements without requiring additional data collection or human annotation. Experimental results on challenging layout pattern generation tasks demonstrate that ForgePattern can effectively generate high-quality layout patterns while meeting diverse customization requirements.

CCS Concepts: • **Hardware → Physical design (EDA);**

Additional Key Words and Phrases: Pattern generation, design for manufacturability, diffusion models, reinforcement learning, process window

ACM Reference Format:
Yu Feng, Ziyang Yu, Haoyu Wang, Yiming Xu, Jiaqi Liu, Bei Yu, and Shisheng Xiong. 2026. ForgePattern: Flexible Layout Pattern Generation through Reinforcement Learning. *ACM Trans. Des. Autom. Electron. Syst.* 31, 5, Article 113 (April 2026), 22 pages. https://doi.org/10.1145/3799991

Authors' Contact Information: Yu Feng, Fudan University, Shanghai, China; e-mail: yufeng23@m.fudan.edu.cn; Ziyang Yu, The Chinese University of Hong Kong, Hong Kong, Hong Kong, Hong Kong; e-mail: yuziyang96@gmail.com; Haoyu Wang, Fudan University, Shanghai, Shanghai, China; e-mail: 22110720056@m.fudan.edu.cn; Yiming Xu, Fudan University, Shanghai, Shanghai, China; e-mail: yimingxu23@m.fudan.edu.cn; Jiaqi Liu, Zhangjiang Laboratory, Shanghai, Shanghai, China; e-mail: liujq@zjlab.ac.cn; Bei Yu, The Chinese University of Hong Kong, Hong Kong, Hong Kong, China; e-mail: byu@cse.cuhk.edu.hk; Shisheng Xiong (corresponding author), Fudan University, Shanghai, Shanghai, China and Zhangjiang Laboratory, Shanghai, Shanghai, China; e-mail: sxiong@fudan.edu.cn.



This work is licensed under a Creative Commons Attribution 4.0 International License.

© 2026 Copyright held by the owner/author(s).
ACM 1084-4309/2026/04-ART113
https://doi.org/10.1145/3799991

1 Introduction

Very-large-scale integration (**VLSI**) layout pattern libraries serve as a foundational resource for various **design for manufacturability** (**DFM**) studies, including design rule refinement, **optical proximity correction** (**OPC**) recipe optimization [1–4], lithography simulation [5–7], and layout hotspot detection [8, 9]. Each application requires patterns with distinct geometric and statistical properties. The rise of machine learning-based lithography design applications further magnifies the demand for diverse, high-quality patterns, yet constructing dedicated, extensive, large-scale libraries for every objective is highly time-consuming due to the long logic-to-chip design cycle. In response, researchers have shifted their focus toward synthesizing diverse and task-specific VLSI layout patterns to enable more efficient and targeted DFM studies.

Previous studies [10, 11] began with rule-based approaches for synthesizing novel layout patterns. Predefined unit patterns were manually selected, augmented through simple manipulations such as rotation and flipping, and then combined to generate additional layouts. While simple, these methods often lacked sufficient diversity and scale. To address this, deep learning-based approaches were developed. **Convolutional autoencoders** (**CAEs**) [12] generated new patterns by perturbing latent representations of existing layouts but were limited to one-dimensional patterns. **Variational CAEs** (**VCAEs**) [13] extended this to two-dimensional patterns and incorporated conditional GANs to reduce DRC violations. LayoutTransformer [14] framed pattern generation as a sequence modeling task. Recently, diffusion models [15–17] have emerged as a powerful generative framework. These models gradually add noise to samples until reaching a known distribution and iteratively denoise to reconstruct patterns. DiffPattern [18] applies a discrete diffusion model to layout generation, interpreting the task as denoising over discrete pixel states and introducing a nonlinear legalization method to ensure high-quality, DRC-compliant patterns.

Although these methods significantly advanced pattern diversity and DRC compliance, they predominantly generate patterns randomly. Consequently, they tend to produce numerous irrelevant patterns, leading to low sampling efficiency when task-specific layout generation is required. This gap highlights the need for a flexible pattern synthesis framework that can efficiently generate large numbers of design-style-consistent layout patterns while adapting its generation preferences to different objectives. Among the various objectives, two key practical needs stand out.

(1) Flexibly adjusting the complexity of generated layouts to explore both simpler and more complex patterns, thereby enlarging the design space and increasing the diversity of the synthesized layout library. Such more diverse patterns provide comprehensive data for OPC recipe refinement, helping to identify patterns that are difficult to correct and guiding subsequent design-rule adjustment to prevent such patterns in future designs.

(2) Discovering patterns with small process windows. The process window is a critical manufacturability metric, quantifying a layout pattern's tolerance to focus and exposure variations during lithography. Patterns with small process windows are more prone to patterning failures, making their early identification essential for refining design rules and optimizing OPC recipes [19–23]. Yet such weak patterns are subtle, non-obvious, and sparse in existing datasets, which makes discovering a diverse set of them particularly challenging.

To tackle these challenges, we propose ForgePattern, a flexible framework that dynamically adjusts generation preferences to meet diverse optimization objectives. By formulating the reverse diffusion process as a sequential decision-making problem, ForgePattern treats each denoising step as an action selected by a **reinforcement learning** (**RL**) policy, with the policy implemented by the U-Net backbone [24]. For each denoising trajectory, the resulting layout pattern is evaluated by a task-specific reward function, producing a *scalar* reward signal. The pretrained diffusion model is then fine-tuned using policy-gradient updates to maximize the expected reward over trajectories,

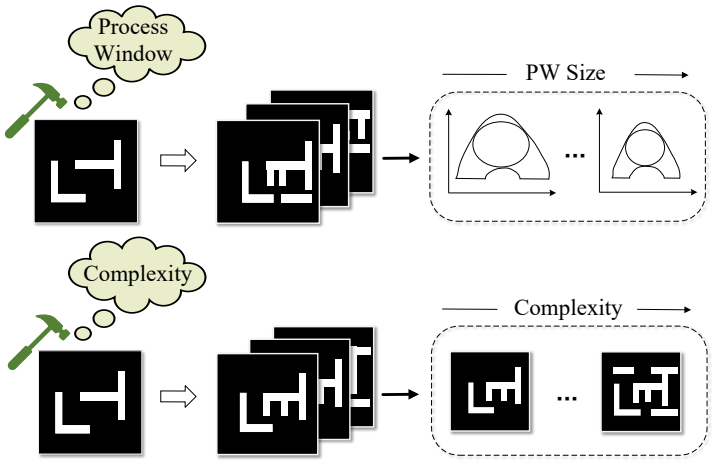


Fig. 1. Customizing layout pattern generation via flexible reward functions. (white regions represent the metal shapes).

which updates the U-Net parameters and steers the learned reverse-process dynamics toward high-reward layout patterns.

In contrast to unconditional layout diffusion methods such as DiffPattern [18], which generate patterns primarily by random sampling from the learned distribution and thus relying on an inefficient generate-and-filter paradigm with low hit rates for rare or extreme patterns, ForgePattern directly guides generation toward user-specified DFM objectives through reward optimization. Moreover, unlike text-to-image conditional diffusion approaches [25, 26] that rely heavily on paired prompt–layout datasets and struggle to express quantitative manufacturability objectives (e.g., process-window size) via natural-language prompts, ForgePattern requires neither paired annotations nor textual prompts. As shown in Figure 1, RL is applied during a fine-tuning stage under different reward functions, enabling the model to generate batches of layout patterns that satisfy the desired specifications during inference.

Previously, identifying patterns with small process windows relied entirely on OPC engineers' experience and verification through simulation. With ForgePattern, such weak patterns can be generated efficiently and at scale, providing a diverse set of examples for refining OPC recipes and adjusting design rules. Similarly, the framework can flexibly adjust layout complexity to explore both simple and highly complex patterns. Importantly, ForgePattern is not limited to these two objectives. By specifying different reward functions, ForgePattern can flexibly adjust various pattern attributes independently, supporting a wide range of user-defined generation preferences.

In summary, our main contributions can be summarized as follows:

(1) To the best of our knowledge, ForgePattern is the first approach that dynamically tunes the layout pattern generation process to meet distinct, user-specified optimization goals.
(2) We develop a novel layout pattern generation framework that optimizes the diffusion model via RL for customizable generation, and accelerates generation through our temporal cache reuse mechanism.
(3) Our framework offers a promising approach for constructing layout libraries that contain certain rare or subtle characteristics at low cost.
(4) Compared to previous text-to-image methods for layout pattern generation, our method eliminates the need for additional human annotations, overcoming the limitations of fixed training datasets.

(5) We extensively evaluate our methodology on benchmark datasets, showing that ForgePattern can achieve **state-of-the-art** (**SOTA**) performance.

The remainder of this article is organized as follows. Section 2 gives some preliminaries of this work. Subsequently, we delve into the details of the ForgePattern framework in Section 3. Experimental results and analysis are presented in Section 4. Finally, Section 5 concludes this work.

2 Preliminaries

2.1 Diffusion Models

Denoising diffusion probabilistic models (**DDPMs**) [27] are a class of generative models that learn to model complex data distributions through a parameterized Markovian diffusion process. The core idea is to transform data into Gaussian noise via a forward diffusion process, and then learn a reverse generative process that gradually removes noise to recover data samples.

The forward process is a fixed Markov chain that iteratively adds Gaussian noise to the input data $\mathbf{x}_0$ over T steps, producing a sequence of latent variables $\mathbf{x}_1, \dots, \mathbf{x}_T$. Each step follows:

$$q(\mathbf{x}_t \mid \mathbf{x}_{t-1}) = \mathcal{N}\left(\mathbf{x}_t; \sqrt{1-\beta_t}\mathbf{x}_{t-1}, \beta_t \mathbf{I}\right), \tag{1}$$

where $\{\beta_t \in (0,1)\}_{t=1}^{T}$ is a predefined variance schedule that controls the rate of noise injection. As $t \to T$, the marginal $q(\mathbf{x}_T)$ approaches an isotropic Gaussian distribution.

The reverse process is defined as a parameterized Markov chain intended to invert the forward process:

$$p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t) = \mathcal{N}\left(\mathbf{x}_{t-1}; \mu_\theta(\mathbf{x}_t, t), \sigma_t^2 I\right), \tag{2}$$

where a neural network with parameters θ predicts the mean of the reverse distribution. σ_t^2 is a non-trainable parameter equivalent to $\frac{1-\bar{\alpha}_{t-1}}{1-\bar{\alpha}_t}$ and $\bar{\alpha}_t = \prod_{i=1}^{t}(1-\beta_i)$. The training of DDPMs aims to maximize the log-likelihood function by optimizing the variational bound on the negative log-likelihood $\mathbb{E}_q[-\log p_\theta(x_0)]$, which is equivalent to optimizing:

$$\mathcal{L}_{\text{VLB}} = \mathbb{E}_q\left[\sum_{t=1}^{T} \text{KL}(q(\mathbf{x}_{t-1} \mid \mathbf{x}_t, \mathbf{x}_0) \parallel p_\theta(\mathbf{x}_{t-1} \mid \mathbf{x}_t))\right], \tag{3}$$

where $\text{KL}(q \parallel p)$ is the **Kullback–Leibler** (**KL**) divergence between distributions q and p. In practice, many implementations simplify training by predicting the added noise ϵ instead of directly modeling the data distribution. The simplified objective is as

$$\mathcal{L}_{\text{simple}} = \mathbb{E}_{\mathbf{x}_0,\epsilon,t}\left[\left\|\epsilon - \epsilon_\theta\left(\sqrt{\bar{\alpha}_t}\mathbf{x}_0 + \sqrt{1-\bar{\alpha}_t}\epsilon, t\right)\right\|^2\right]. \tag{4}$$

At inference time, the model generates samples by drawing $\mathbf{x}_T \sim \mathcal{N}(0, \mathbf{I})$ and iteratively applying the learned reverse process to obtain a new sample $\mathbf{x}_0$ following the original data distribution.

2.2 Markov Decision Processes (MDPs) and Reinforcement Learning

MDP provides a mathematical framework for modeling sequential decision-making problems. Formally, an MDP is defined by a tuple (S, A, ρ_0, P, R), where S is the state space, A is the action space, ρ_0 is the initial state distribution, P is the state transition kernel, and R is the reward function. At each timestep t, an agent observes a state $s_t \in S$, takes an action $a_t \in A$, receives a reward $R(s_t, a_t)$, and transitions to a new state $s_{t+1} \sim P(s_{t+1}|s_t, a_t)$ according to the transition kernel. The agent's behavior is defined by a policy $\pi(a|s)$ that specifies the probability of taking action a in state s.

RL provides methods for learning optimal policies in MDPs through interaction with an environment. As the agent acts in the environment, it generates trajectories $\tau = (s_0, a_0, s_1, a_1, \dots, s_T, a_T)$

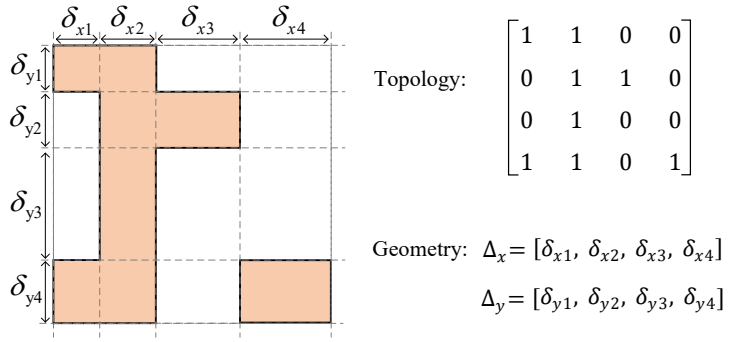


Fig. 2. Squish pattern representation.

consisting of sequences of states and actions. The objective in RL is to find a policy π that maximizes the expected cumulative reward over trajectories:

$$J_{RL}(\pi) = \mathbb{E}_{\tau \sim p(\tau|\pi)} \left[\sum_{t=0}^{T} R(s_t, a_t) \right]. \tag{5}$$

This optimization is typically performed using policy gradient methods that update the policy parameters in the direction that increases expected returns.

The reverse process in diffusion models can be naturally viewed as a sequential decision-making problem that can be solved by RL, where each denoising step corresponds to an action taken by the model to gradually transform noise into meaningful data.

2.3 Squish Pattern Representation

The squish pattern representation [28] provides a lossless and efficient structured way to encode layout patterns into a topology matrix and geometric information. For a given layout pattern, a set of scan lines are placed along polygon edges to partition the layout into a grid structure. The distances between adjacent scan lines are stored in Δx and Δy vectors, while the topology matrix uses binary values to indicate the presence of shapes (1 for shape, 0 for empty space), as illustrated in Figure 2. This representation provides a consistent and clear way to extract scan lines from a layout, which can be used to compute the diversity of the generated pattern library.

2.4 Problem Formulation

The controllability of generated patterns is practical for DFM studies. To quantitatively evaluate the controllability, we introduce several key metrics. First, following [12], we define the pattern complexity as (c_x, c_y), where c_x and c_y represent the numbers of scan lines minus one along the x-axis and y-axis, respectively.

Definition 1 (Pattern Legality). A layout pattern is considered legal if it satisfies all **design rule checks** (**DRC**s), including minimum spacing between polygons, minimum width requirements, and area constraints as illustrated in Figure 3.

Definition 2 (Process Window). The process window size measures the manufacturability robustness of a layout pattern under focus and exposure variations. Figure 4 shows a process window plot with defocus (ΔF) on the horizontal axis and dose offset (ΔE) on the vertical axis. We perform lithography simulations on a discrete **focus–exposure** (**F–E**) matrix spanning the manufacturing range $\Delta F \in [-90\,\text{nm}, 90\,\text{nm}]$ and $\Delta E \in [-10\%, 10\%]$. At each grid point $(\Delta F, \Delta E)$, the printed

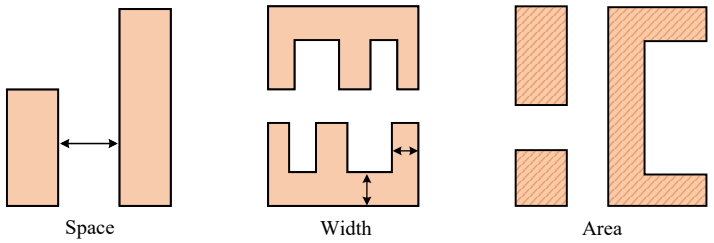


Fig. 3. Layout geometry concepts for design rules.

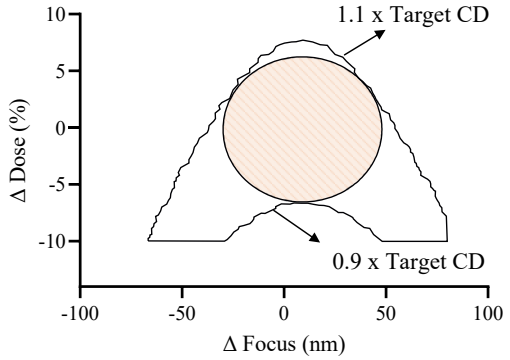


Fig. 4. Illustration of process window. The X-axis shows defocus ΔF (nm), and the Y-axis shows dose offset ΔDose (%). The boundary corresponds to 0.9× and 1.1× target CD. The shaded ellipse represents the largest inscribed ellipse, used to quantify the size of the process window.

critical dimension (**CD**) is evaluated and the pattern is considered in-spec if the CD stays within the allowed tolerance band (e.g., 0.9×–1.1× the target CD). The set of in-spec F–E points forms the feasible process window region, whose boundary corresponds to extreme process corners. The area of the largest inscribed ellipse within this feasible region is used as a quantitative measure of the size of the process window, denoted as pw_size.

Definition 3 (Pattern Diversity). The diversity metric H of a pattern library is defined as the Shannon Entropy of the pattern complexity distribution:

$$H = -\sum_{i}\sum_{j} P\left(c_{xi}, c_{yj}\right) \log P\left(c_{xi}, c_{yj}\right), \tag{6}$$

where $P(c_{xi}, c_{yj})$ denotes the probability of sampling a pattern with complexity (c_{xi}, c_{yj}) from the library.

Based on these metrics, we formulate the pattern generation problem as follows:

Problem 1. *Given a set of existing patterns, design rules, and target specifications, the goal is to synthesize a legal pattern library that meets specified DFM requirements, such as layout patterns with a narrow process window size.*

3 Algorithm

3.1 Overview

Developing a flexible pattern generation framework that can be retargeted to varied DFM objectives is both crucial and notoriously challenging. As far as we know, ForgePattern is the first method to

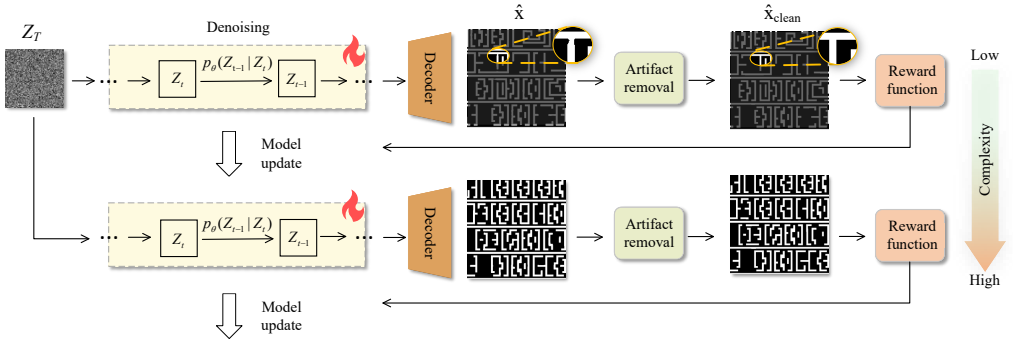


Fig. 5. An illustration of the RL fine-tuning stage in our ForgePattern framework. Except for the denoising module, all other components remain frozen during RL fine-tuning. Rewards are obtained by sampling layout patterns online from the generator and evaluating them with the reward function. Policy-gradient methods are then used to iteratively update the diffusion model weights based on these reward signals, steering the model to satisfy specific DFM requirements.

dynamically adjust the generation process in response to distinct optimization goals specified by the user. In this section, we detail the ForgePattern framework and present its fine-tuning workflow, as illustrated in Figure 5. In the first step, we randomly sample a noise input Z_T, and iteratively denoise it with the learned transition distribution $p_\theta(z_{t-1}|z_t)$ until obtaining the output Z_0. Then the latent Z_0 is decoded into an initial layout pattern and subsequently refined by our artifact removal module to eliminate defects such as jagged edges and false contours. To align the generated patterns with DFM requirements, we reformulate the denoising phase as an MDP, where each denoising step corresponds to a state transition governed by a policy. We subsequently optimize this policy with a policy gradient RL that maximizes task-specific reward functions.

3.2 Layout Pattern Generation Backbone

Latent Diffusion with Temporal Cache Reuse. Diffusion models offer outstanding generative performance in image-synthesis tasks. However, their substantial computational cost remains a major bottleneck for practical applications, particularly when dealing with high-resolution layout patterns with sophisticated pattern geometries. To address this, a common strategy is to execute the generative process in a learned latent space. **Latent Diffusion models (LDMs)** [29], for instance, reduce memory and computation requirements by conducting the denoising process in a lower-resolution latent space, while still maintaining generation quality. Our backbone adopts the similar approach as well: given a layout pattern $\boldsymbol{x} \in \mathbb{R}^{H\times W}$, we first encode it into a compact latent representation $\boldsymbol{z} = \mathcal{E}(\boldsymbol{x}) \in \mathbb{R}^{C_z\times h\times w}$, where $h \ll H$, $w \ll W$, and C_z denotes the number of latent channels. The diffusion process is then applied to $\boldsymbol{z}$ instead of $\boldsymbol{x}$, following the same noise schedule and denoising formulation introduced in Section 2.1. After obtaining the final latent $\boldsymbol{z}_0$, a decoder $\mathcal{D}$ maps it back to the pixel space: $\hat{x} = \mathcal{D}(\boldsymbol{z}_0)$.

Beyond the computational burden of high-resolution input, generation speed is also hampered by the inherently sequential nature of the denoising procedure, which resists parallelization. Each denoising step normally requires a full pass through the entire U-Net [24], resulting in significant cumulative computation. U-Net consists of a symmetric encoder-decoder structure with skip connections between downsampling blocks D_i and upsampling blocks U_i. At each upsampling block U_i, the model fuses two feature sources: low-level features from the skip-connected encoder block D_i, and high-level features from the preceding decoder block U_{i+1}.

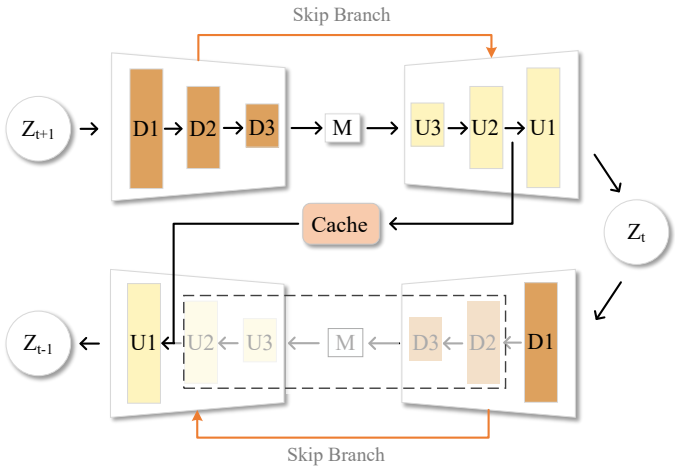


Fig. 6. Illustration of the temporal reuse mechanism. At step $t-1$, the feature z_{t-1} is generated by reusing the cached features from step t. As a result, blocks $D2, D3, U2, U3$ are skipped, enabling more efficient inference.

During the denoising process of layout pattern generation, we observe strong similarity in high-level features across adjacent timesteps. This temporal redundancy suggests that it is unnecessary to recompute these features at every step. Inspired by this observation, our backbone incorporates a temporal reuse mechanism during inference, similar to DeepCache [30], which selectively caches high-level features at key timesteps. Specifically, we define the set of *key timesteps* as

$$\mathcal{K} = \left\{ iN \mid i \in \mathbb{Z}, 0 \le i < \left\lfloor \frac{T}{N} \right\rfloor \right\}, \tag{7}$$

where N denotes the temporal interval for feature caching, and $\lfloor \cdot \rfloor$ represents the floor operator, which maps a real number to the largest integer less than or equal to it. To avoid error accumulation caused by excessive reuse of cached features, N should not be set too large. At each $t \in \mathcal{K}$, we perform a full U-Net inference and cache the feature maps from the penultimate upsampling block U_2 as $F^t_{\text{cache}} = U^t_2(\cdot)$. The cached features are then reused over the subsequent $N-1$ steps to replace the original decoder outputs $\{U^{t-n}_2(\cdot) \mid n = 1, \dots, N-1\}$. For these $N-1$ steps, we avoid full forward computation by performing only a partial forward pass to obtain the low-level features $\{D^{t-n}_1(\cdot) \mid n = 1, \dots, N-1\}$, and substitute the main branch compute with a retrieval operation from the cached high-level features. This allows us to reconstruct the upsampling block input efficiently. For instance, when $N = 2$, the temporal reuse mechanism is illustrated in Figure 6. At timestep $t-1$, the input to U^{t-1}_2 can be formulated as

$$\tilde{f}_t = \phi\left(D^{t-1}_1(\cdot), F^t_{\text{cache}}\right), \tag{8}$$

where ϕ denotes a feature fusion operator such as concatenation. This temporal reuse introduces only a modest memory overhead of one additional U_2-resolution feature map per sample and is reset for each new pattern generation to avoid cross-sample interference. Generating one layout patch requires T denoising steps, where each step corresponds to one forward pass through the U-Net backbone in the latent space. Accordingly, the denoising stage incurs a computational cost of $T \cdot C_{\text{U-Net}}(h, w)$, where $C_{\text{U-Net}}(h, w)$ denotes the cost of a single U-Net forward pass on a latent tensor of spatial resolution $h \times w$. With a fixed backbone architecture, $C_{\text{U-Net}}(h, w)$ increases roughly linearly with the latent spatial area. This is because the runtime is dominated by convolutional blocks (linear in $h \times w$), and although attention layers may introduce slight super-linear scaling,

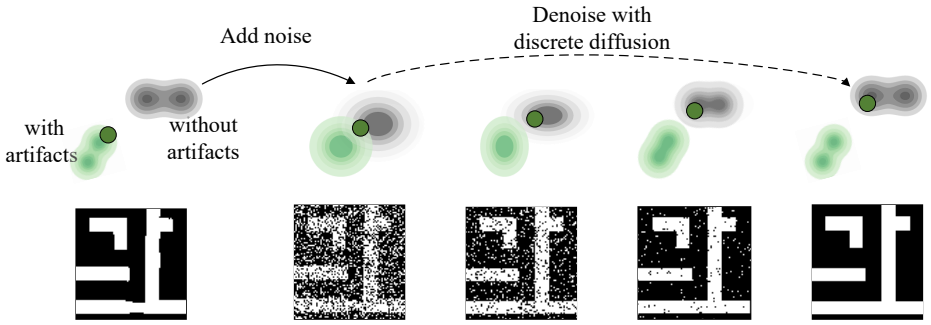


Fig. 7. Removing artifacts with the discrete diffusion. The bottom row shows representative layout snapshots along the post-processing trajectory, from the LDM output $\hat{x}$ (may contain artifacts) to the cleaned result $\hat{x}_{\text{clean}}$ after noise addition and discrete-diffusion denoising. The top contours schematically indicate the distributions of artifact-prone (green) and artifact-free (gray) layouts. The green dot marks the current sample and corresponds to the snapshot shown directly below it. Intuitively, noise injection moves the sample toward the overlap between the two distributions, and the discrete diffusion model then denoises it toward the artifact-free distribution.

their impact remains secondary at the latent resolutions considered in this work. The encoder and decoder introduce additional one-time costs $C_{\mathcal{E}}(H,W)$ and $C_{\mathcal{D}}(H,W)$ per patch, so the total per-patch inference cost is $C_{\mathcal{E}}(H,W) \;+\; T \cdot C_{\text{U-Net}}(h,w) \;+\; C_{\mathcal{D}}(H,W)$. Temporal cache reuse does not change the asymptotic scaling in T and hw, but reduces the constant factor of per-step computation in practice.

Artifact Removal Module. Although latent diffusion models are highly efficient for layout pattern generation, they are inherently designed for continuous state spaces. The continuous outputs of the decoder are misaligned with the discrete nature of layout patterns. Applying simple binary truncation upon generating continuous examples often introduces artifacts such as jagged edges and false contours. One potential solution would be to abandon LDMs and directly employ discrete diffusion models, which are specifically optimized for discrete data spaces. However, these models typically operate in pixel space, thus sacrificing the computational efficiency advantages offered by latent diffusion. To address this dilemma, we propose an artifact removal module inspired by the SDEdit framework [31]. Instead of iteratively adding noise to the layout pattern $\hat{x}$ over the full T steps as in standard diffusion forward process, we truncate the forward process at $0.4T$ steps, leading to an intermediate state:

$$\hat{\boldsymbol{x}}_{0.4T} = \sqrt{\bar{\alpha}_{0.4T}}\hat{\boldsymbol{x}} + \sqrt{1-\bar{\alpha}_{0.4T}}\epsilon, \tag{9}$$

where $\bar{\alpha}_{0.4T} = \prod_{t=1}^{\lfloor 0.4T \rfloor}(1-\beta_t)$, and $\epsilon \sim \mathcal{N}(0, I)$. Then, a discrete diffusion model such as D3PM[32] or Bit Diffusion[33] is used to take over the reverse process from this intermediate state until getting an artifact-free layout pattern $\hat{\boldsymbol{x}}_{\text{clean}}$, as shown in Figure 7. Since the discrete diffusion model is used for post-processing with fixed weights, only the latent diffusion model requires RL fine-tuning, substantially reducing memory usage and training cost. The artifact removal module is a pixel-space refinement stage applied after decoding, implemented via a discrete diffusion process. The forward noising to $\hat{\boldsymbol{x}}_{0.4T}$ is computed in closed form and involves only a single Gaussian perturbation, incurring negligible overhead compared to iterative diffusion sampling. The additional cost comes from running a discrete diffusion sampler for $T_{\text{clean}} = \lfloor 0.4T \rfloor$ reverse steps. Therefore, artifact removal adds approximately $0.4T$ denoising iterations per patch, while the overall inference cost remains linear in the total number of diffusion steps.

3.3 Controllable Layout Pattern Customization via RL

Similar to the setting in prior work [34][35], we model the denoising process as a multi-step MDP. Unlike those studies that focus on conditional generation, we target controllable layout pattern generation without external conditioning.

MDP Formulation. Specifically, the iterative denoising procedure is formulated as the following MDP: The **state** at timestep t is given by

$$s_t \triangleq (t, \boldsymbol{z}_t), \tag{10}$$

where $\boldsymbol{z}_t \in \mathbb{R}^{C_z \times h \times w}$ is the latent variable at step t, and $t \in \{0, \dots, T\}$ indicates the current timestep. Therefore, the state space is hybrid, consisting of a continuous component $\boldsymbol{z}_t$ and a discrete component t. The **action** is defined as

$$a_t \triangleq \boldsymbol{z}_{t-1}, \tag{11}$$

which corresponds to the predicted latent variable for the preceding step. The action space is therefore continuous, with $a_t \in \mathbb{R}^{C_z \times h \times w}$. The **policy** π_θ is parameterized by θ and aligns with the reverse diffusion distribution:

$$\pi_\theta(a_t \mid s_t) \triangleq p_\theta(\boldsymbol{z}_{t-1} \mid \boldsymbol{z}_t). \tag{12}$$

When using a standard sampler with $p_\theta(\boldsymbol{z}_{t-1} \mid \boldsymbol{z}_t)$, the policy follows an isotropic Gaussian distribution, enabling the exact evaluation of log-likelihoods and their gradients with respect to the diffusion model parameters. The **initial state** is drawn from a distribution given by

$$p_\theta(s_0) \triangleq (\delta_T, \mathcal{N}(0, I)), \tag{13}$$

where δ_T is the Dirac delta distribution assigning all probability mass to the initial timestep T. The **transition dynamics** are deterministic:

$$P(s_{t+1} \mid s_t, a_t) \triangleq \left(\delta_{t-1}, \delta_{\boldsymbol{z}_{t-1}}\right), \tag{14}$$

indicating a fixed update from $\boldsymbol{z}_t$ to $\boldsymbol{z}_{t-1}$, and a decrement in the timestep. The **reward function** is sparse, defined as

$$R(s_t, a_t) \triangleq \begin{cases} r(\hat{\boldsymbol{x}}_{\text{clean}}) & \text{if } t = 0, \\ 0 & \text{otherwise}, \end{cases} \tag{15}$$

where $\hat{x}_{\text{clean}}$ represents the $\hat{\boldsymbol{x}}$ after processing through the artifact removal module. This MDP spans T timesteps per trajectory, with a termination state reached at $t = 0$ according to the transition probability P. The cumulative reward of each trajectory is equal to $r(\hat{\boldsymbol{x}}_{\text{clean}})$. Therefore, maximizing $\mathcal{J}_{\text{RL}}(\pi_\theta)$ in this MDP is equivalent to fine-tuning the underlying DDPM.

Gradient-based Policy Optimization. To maximizing the RL objective $\mathcal{J}_{\text{RL}}(\pi_\theta)$, we adopt the policy gradient method that alternates between (i) sampling denoising trajectories $\{\boldsymbol{x}_T, \dots, \boldsymbol{x}_0\}$, (ii) estimating $\nabla_\theta \mathcal{J}_{\text{RL}}$ from these trajectories, and (iii) updating the policy parameters via gradient ascent.

As a widely used gradient estimator in policy gradient methods, REINFORCE [36][37] leverages the likelihood ratio trick to estimate the policy gradient. Based on our MDP formulation, the gradient can be expressed as

$$\nabla_\theta \mathcal{J}_{\text{RL}} = \mathbb{E}\left[\sum_{t=0}^{T} \nabla_\theta \log p_\theta(\boldsymbol{z}_{t-1} \mid \boldsymbol{z}_t) \cdot r(\hat{\boldsymbol{x}}_{\text{clean}})\right], \tag{16}$$

where the expectation is taken over denoising trajectories generated by the current parameters θ. This on-policy property restricts learning efficiency, as each round of collected data allows only

a single gradient update. To perform multiple steps of optimization, the importance sampling estimator [38] can be employed to reuse previously collected trajectories.

$$\nabla_\theta \mathcal{J}_{\mathrm{RL}} = \mathbb{E}\left[\sum_{t=0}^{T} \frac{p_\theta(\boldsymbol{z}_{t-1} \mid \boldsymbol{z}_t)}{p_{\theta_{\mathrm{old}}}(\boldsymbol{z}_{t-1} \mid \boldsymbol{z}_t)} \nabla_\theta \log p_\theta(\boldsymbol{z}_{t-1} \mid \boldsymbol{z}_t)\, r(\hat{\boldsymbol{x}}_{\mathrm{clean}})\right], \tag{17}$$

where the expectation is taken over denoising trajectories generated by the parameters θ_{old} . The importance weights compensate for policy drift, allowing unbiased gradient estimates from old denoising trajectories. However, this estimator becomes inaccurate when p_θ diverges significantly from $p_{\theta_{\mathrm{old}}}$, which can be mitigated by applying trust region methods [39] that constrain the update magnitude. To prevent instability from large policy updates, we employ trust region constraints through clipping, similar to PPO:

$$\mathrm{ClippedRatio} = \mathrm{clip}\left(\frac{p_\theta(\boldsymbol{z}_{t-1} \mid \boldsymbol{z}_t)}{p_{\theta_{\mathrm{old}}}(\boldsymbol{z}_{t-1} \mid \boldsymbol{z}_t)}, 1-\epsilon, 1+\epsilon\right), \tag{18}$$

where ϵ is a small constant that limits how far the new policy can deviate from the old one.

Flexible Reward Functions. To accommodate different layout pattern generation preferences, we formulate a flexible reward evaluated on the final denoised layout $\hat{\boldsymbol{x}}_{\mathrm{clean}}$:

$$r(\hat{\boldsymbol{x}}_{\mathrm{clean}}) = \lambda \cdot f(\hat{\boldsymbol{x}}_{\mathrm{clean}}) - \psi(\hat{\boldsymbol{x}}_{\mathrm{clean}}), \tag{19}$$

where $f(\cdot)$ is a pluggable objective reflecting the generation goal, $\psi(\cdot)$ denotes the total number of design rule violations, and λ balances the objective term against the DRC penalty. This formulation allows our framework to adapt to a range of task objectives while discouraging illegal patterns.

Complexity-based Objective. To generate layout patterns with controllable complexity, we introduce a complexity-based objective that can be configured to favor either more complex or simpler layout patterns. The objective function $f(\hat{\boldsymbol{x}}_{\mathrm{clean}})$ is defined as

$$f(\hat{\boldsymbol{x}}_{\mathrm{clean}}) = s \cdot (c_x + c_y), \tag{20}$$

where c_x and c_y denote the number of scan lines minus one along the x- and y-axes, respectively. The sign parameter $s \in \{+1, -1\}$ determines the optimization direction, with $s = +1$ favoring higher-complexity layout patterns and $s = -1$ favoring simpler layout patterns. We set $\lambda = 1.0$ to balance the complexity objective against the DRC penalty. This objective formulation allows the diffusion model to be tuned to different complexity preferences, so that it can generate large batches of layout patterns matching the intended complexity during inference, resulting in a diverse pattern library.

Process Window-based Objective. Another representative objective is to identify layout patterns with minimal process window sizes, which is valuable for design rule refinement and OPC calibration during early technology development. For a generated layout $\hat{\boldsymbol{x}}_{\mathrm{clean}}$, lithography simulation evaluates the printed CD as a function of defocus ΔF and dose offset ΔE, denoted as $CD(\Delta F, \Delta E; \hat{\boldsymbol{x}}_{\mathrm{clean}})$. For each fixed ΔF, the curve of CD versus ΔE is the Bossung curve[40]. Given the target CD CD_{target} and the tolerance band ΔCD_{tol} (e.g., corresponding to 0.9×–1.1× the target CD), we define the feasible process window region:

$$\mathcal{W}(\hat{\boldsymbol{x}}_{\mathrm{clean}}) = \left\{(\Delta F, \Delta E) \in \mathcal{D} \;\middle|\; \left|CD(\Delta F, \Delta E; \hat{\boldsymbol{x}}_{\mathrm{clean}}) - CD_{\mathrm{target}}\right| \le \Delta CD_{\mathrm{tol}}\right\}, \tag{21}$$

where $\mathcal{D} = [-90\,\mathrm{nm},\ 90\,\mathrm{nm}] \times [-10\%,\ 10\%]$ is the simulated focus–dose range.

Following Definition 2, we quantify the process window size by the area of the largest ellipse that can be fully inscribed in $\mathcal{W}(\hat{\boldsymbol{x}}_{\mathrm{clean}})$. We parameterize an ellipse by

$$\mathcal{E}(\mu, \Sigma) = \left\{u \in \mathbb{R}^2 \;\middle|\; (u-\mu)^\top \Sigma^{-1} (u-\mu) \le 1\right\}, \quad u = (\Delta F, \Delta E), \tag{22}$$

ALGORITHM 1: RL Fine-tuning for Layout Pattern Generation

Input: Pretrained diffusion model p_θ, total steps T, reward function $r(\cdot)$, total epochs E
Output: Updated policy parameters θ

// Fine-Tuning Loop
for epoch = 1 to E **do**
  Sample noise $z_T \sim \mathcal{N}(0, I)$;
  Initialize trajectory $\tau \leftarrow \{\}$;
  for $t = T,\ T-1,\ \ldots,\ 1$ **do**
    Compute $z_{t-1} \sim p_\theta(z_{t-1} \mid z_t)$;
    Add transition $(s_t = (t, z_t), a_t = z_{t-1})$ to τ;
  end for
// Decoding and Reward Computation
  Decode: $\hat{x} \leftarrow \mathcal{D}(z_0)$;
  Apply artifact removal to get $\hat{x}_{\text{clean}}$;
  Compute reward $r = r(\hat{x}_{\text{clean}})$ using Equation (19);
// Policy Update
  Estimate gradient using Equation (17);
  Update $\theta \leftarrow \theta + \eta \cdot \nabla_\theta \mathcal{J}_{\text{RL}}$;
end for
return θ;

where $\mu \in \mathbb{R}^2$ is the ellipse center and $\Sigma \succ 0$ is a 2×2 positive-definite matrix determining the ellipse shape (axis lengths and rotation). The ellipse area is $\text{Area}(\mathcal{E}(\mu, \Sigma)) = \pi\sqrt{\det(\Sigma)}$. We then define:

$$\text{pw_size}(\hat{x}_{\text{clean}}) = \max_{\mu,\ \Sigma \succ 0} \pi\sqrt{\det(\Sigma)} \quad \text{s.t.} \quad \mathcal{E}(\mu, \Sigma) \subseteq \mathcal{W}(\hat{x}_{\text{clean}}). \tag{23}$$

Accordingly, we set:

$$f(\hat{x}_{\text{clean}}) = -\,\text{pw_size}(\hat{x}_{\text{clean}}), \tag{24}$$

so that patterns with smaller process windows receive higher reward and are preferentially generated. We set $\lambda = 2.0$ to increase the influence of the process window term relative to the DRC penalty. In practice, $CD(\Delta F, \Delta E; \hat{x}_{\text{clean}})$ and $\text{pw_size}(\hat{x}_{\text{clean}})$ are computed by commercial lithography simulation tools on a discrete focus–exposure matrix over $\mathcal{D}$.

Our layout pattern generation framework fine-tunes the original diffusion model via RL, eliminating the need for manual annotation or supervised training, thereby enabling the generation of novel pattern combinations beyond the original data distribution. By flexibly designing reward functions, the framework can adapt to various DFM requirements. We summarize the RL fine-tuning procedure in Algorithm 1.

3.4 Training and Inference Workflow

ForgePattern is an RL-based fine-tuning framework built on top of a pretrained diffusion layout generator. We describe the full pipeline below for completeness, including how the base generator is obtained, how it is fine-tuned by ForgePattern, and how inference is performed.

Diffusion Pretraining. We pretrain a latent diffusion model using the standard DDPM noise-prediction objective (Section 2.1) on unlabeled layout patterns. This pretrained U-Net provides the generative backbone for ForgePattern and serves as the starting point for subsequent RL fine-tuning. It can already generate layout samples via standard reverse diffusion.

RL Fine-tuning. Starting from the pretrained generator, we fine-tune the denoising U-Net using policy-gradient optimization (Section 3.3, Algorithm 1). Each iteration samples denoising trajec-

tories, decodes and post-processes the resulting layout, evaluates a task-specific scalar reward computed from DFM metrics, and updates the model parameters to maximize expected reward.

Inference. At inference time, model parameters are fixed. We sample $z_T \sim \mathcal{N}(0, I)$, run T reverse denoising steps to obtain z_0, decode to $\hat{x}$, and apply the artifact removal module to produce $\hat{x}_{\text{clean}}$. Temporal cache reuse can be enabled to accelerate denoising (Section 3.2). No reward is computed and no parameter update is performed during inference.

4 Experiment

4.1 Experimental Setup

ICCAD Dataset. Following previous work [13],[14],[18],[25],[26], we construct the dataset by splitting a $160 \times 400\ \mu m^2$ layout map from the ICCAD 2014 contest into small layout patterns with the size of $2048 \times 2048\ nm^2$, resulting in 13,869 images.

ASAP7 Dataset. To verify our proposed method on advanced technology nodes, we obtain the dataset of small layout patterns by splitting a $200 \times 124\ \mu m^2$ layout map from the 7 nm RISC-V BOOM core CPU design. The layout is generated by Chipyard using the ASAP7 PDK. This dataset contains 15,209 images with the size of $1024 \times 1024\ nm^2$.

DRC Rule Settings. As illustrated in Figure 3, the design rules of a layout pattern include minimum polygon spacing, minimum polygon width, and polygon area constraints. For the ICCAD dataset, following prior work[18], the minimum polygon spacing is set to 6 nm, the minimum width to 2 nm, the minimum polygon area to 48 nm^2, and the maximum polygon area to $3.87 \times 10^6 nm^2$. For the ASAP7 dataset, we set both the minimum polygon spacing and minimum width to 18 nm, while keeping the minimum and maximum polygon area unchanged.

Diffusion Model Configuration. We use a U-Net [24] as the backbone in the latent diffusion model to predict the posterior distribution during the reverse diffusion process, implemented using `UNet2DModel` from HuggingFace `diffusers`. The input is a 64×64 latent representation with 4 channels. The model includes five resolution levels with spatial sizes [64×64, 32×32, 16×16, 8×8, 4×4], and channel widths of $[256, 512, 512, 512, 1024]$. Each level contains two residual blocks. Self-attention is applied at intermediate resolutions (32×32, 16×16, and 8×8) via `AttnDownBlock2D`.

Diffusion Model Training Details. We train the latent diffusion model for 100,000 iterations with a batch size of 32, using the Adam optimizer with a learning rate of 1×10^{-4}. We set diffusion steps $T = 1000$ to ensure that the forward process converges to a Gaussian distribution. The noise schedule linearly increases β_t from 0.00085 to 0.012. Other hyperparameters follow the default settings of Stable Diffusion v1.4. Training takes about 23 hours on 2 NVIDIA A100 GPUs.

Diffusion Model Fine-tuning Details. For compute-efficient fine-tuning, we use **Low-Rank Adaptation** (**LoRA**) [41], which introduces low-rank trainable weights into the attention modules of the pre-trained model while keeping the original weights frozen. We apply LoRA to the UNet module and only update the added weights. We collect 256 samples per training iteration, performing eight gradient updates per iteration with mini-batches of size 32. We use the AdamW optimizer with a learning rate of 3×10^{-4} and $\beta = (0.9, 0.999)$. The gradient clipping threshold ϵ in Equation (10) is set to 1×10^{-4}. In our experiments, 50 iterations of RL fine-tuning take approximately 1 hour for complexity control on 2 NVIDIA A100 GPUs. The same 50 iterations take about 15 hours for process-window control, primarily because computing the `pw_size` reward requires costly runs of a commercial lithography simulator. During RL fine-tuning, terminal rewards are normalized to zero mean and unit variance within each minibatch before computing policy-gradient updates, which serves as a variance-reduction baseline and improves training stability.

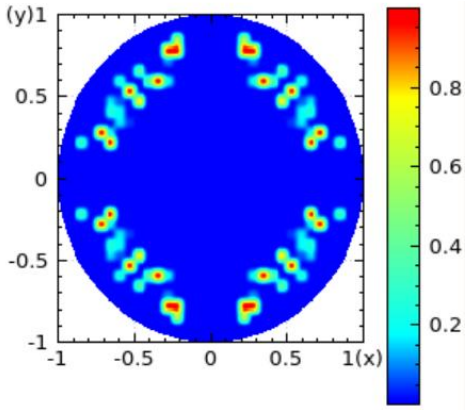


Fig. 8. Illumination source used for process window calculation.

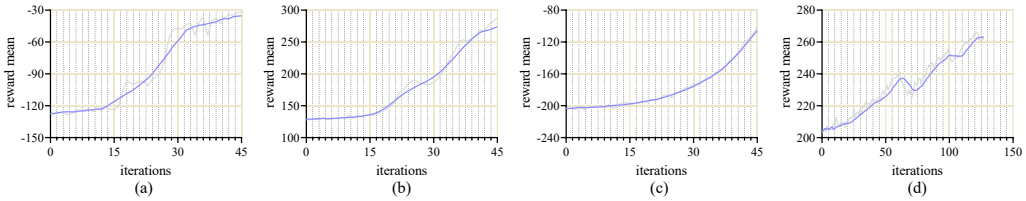


Fig. 9. Reward curves of generated layout patterns. (a) Low-complexity patterns trained on the ICCAD dataset. (b) High-complexity patterns trained on the ICCAD dataset. (c) Low-complexity patterns trained on the ASAP7 dataset. (d) High-complexity patterns trained on the ASAP7 dataset.

Lithography Simulation Setup. Generated images are vectorized into GDSII format using gdspy with a 5× scaling, corresponding to 25 nm^2 per pixel. Multiple layout patterns are tiled into a single GDS cell with a spacing of 300 nm to enable batch lithography simulation. CD gauges are automatically placed at geometry-sensitive regions, and the resulting GDS and gauge files are passed to the lithography simulator for process-window evaluation under the specified focus–exposure conditions. The exposure wavelength is set to 193 nm with a **numerical aperture** (**NA**) of 1.35. The illumination source is shown in Figure 8. The F–E matrix spans $\Delta F \in [-90\,\text{nm}, +90\,\text{nm}]$ and $\Delta D \in [-10\%, +10\%]$, sampled on a uniform grid.

4.2 Pattern Diversity and Legality

To generate layout patterns with high complexity, we define the $f(\hat{x}_{\text{clean}})$ in Equation (19) as the complexity of $\hat{x}_{\text{clean}}$. During training, we generate 256 layout patterns per iteration and track the progression of their average complexity over time. As illustrated in Figure 9(b) and (d), the average complexity of the generated layout patterns steadily increases during training across different datasets. Conversely, when $f(\hat{x}_{\text{clean}})$ is defined as the negative of the complexity of $\hat{x}_{\text{clean}}$, the model is encouraged to generate layout patterns with lower complexity. As shown in Figure 9(a) and (c), the model effectively reduces complexity in both datasets. We further study how the complexity optimization trend relates to the reward scaling hyperparameter λ. Figure 10 reports the evolution of the mean complexity on ASAP7 under different λ values. When λ is set too small ($\lambda = 0.1$), the complexity optimization becomes slower and less stable. In contrast, the trends are similar for $\lambda \geq 1$, suggesting that performance is not sensitive to λ as long as it is chosen within a reasonable

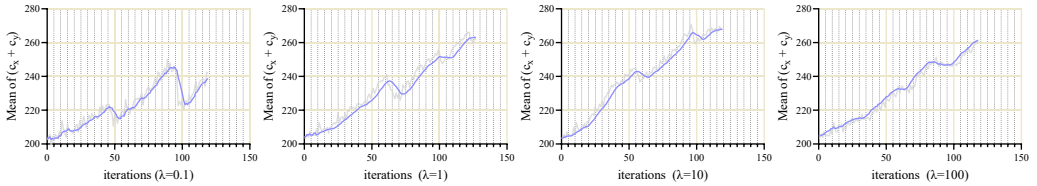


Fig. 10. Evolution of the mean pattern complexity $(c_x + c_y)$ during RL fine-tuning on the ASAP7 dataset under different reward scaling parameters λ. Similar optimization trends are observed for $\lambda \geq 1$, while a too-small value ($\lambda = 0.1$) results in slower and less stable improvement.

range. Figure 11 presents layout pattern examples from different training iterations, validating that Forge Pattern can successfully generate detailed, reasonable patterns and control their complexity across multiple datasets.

Building on these observations, we further enhance the diversity of our layout pattern library by utilizing models with different levels of complexity. Specifically, we select two intermediate models at 10-iteration intervals from both high-complexity and low-complexity training trajectories. Together with the original diffusion model, this yields five distinct complexity levels. Each model is used to generate 20,000 layout patterns, resulting in a comprehensive library of 100,000 layout patterns. Then we adopt the same legalization method as in [18], and the legality is checked via a tool Klayout based on the design rules described in Figure 3.

We compare ForgePattern with several learning-based layout pattern generation pipelines. CAE [12] refers to a standard convolutional autoencoder model and VCAE [13] refers to a variational convolutional auto-encoder model. By applying random perturbations in the probabilistic latent space, VCAE can generate more diverse layout pattern. Legal-GAN [13] is a post-processing method based on conditional **generative adversarial networks** (**CGANs**), which refines the generated topologies to ensure compliance with design rules. LayouTransformer [14] is a sequence-based method that converts 2D layout patterns into 1D token sequences and uses a Transformer architecture to generate diverse and legal layout patterns autoregressively. DiffPattern [18] utilizes a discrete diffusion model to synthesize layout topology matrices. ChatPattern [25] and ControLayout [26] extend this approach by constructing image-text pairs to train a conditional diffusion model, enabling category-specific or complexity-specific layout pattern generation. Although the ChatPattern and ControLayout support limited layout customization, their reliance on paired training data makes them unsuitable for flexible or implicit generation goals. In addition, fine-tuning on fixed datasets often leads to degradation in image quality. Since prior works were only evaluated on the ICCAD dataset, we also focus our comparison on this dataset to provide a fair and direct performance assessment. We adopt three key evaluation metrics: *Generated Pattern Diversity*, which measures the diversity across all generated patterns regardless of legality. *Legality*, defined as the proportion of generated layout patterns (out of 100,000 samples) that satisfy all design rules, and *Legal Diversity*, which assesses diversity within the subset of legal layout patterns. As shown in Table 1, thanks to the superior complexity control of ForgePattern, our synthesized layout pattern library covers a wider range of pattern complexities, resulting in significantly higher diversity compared to previous methods. Although the increased complexity leads to a slight decrease in legality compared to DiffPattern, the improvement in legal diversity remains substantial. Since diversity is quantified by the Shannon entropy of the distribution of pattern complexity (Equation (6)), we plot in Figure 12 the complexity distributions of real layouts and generated layout patterns across different datasets to show how ForgePattern changes the complexity distribution. As shown in Figure 12, ForgePattern expands the complexity range beyond real layouts, demonstrating its ability to synthesize novel combinations of existing patterns.

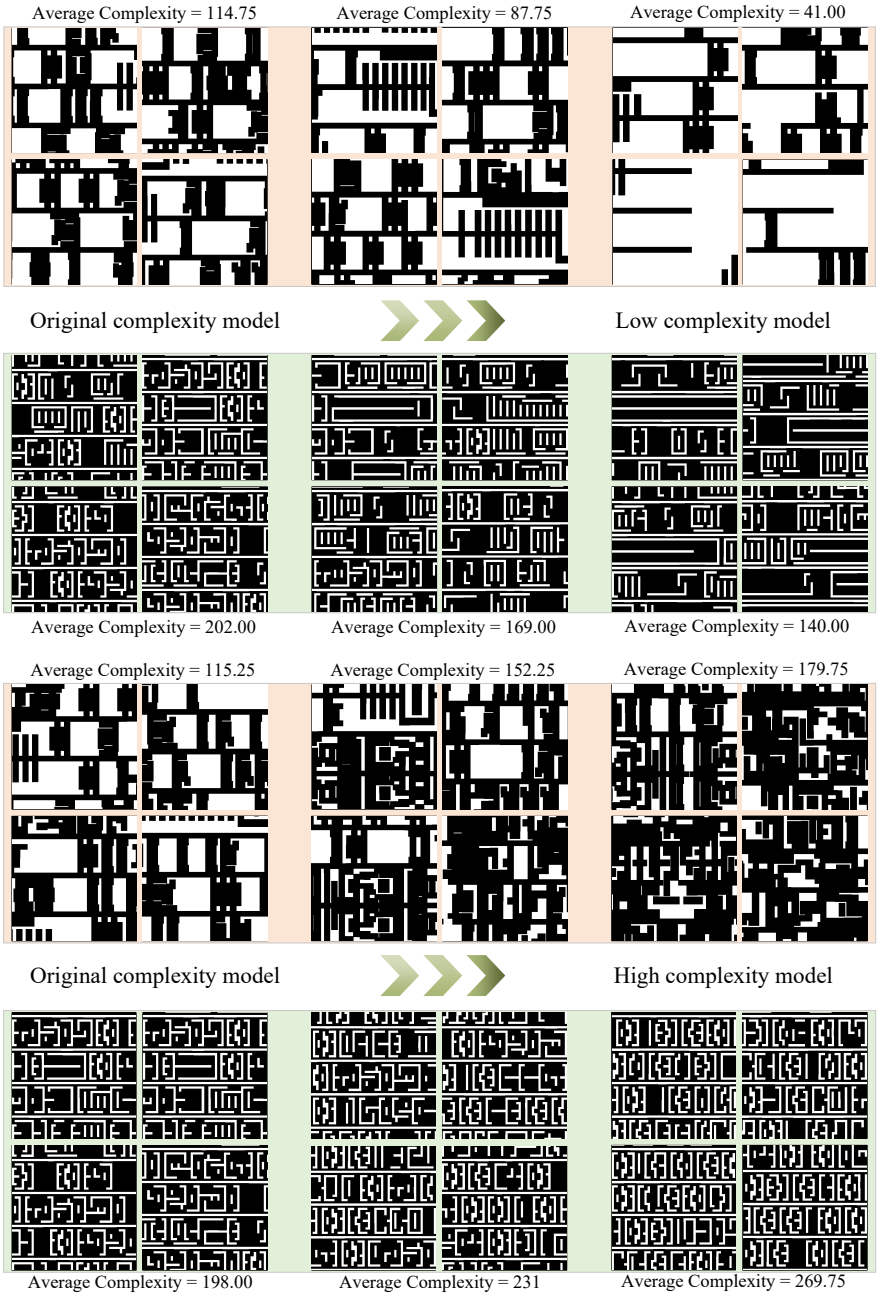


Fig. 11. Examples of layout patterns generated during fine-tuning. The original diffusion model is progressively fine-tuned by a reward function, generating layout patterns of different complexity.

To systematically evaluate the impact of RL fine-tuning and the temporal reuse mechanism, we also performed ablation studies on the ASAP7 dataset. Results are reported in Table 2. After applying our RL fine-tuning strategy, the legal diversity achieves a significant increase from 10.660 to 11.806, which can be attributed to the enhanced complexity coverage of generated layout patterns.

Table 1. Comparison of Layout Pattern Diversity and Legality

Method	Published In	Generated Pattern Diversity (↑)	Legality	Legal Diversity (↑)
Real Patterns	–	–	–	10.777
CAE [12]	DAC 2019	4.588	0.02%	3.787
VCAE [13]	ICCAD 2020	10.931	2.13%	9.978
CAE+LegalGAN [13]	ICCAD 2020	5.847	3.74%	5.814
VCAE+LegalGAN [13]	ICCAD 2020	9.869	84.51%	9.866
LayoutTransformer [14]	ICCAD 2022	10.532	89.73%	10.527
DiffPattern [18]	DAC 2023	10.815	99.97%	10.711
ControLayout [26]	GLSVLSI 2024	10.639	92.32%	10.622
ChatPattern [25]	DAC 2024	10.815	99.97%	10.796
ForgePattern (Ours)	This article	**11.141**	98.12%	**11.072**

Note: 'Real Patterns' are from the ICCAD 2014 contest. '-' denotes an inapplicable result.

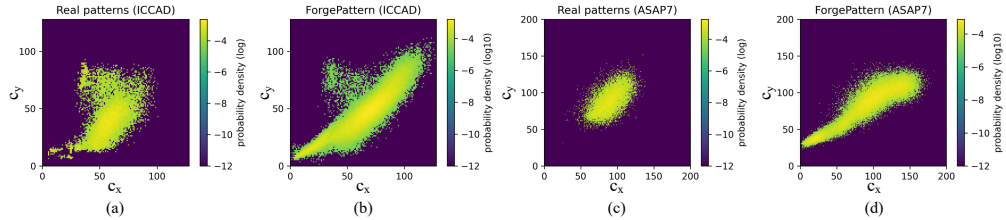


Fig. 12. Complexity distributions of real designs and generated layout patterns. (a) Distribution for the ICCAD dataset. (b) Distribution for layout patterns generated by ForgePattern pretrained on the ICCAD dataset. (c) Distribution for the ASAP7 dataset. (d) Distribution for layout patterns generated by ForgePattern pretrained on the ASAP7 dataset.

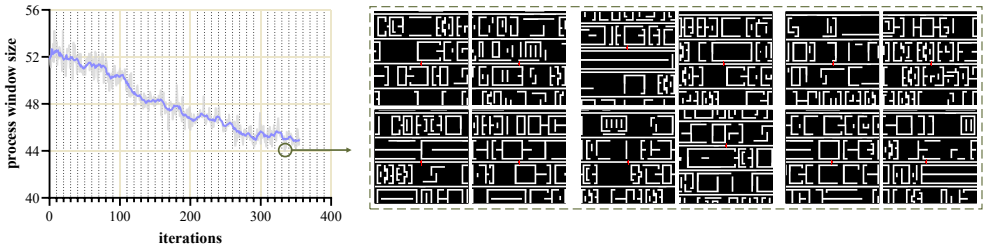


Fig. 13. The curve of generated layout patterns about the size of the process window. The dashed box highlights generated layout patterns from a representative iteration, showcasing ForgePattern's capability to produce diverse weak patterns. Gauges are placed at the red markers in each layout pattern.

In Table 2, we also report the generation time for a single layout pattern under different ablation settings on a single NVIDIA A100 GPU. By employing the temporal reuse mechanism and setting $N = 3$ in Equation (7), our proposed ForgePattern achieves approximately a 2.5× speedup in layout pattern generation. Under this configuration, a library of more than 10,000 patterns can be constructed in about 3 hours on a single A100 GPU, with a peak GPU memory footprint of about 30 GB at a batch size of 16. These results demonstrate the effectiveness of both RL fine-tuning and the temporal reuse mechanism.

4.3 Identification of Patterns with Small Process Windows

The process window serves as a key manufacturability indicator in semiconductor manufacturing, capturing how robust a layout pattern is to variations in focus and exposure. Patterns exhibiting

Table 2. Ablation Studies of Our Proposed ForgePattern Using the ASAP7 Dataset

Method	Legality	Legal Diversity (↑)	Tgen (s)
w/o. RL and Temporal Reuse	100%	10.653	2.95
w/o. RL	100%	10.660	1.21
w/o. Temporal Reuse	100%	11.795	2.96
Ours	100%	**11.806**	**1.21**

Note: 'w/o' denotes 'without'. 'Tgen (s)' represents the single layout pattern generation time in seconds.

small process windows are more prone to lithographic failures, so pinpointing them is crucial for improving OPC strategies. Yet collecting a broad and representative set of such weak layouts remains difficult because their geometric features are subtle, non-intuitive, and only sparsely represented in existing datasets.

Traditionally, identifying such weak patterns relied heavily on the expertise of OPC engineers in conjunction with commercial simulation tools. By contrast, ForgePattern is able to synthesize them automatically, producing large volumes of weak patterns with high diversity. During training, we compute the average reward of legal layouts generated at each iteration according to Equation (19), where $f(\hat{x}_{\text{clean}}) = -pw_size$. As shown in Figure 13, it can be observed that, as training progresses, the process window sizes of the layout patterns generated by ForgePattern become increasingly smaller. We also present the generated results from one representative iteration to demonstrate ForgePattern's ability to produce diverse weak patterns. The results reveal recurring characteristics among some patterns, such as sparse geometric features, which provide new insights for refining design rules.

Although prior generative methods may also produce weak patterns with small process windows, the key challenge lies in sampling efficiency and controllability. Unconditional models such as DiffPattern [18] typically rely on a "generate-and-filter" strategy: they randomly generate many candidates and then use lithography simulation to identify weak patterns. Since such weak patterns are rare, this approach requires extensive sampling and simulation, resulting in prohibitive cost.

Text-conditioned methods [25, 26] have additional limitations. They require paired prompt–layout supervision, yet weak patterns are rare, making it difficult to collect sufficient prompt–layout pairs. Moreover, quantitative manufacturability targets such as a specific **process window** (**PW**) value computed by lithography simulation cannot be expressed precisely through natural-language prompts. As a result, text conditioning typically provides only coarse control, can be sensitive to prompt wording, and is difficult to align consistently to specific PW values across different samples.

In contrast, ForgePattern uses the size of the process window as a direct numeric reward for RL fine-tuning. By maximizing the expected reward, it explicitly shifts the sampling distribution toward smaller process windows. After fine-tuning, the model can generate weak patterns with small process windows in a stable and efficient manner.

4.4 Monitoring Strategy Changes via Fixed Noise

As shown in Figure 14, we can fix the random seed for noise sampling, which allows us to track the progression of generated images from the same noise input over the course of training. The use of fixed noise input ensures that variations in generated patterns arise solely from updates to the model weights, resulting in high structural similarity across different generation strategies. This consistency, in turn, facilitates the identification of critical shifts in generation strategies. For instance, Figure 14(b) shows that density changes around the gauge can significantly impact the process window size, which offers new insights into the underlying causes of process window size reductions.

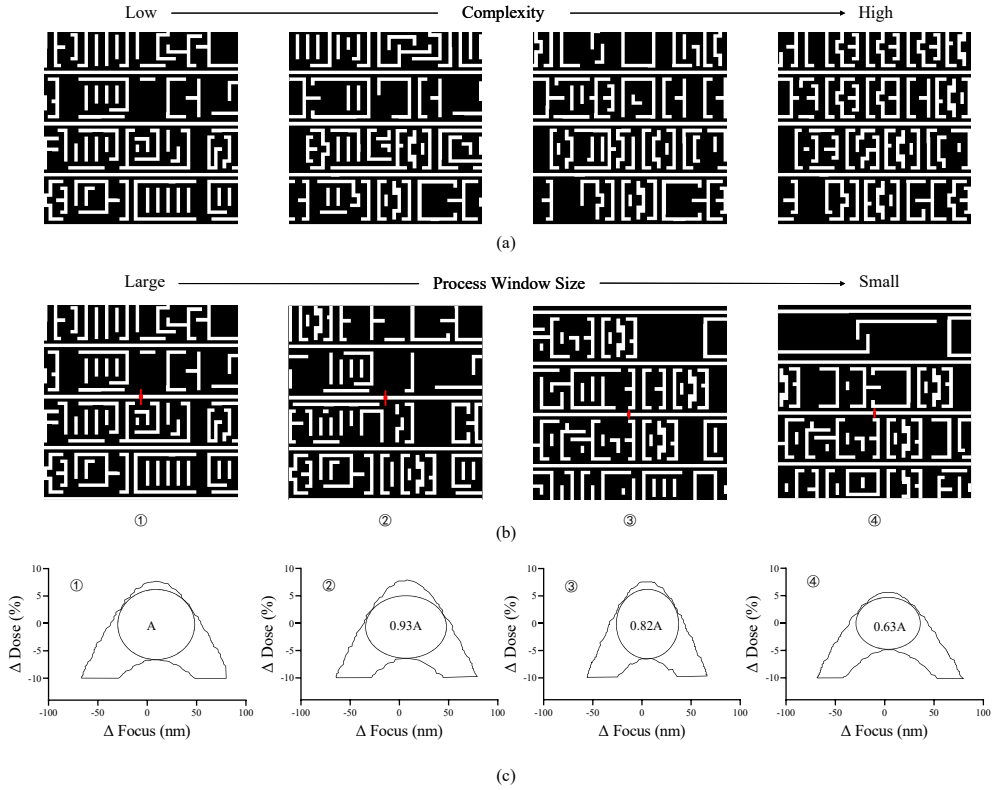


Fig. 14. (a) illustrates the progression of generated layout patterns under a fixed noise input during the training process, with a focus on the complexity metric. (b) shows a similar progression, this time in terms of the process window size metric. Gauges are placed at the red markers in each layout pattern of (b), and commercial software is used to compute the corresponding process windows shown in (c). The variations in complexity and process window size demonstrate the effectiveness of our method in customizing layout pattern generation. (white regions represent the metal shapes).

Thus, the benefits of our proposed ForgePattern extend beyond generating high-reward layout patterns. It also provides a clear and interpretable evolution path, making it a valuable tool for downstream DFM studies and analysis.

4.5 Large-scale Layout Generation via Outpainting

In practical DFM workflows, layout clips of different sizes are sometimes required depending on the specific analysis or optimization objective, such as full-chip-level OPC. Although ForgePattern is trained with a fixed layout size, its diffusion-based formulation naturally supports generating much larger layouts through patch-wise outpainting.

Let the target layout size be $W \times H$. The diffusion model generates square patches of size $L \times L$ (in our case $L = 512$), which we treat as its generation window. During outpainting, this $L \times L$ window is moved in a raster-scan manner: starting from the top-left corner, the window shifts horizontally with stride S until the right boundary is reached, then moves down by stride S to start the next row, as illustrated in Figure 15(a). Only the content within the current window participates in the diffusion process, which means that the memory cost depends on L but is independent of the final layout size $W \times H$.

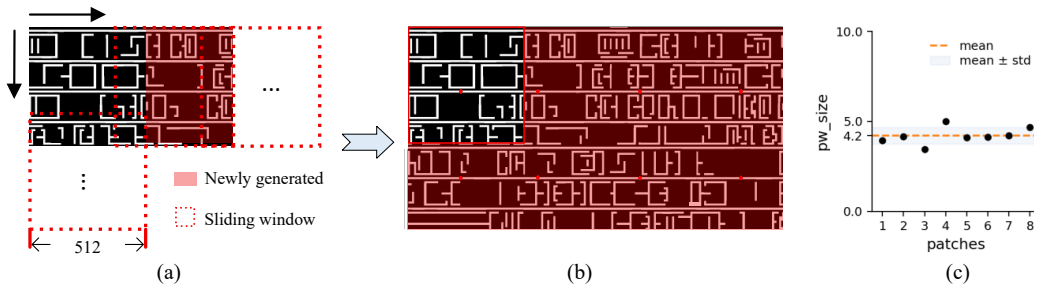


Fig. 15. Pattern extension with sliding-window out-painting. (a) A 512 × 512 patch window slides over the canvas from left to right and then top to bottom, generating the adjacent region (red) conditioned on the already generated part of the layout. (b) The final extended layout produced by this sliding out-painting procedure. (c) pw_size of each generated patch, with the dashed line indicating the mean and the shaded band showing the mean ± standard deviation.

Given (W, H), window size L, and stride S, the numbers of horizontal and vertical window locations are

$$N_x = \left\lceil \frac{W - L}{S} \right\rceil + 1, \qquad N_y = \left\lceil \frac{H - L}{S} \right\rceil + 1, \tag{25}$$

where $\lceil \cdot \rceil$ denotes the ceiling operator, which maps a real number to the smallest integer greater than or equal to it. The total number of outpainting samplings is $N_{\text{out}} = N_x N_y$. Each sampling has the same computational and memory footprint as generating a single $L \times L$ patch. Therefore the method can scale to arbitrarily large W and H while keeping the memory requirement constant. The scalability of ForgePattern primarily arises from its diffusion-based outpainting capability and the patch-wise generation scheme, and is not intrinsically tied to any specific technology node.

To evaluate scalability in practice, we use the diffusion model that has been fine-tuned with the pw_size as the RL reward. Starting from a 512 × 512 seed layout, we extend it to a 1664 × 896 region using a 512 × 512 sliding window with stride $S = 384$, which corresponds to a 128-pixel overlap between neighboring windows. According to Equation (25), this configuration yields $N_x = 4, N_y = 2, N_{\text{out}} = 8$. Figure 15(a) illustrates the sliding-window outpainting process, and Figure 15(b) shows the resulting large layout. For each of the eight generated patches we compute the pw_size. The scatter plot in Figure 15(c) shows the pw_size of each patch, together with the global mean and the one-standard-deviation interval (±0.46). The pw_size remains tightly clustered around the mean, indicating that our ForgePattern method can be scaled to substantially larger layouts via sliding-window outpainting while preserving local process-window characteristics.

5 Conclusion

In this article, we propose ForgePattern, a novel and flexible framework for layout pattern generation tailored to diverse DFM objectives. By integrating RL into the denoising process of our efficient layout pattern generation backbone, ForgePattern enables fine-grained control over objectives such as pattern complexity and process window size. This framework eliminates the need for annotated datasets and adapts easily to varying DFM requirements. Experimental results show that ForgePattern can accurately tune the generation process to meet user-specific goals, thus providing valuable resources for downstream DFM studies. In addition, the generated layouts can be vectorized into GDSII format, ensuring compatibility with mainstream commercial EDA tools.

We also acknowledge several limitations. Our current implementation focuses on a single layer, lacks modeling of hierarchical layout structures, and primarily targets single-objective optimization.

In future work, we plan to incorporate hard DRC constraints during generation, model multi-layer interactions, and investigate multi-objective optimization, especially for jointly optimizing competing objectives.

References

[1] Jhih-Rong Gao, Xiaoqing Xu, Bei Yu, and David Z. Pan. 2014. MOSAIC: Mask optimizing solution with process window aware inverse correction. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[2] Ziyang Yu, Guojin Chen, Yuzhe Ma, and Bei Yu. 2022. A GPU-enabled level-set method for mask optimization. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 2 (2022), 594–605.

[3] Ziyang Yu, Peiyu Liao, Yuzhe Ma, Bei Yu, and Martin D. F. Wong. 2023. CTM-SRAF: Continuous transmission mask-based constraint-aware subresolution assist feature Generation. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 42, 10 (2023), 3402–3411.

[4] Su Zheng, Yuzhe Ma, Bei Yu, and Martin D. F. Wong. 2024. EMOGen: Enhancing mask optimization via pattern generation. In *Proceedings of the 61st ACM/IEEE Design Automation Conference*. 1–6.

[5] Jian Kuang and Evangeline F. Y. Young. 2013. An efficient layout decomposition approach for triple patterning lithography. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[6] Bei Yu, Kun Yuan, Duo Ding, and David Z. Pan. 2015. Layout decomposition for triple patterning lithography. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems* 34, 3 (2015), 433–446.

[7] Su Zheng, Haoyu Yang, Binwu Zhu, Bei Yu, and Martin Wong. 2023. Lithobench: Benchmarking Ai computational lithography for semiconductor manufacturing. *Advances in Neural Information Processing Systems* 36 (2023), 30243–30254.

[8] Ran Chen, Wei Zhong, Haoyu Yang, Hao Geng, Xuan Zeng, and Bei Yu. 2019. Faster region-based hotspot detection. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[9] Hao Geng, Haoyu Yang, Lu Zhang, Jin Miao, Fan Yang, Xuan Zeng, and Bei Yu. 2020. Hotspot detection via attention-based deep layout metric learning. In *Proceedings of the 39th International Conference on Computer-Aided Design*. 1–8.

[10] Gaurav Rajavendra Reddy, Constantinos Xanthopoulos, and Yiorgos Makris. 2018. Enhanced hotspot detection through synthetic pattern generation and design of experiments. In *Proceedings of the IEEE VLSI Test Symposium (VTS)*. IEEE, 1–6.

[11] Wei Ye, Yibo Lin, Meng Li, Qiang Liu, and David Z. Pan. 2019. LithoROC: Lithography hotspot detection with explicit ROC optimization. In *Proceedings of the IEEE/ACM Asia and South Pacific Design Automation Conference (ASPDAC)*. 292–298.

[12] Haoyu Yang, Piyush Pathak, Frank Gennari, Ya-Chieh Lai, and Bei Yu. 2019. DeePattern: Layout pattern generation with transforming convolutional auto-encoder. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 1–6.

[13] Xiaopeng Zhang, James Shiely, and Evangeline F. Y. Young. 2020. Layout pattern generation and legalization with generative learning models. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–9.

[14] Liangjian Wen, Yi Zhu, Lei Ye, Guojin Chen, Bei Yu, Jianzhuang Liu, and Chunjing Xu. 2022. Layoutransformer: Generating layout patterns with transformer via sequential pattern modeling. In *Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD)*. 1–9.

[15] Lukas Höllein, Aljaž Božič, Norman Müller, David Novotny, Hung-Yu Tseng, Christian Richardt, Michael Zollhöfer, and Matthias Nießner. 2024. Viewdiff: 3D-consistent image generation with text-to-image models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 5043–5052.

[16] Kaede Shiohara and Toshihiko Yamasaki. 2024. Face2diffusion for fast and editable face personalization. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 6850–6859.

[17] Yanzuo Lu, Manlin Zhang, Andy J. Ma, Xiaohua Xie, and Jianhuang Lai. 2024. Coarse-to-fine latent diffusion for pose-guided person image synthesis. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 6420–6429.

[18] Zixiao Wang, Yunheng Shen, Wenqian Zhao, Yang Bai, Guojin Chen, Farzan Farnia, and Bei Yu. 2023. DiffPattern: Layout pattern generation via discrete diffusion. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. IEEE, 1–6.

[19] Xingdi Zhang, Hunglin Chen, Yin Long, and Kai Wang. 2020. Study of process window discovery methodology for 28nm and beyond technology node process window limiting structures. In *Proceedings of the 2020 31st Annual SEMI Advanced Semiconductor Manufacturing Conference (ASMC)*. IEEE, 1–8.

[20] Ayman Hamouda, Mohamed Bahnas, Dan Schumacher, Ioana Graur, Ao Chen, Kareem Madkour, Hussein Ali, Jason Meiring, Neal Lafferty, and Chris McGinty. 2017. Enhanced OPC recipe coverage and early hotspot detection through automated layout generation and analysis. In *Proceedings of the Optical Microlithography XXX*. SPIE, 223–231.
[21] Uwe Paul Schroeder, Janam Bakshi, Ahmed Mounir Elsemary, and Fadi Batarseh. 2020. Dynamic pattern matching flow to enable low escape rate weak point detection. In *Proceedings of the Design-Process-Technology Co-optimization for Manufacturability XIV*. SPIE, 77–83.
[22] Tsung-Wei Lin, Hung-Yu Lin, Meng-Shiun Chiang, Jung-Kuan Huang, Jason Sweis, Philippe Hurat, Chung-Chen Hsu, Chun-Sheng Wu, Chao-Yi Huang, and Ya-Chieh Lai. 2024. Using pattern analysis to improve process window qualification cycle time. In *Proceedings of the DTCO and Computational Patterning III*. SPIE, 374–379.
[23] Qian Xie, Panneerselvam Venkatachalam, Julie Lee, Zhijin Chen, and Khurram Zafar. 2016. Design guided data analysis for summarizing systematic pattern defects and process window. In *Proceedings of the Metrology, Inspection, and Process Control for Microlithography XXX*. SPIE, 771–778.
[24] Olaf Ronneberger, Philipp Fischer, and Thomas Brox. 2015. U-net: Convolutional networks for biomedical image segmentation. In *Proceedings of the International Conference on Medical Image Computing and Computer-Assisted Intervention (MICCAI)*. Springer, 234–241.
[25] Zixiao Wang, Yunheng Shen, Xufeng Yao, Wenqian Zhao, Yang Bai, Farzan Farnia, and Bei Yu. 2024. ChatPattern: Layout pattern customization via natural language. In *Proceedings of the ACM/IEEE Design Automation Conference (DAC)*. 1–6.
[26] Qijing Wang, Xiaopeng Zhang, Martin D. F. Wong, and Evangeline F. Y. Young. 2024. ControLayout: Conditional diffusion for style-controllable and violation-fixable layout pattern generation. In *Proceedings of the ACM Great Lakes Symposium on VLSI (GLSVLSI)*. 511–515.
[27] Jonathan Ho, Ajay Jain, and Pieter Abbeel. 2020. Denoising diffusion probabilistic models. *Annual Conference on Neural Information Processing Systems* 33 (2020), 6840–6851.
[28] Frank E. Gennari and Ya-Chieh Lai. 2014. Topology design using squish patterns. (Sept. 9 2014). US Patent 8,832,621.
[29] Robin Rombach, Andreas Blattmann, Dominik Lorenz, Patrick Esser, and Björn Ommer. 2022. High-resolution image synthesis with latent diffusion models. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 10684–10695.
[30] Xinyin Ma, Gongfan Fang, and Xinchao Wang. 2024. Deepcache: Accelerating diffusion models for free. In *Proceedings of the IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 15762–15772.
[31] Chenlin Meng, Yutong He, Yang Song, Jiaming Song, Jiajun Wu, Jun-Yan Zhu, and Stefano Ermon. 2021. Sdedit: Guided image synthesis and editing with stochastic differential equations. arXiv:2108.01073. Retrieved from https://arxiv.org/abs/2108.01073
[32] Jacob Austin, Daniel D. Johnson, Jonathan Ho, Daniel Tarlow, and Rianne Van Den Berg. 2021. Structured denoising diffusion models in discrete state-spaces. *Advances in Neural Information Processing Systems* 34 (2021), 17981–17993.
[33] Ting Chen, Ruixiang Zhang, and Geoffrey Hinton. 2022. Analog bits: Generating discrete data using diffusion models with self-conditioning. arXiv:2208.04202. Retrieved from https://arxiv.org/abs/2208.04202
[34] Kevin Black, Michael Janner, Yilun Du, Ilya Kostrikov, and Sergey Levine. 2023. Training diffusion models with reinforcement learning. arXiv:2305.13301. Retrieved from https://arxiv.org/abs/2305.13301
[35] Ying Fan, Olivia Watkins, Yuqing Du, Hao Liu, Moonkyung Ryu, Craig Boutilier, Pieter Abbeel, Mohammad Ghavamzadeh, Kangwook Lee, and Kimin Lee. 2023. Dpok: Reinforcement learning for fine-tuning text-to-image diffusion models. *Annual Conference on Neural Information Processing Systems* 36 (2023), 79858–79885.
[36] Shakir Mohamed, Mihaela Rosca, Michael Figurnov, and Andriy Mnih. 2020. Monte carlo gradient estimation in machine learning. *Journal of Machine Learning Research* 21, 132 (2020), 1–62.
[37] Ronald J. Williams. 1992. Simple statistical gradient-following algorithms for connectionist reinforcement learning. *Machine learning* 8, 3 (1992), 229–256.
[38] Sham Kakade and John Langford. 2002. Approximately optimal approximate reinforcement learning. In *Proceedings of the International Conference on Machine Learning (ICML)*. 267–274.
[39] John Schulman, Sergey Levine, Pieter Abbeel, Michael Jordan, and Philipp Moritz. 2015. Trust region policy optimization. In *Proceedings of the International Conference on Machine Learning (ICML)*. PMLR, 1889–1897.
[40] Terrence E. Zavecz. 2006. Bossung curves: An old technique with a new twist for sub-90-nm nodes. In *Proceedings of the Metrology, Inspection, and Process Control for Microlithography XX*. SPIE, 956–965.
[41] Edward J. Hu, Yelong Shen, Phillip Wallis, Zeyuan Allen-Zhu, Yuanzhi Li, Shean Wang, Lu Wang, Weizhu Chen, et al. 2022. Lora: Low-rank adaptation of large language models. *International Conference on Learning Representations* 1, 2 (2022), 3.

Received 16 October 2025; revised 17 January 2026; accepted 21 February 2026