# DRDebug: Automated Design Rule Debugging

Irina Alam, Tianmu Li, Sean Brock, and Puneet Gupta

***Abstract*—Design Rule Checking (DRC) is an important step in the physical design flow that checks if a design meets the manufacturing constraints or design rules imposed by the process technology. It allows the foundry to ensure high acceptable manufacturing yield. Design rule verification is one of the most challenging steps because of the sheer size of the rule decks and the lack of standardization among these design rule manuals. One way of efficiently discovering missed rule checks is by comparing the rule deck with that of another mature or well-established process. In this work, we develop two complementary techniques for comparing process design rule decks and automatically establishing a one-to-one correspondence between rules from two different Process Design Kits (PDKs). The first approach, Random Layout Generation (RLG), creates random layouts of different shapes and sizes. The generated layout is checked using both rule decks. The rules are then matched based on the violations generated. The second approach, based on rule language processing (RLP), matches rules based on the similarity between rule commands. Rules are directly matched based on the layer names and keywords present in the DRC commands. The two approaches are complementary and together they can correctly match more than 80% of the rules in two design rule manuals.**

***Index Terms*—Design Rule, Manufacturing, Random Layout Generation, Natural Language Processing, Machine Learning**

## I. Introduction

In circuit design, a design rule is meant to impose geometric constraints on the layout to ensure that the design can be manufactured with yield and functions as expected. Design rule checking (DRC) is an important step during design verification signoff along with other checks such as LVS (layout-versus-schematic), ERC (electrical rule check), etc. The design rules are developed by foundries based on the characteristics and capabilities of the particular process to realize design intent. These rules can have a significant impact on manufacturability and design characteristics and performance [1], [2]. Some of the basic design rule checks are shown in Figure 1.

Foundries need to create a unique set of design rules for every new process. Besides, with time, as the process matures, the design rules get updated accordingly. When creating or updating the design rule deck, the biggest challenge is to verify the design rule deck to ensure that the rules are implemented as desired and the rule updates get reflected correctly. This verification is cumbersome as the rule deck is a large body of software and often spans multiple fab acquisitions, programming standards, etc. It is one of the most complex and time consuming step when developing a new process [3]. As a result, there is ample room for missing important checks in the final deck, leading to significant yield manufacturing yield loss which often invokes costly cross-layer debug process. For example, most rule definitions are complex involving multiple layers, use multiple variables and create several derived layers. A minor mistake in the definition can easily go undetected when using standard unit tests. As technology scaling slows down, customized foundry technologies are going to become more common. These foundry technologies are usually derivatives of existing processes and contain similar rules. As a result, developing, debugging, and maintaining design rule decks has become a challenge.

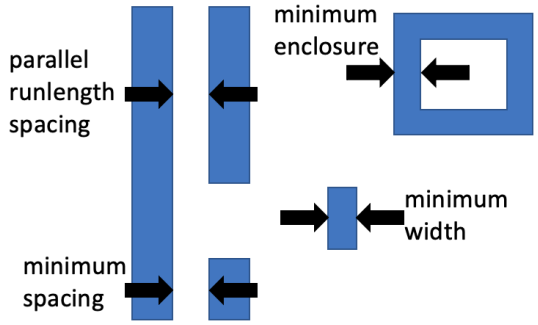


Figure 1: Some of the common design rule checks.

One efficient way of discovering missed rules or ensuring the changes in the rule definitions are as expected is by comparing the rule deck with that of an older process or an older rule deck version of the same process. A typical DRM can contain several thousand rules. We try to find one-to-one (or one-to-many) functional correspondence between rules from different DRMs. The first problem we try to address is that the same rule can be written in a variety of ways in the rule description language (e.g., SVRF), and establishing correspondence between equivalent rules is an important aid to debugging design rule decks. The second problem we try to address is establishing correspondence between two technologies that are independently or partly independently developed. This can happen, for example, when a foundry acquires an existing process technology from another foundry or when it creates a derivative technology for a different market niche, developed and maintained by a different team. The third issue we address is to identify inadvertent errors that can creep into different revisions of the same process (e.g., DRM v1.1 vs. DRM v1.2). Many of these changes under these revisions may not be minor and require a substantial rewrite of the design rule deck (e.g., for efficiency reasons, convention changes, etc). Nevertheless, no usable ”diff” of design rule decks exists, which makes the problem of finding and debugging design rule decks a tedious, challenging manual task. Regression tests for rules, usually manually designed, can only help for known rules. They cannot identify rules

I. Alam, T. Li and P. Gupta are with the Department of Electrical and Computer Engineering, University of California, Los Angeles.
E-mail: {irina1, litianmu1995, puneetg}@ucla.edu
S. Brock was with ON Semiconductor, Austin, TX.

**Spacing rule in FreePDK45**

```
L111 = SIZE metal1 BY 0.045 UNDEROVER
L112 = AND L111 metal1
L113 = COINCIDENT EDGE metal1 L112
L114 = LENGTH L113 >= 0.3
Metal1.5 {
@Minimum spacing of metal1 wider than 0.09 & longer than 0.3 = 0.09
EXTERNAL L114 < 0.09
}
```

**Spacing rule in FreePDK15**

```
RULE_M1008A{
@ RULE_M1008A
@ Minimum spacing of M1A when M1A is wider than 32nm and longer than 240nm is 68nm
M1A_wide = M1A WITH WIDTH > 0.032
d = INT M1A_wide <= 1 OPPOSITE
d_p = DFM PROPERTY M1A_wide d OVERLAP ABUT ALSO [EC1 = EC(d)] > 0.240
EXT d_p M1A < 0.068
}
```

Figure 2: The same spacing rule is expressed differently in two different design rule manuals.

that were missed or are there unnecessarily. Comparing rule decks across multiple processes is extremely challenging as there is no standard format of creating rule decks and they greatly vary between process nodes and/or foundries. Because of the lack of standardization, every foundry expresses design rules differently. For example, as shown in Figure 2, both rules are spacing rules and specifies the minimum distance between adjacent shapes in metal1 that are wider than X and longer than Y (X and Y values are different for different technology nodes). But the rules are expressed differently in the two design rule manuals. This makes the comparison challenging. In fact, comparing two rule deck versions of the same process can also be challenging if the number of changes between the two versions is large as it has to be ensured that all the differences are correctly highlighted so that they can be verified.

In order to efficiently compare two design rule decks and establish a one-to-one correspondence between rules from two different Process Design Kits (PDKs), we developed two complementary techniques in DRDebug. The first approach, Random Layout Generation (RLG), creates polygons of different shapes and sizes and places them on a grided layout. The generated layout is passed through design rule check using both rule decks and the rules are matched intelligently based on the locations of the violations generated. The second approach, based on rule language processing (RLP), matches rules based on the similarity of the rules commands being used. The first approach is agnostic of the format and syntax of rule commands and performs exceptionally well even when the rules in the PDKs are expressed very differently. The second approach, on the other hand, does not require appropriate layout generation that would violate all target rules in order to be able to correctly match them and hence, is best suited for density/coverage rules and rules that span across multiple layers. Given the complementary nature of the two techniques, they combined can correctly match more than 80% of rules in two different commercial design rule manuals. The techniques are significantly effective (upto more than 70% correct matches) even when the design rule manuals are for two different technology nodes and from two different foundries.

## II. Related Work

DRC rules are verified by creating test patterns for each design rule and verifying the output of the rules for those patterns. It is tested whether the passing and the failing structures are as per expectation. The process of generating a complete set of test structures for each design rule deck is difficult and time consuming. There has been past work on improving and automating design rule evaluation and verification. In [3], the authors enhance the PCELL Verification method [4] by using a parameterized pattern generation methodology. They reduce the number of test patterns generated for each design rule category by limiting the parameters that are varied across the different shaped polygons to create the final set of test structures. While they reduce the time and complexity of the design rule verification process, the proposed technique does not help to catch any critical rules that might be missing from the rule deck. Similarly, in [5], the authors do a line-by-line analysis to check if every component in a design rule has a corresponding test pattern in the regression suite. Doing so helps to make sure every part of a design rule that already exists in the manual is being verified. But, unlike DRDebug, it does not help to figure out if any necessary checks have been left out in the manual. Some other past works [6], [7] have looked into early design rule evaluation before exact process and design technologies are known. While the proposed framework systematically explores area-manufacturability-variability tradeoffs in design rules, it does not debug and verify design rule decks.

## III. Methodology

### A. Random Layout Generation

The first approach we use for design rule matching is random layout generation (RLG). The overall idea is to generate a single layout with geometries of different sizes and shapes in each layer. This generated layout would be checked using rule decks from both PDKs. The violations corresponding to same or similar rules between the two decks are expected to be in the same locations in the layout. For example, if both rule decks have the exact same minimum width criteria for each layer, the list of violation locations highlighting all shapes with width smaller than the minimum width would be identical in both cases. No other rule is expected to have the exact same list of violation locations and, therefore, using this, the two rules in the two PDKs can be matched with each other. However, in a real scenario, two rule decks are never identical, making the matching process more complicated.

In the following sections we will discuss the challenges we faced when matching violation locations using RLG and the corresponding solutions to these problems. Finally we summarize the flow in Section III-A5

*1) Generating polygons of different shapes and dimensions:* We used C++ based OpenAccess to create shapes in each layer of the generated layout. We then used Mentor Graphics tool Calibre to perform design rule check using the two target design rule manuals (DRM). Generating completely random polygons with random number of edges and corners results in illegal shapes that end up failing only a certain subset of rules

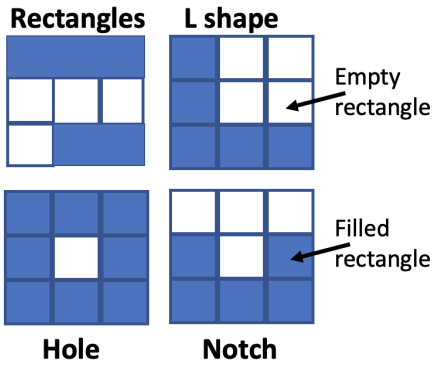


Figure 3: Four different shapes generated using 3x3(=9) tiles. Each of the 9 rectangular tiles are either filled or empty depending on the shape being constructed.

in the manual. In order to ensure that all rules in a particular layer get violated, a huge variety of polygons of different shapes and sizes would have to be created that would make the whole RLG flow unscalable. As a result, we introduced some structure to the random layout generation methodology. We first created a scalable shape library. Each polygon in the library is generated using a tile based approach. Each shape consists of NxN rectangular tiles as shown in Figure 3. Each tile is of different dimension. Based on the polygon shape that we are constructing, each tile is either filled or empty. For this work, we generated each shape using 3x3(=9) rectangular tiles. The shape library supports polygons of certain shapes and can be easily extended to add more shapes in the future. There is also an option of generating a random shape in the library by randomly choosing which of the 9 tiles will be filled.

Using the shape library ensures that polygons are generated somewhat systematically in the layout and hence, helps to dramatically reduce the number of polygons required per layer to violate all rules in the DRM. The next challenge in polygon creation is the dimension of each polygon. For each shape in the library, tile dimensions follow a random normal distribution. We created a tri-modal distribution of shape dimensions for creating small, medium and large sized shapes in order to limit the number of shapes required. The shape dimensions for each layer are chosen based on a mature technology and scaled up or down appropriately. Besides, the width and length distributions of the polygons are defined separately. This helps to better control the polygon types, e.g., ensure most shapes are rectangular than square while using limited number of shapes. Thus, using an easily extendable shape library supporting polygons of different shapes and a tri-modal distribution of grid length and width helps to limit the total number of shapes required to be generated per layer to violate all rules in the DRM.

*2) Matching PDKs for two different technology nodes:* Foundries often want to establish a one-to-one correspondence between rules from a newer technology node against rules from a mature, older technology node. When comparing rules from design rule manuals (DRM) of two different technology nodes, the rule dimensions are expected to vary. Thus, even if the rule definition is exactly the same, the violation list for the deck with stricter dimensions would be a superset of that of the other rule deck. For example, the minimum wire width required in metal layer 1 (M1) is 70nm in rule deck 1 and 45nm in rule deck 2. Therefore, the shapes failing minimum width in deck 1 would be a superset of the shapes failing the same rule in deck 2. If we do an exact matching of the violation locations, the rules will not be matched.

As a result, the generated layouts have to be appropriately scaled using scaling factors. When comparing two different technology nodes (X nm node vs. Y nm. node, where X>Y), we use a range of scaling factors. The scaling factors range from $\frac{X}{Y} - 1$ to $\frac{X}{Y} + 1$ in increments of 0.5 or 0.1. All shape dimensions and locations in the layout generated for X nm node is scaled down by the scaling factor when creating the same layout for Y nm node. The scaled down layout is checked against the DRM of Y nm node. The list of violation locations generated after the design rule check (DRC) are scaled back up before comparing them against the DRC violations of X nm node. This is done at each scaling factor. The reason behind considering multiple scaling factors is that different rule dimensions scale by different factors between the two DRMs. Hence, a single scaling factor is not expected to work across all the rules in the manual. After running the same flow for each scaling factor, every set of matches are considered across all scaling factors and ultimately, the best matching pair or the pair of rules with the maximum number of matching violation locations is declared as a match.

*3) Same rules with different rule commands:* The other challenge that we faced in RLG is that the same rule is defined differently. As a result, the violation markers generated after design rule check may highlight the same shape, but the exact violation polygon generated would be different. One such example is a parallel runlength spacing rule. This rule typically checks for the minimum spacing between two geometries on the same layer that run parallel for length L. The minimum width required is different for different ranges of L. When a pair of geometries violate this rule, the violation marker either highlights the two geometries violating this rule or it highlights the spacing between these two geometries. The violation polygon depends on how the rule has been exactly defined. Examples of two different definitions of such a rule is provided in Figure 2 and an example of showing the two different violation marker outcomes is shown in Figure 4. While both rules are identical, the difference in the way the rule command is written changes the violation polygon shape and location and hence, they cannot be matched.

In order to tackle this problem, we changed our layout generation technique to a grid based approach. The entire layout is divided into PxQ grid segments. $Length\ of\ grid\ segment = N \times (max.\ length\ of\ tile)$. Same is true for width of grid segment. Here, tile refers to each of the NxN rectangles that are used to construct a shape (as shown in Figure 3). Therefore, $P/Q = \frac{Total\ length/width\ of\ layout}{Length/width\ of\ each\ grid\ segment}$. Every shape that is created in the layout is placed at a randomly selected grid intersection point that is unused (a layout snippet is shown on the left in Figure 5). In this work, P=30 and Q=10 was used for each layer. All rules in the DRM that a particular shape violates would fall in the same grid segment. Therefore,

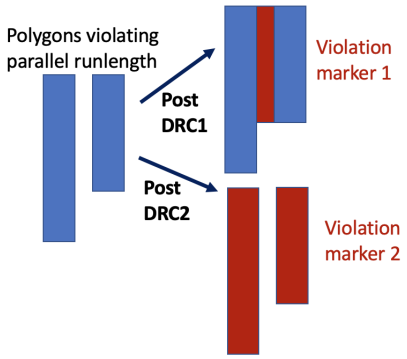


Figure 4: Example showing how the same polygons violating the same parallel runlength rule ends up with different violation markers post design rule check. The violation markers/polygons are shown in red. This happens because the same rule was defined differently in the two design rule files.

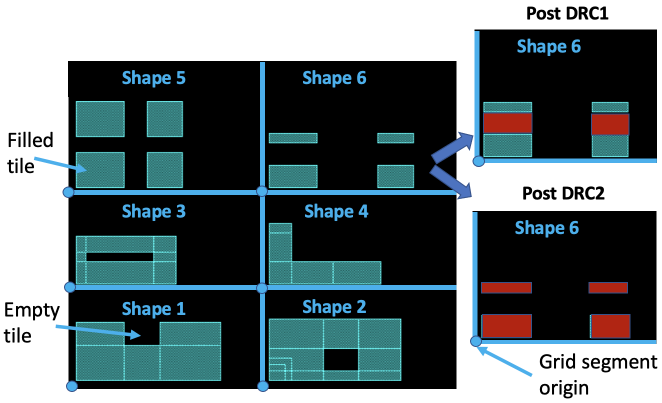


Figure 5: A small snippet of an actual generated layout (6 grid segments out of 30x10 [P=30, Q=10] segments) is shown. Each grid segment has a unique shape or polygon placed inside it. Each polygon is constructed using 3x3(=9) tiles where some tiles are filled and the rest are empty depending on the shape of the polygon. On the right, Shape 6 violates the same rule in two different DRMs. But the violation markers, in red, are placed differently. To catch such cases, we match based on grid segment origin inside which the violation marker(s) reside instead of actual violation marker locations.

even if the violation marker locations and shapes are not the same, as is shown for Shape 6 in Figure 5, if two rules violate the same set of polygons (situated in the same grid position), they would be considered a match. Thus, in the examples in Figures 4 and 5, the two rules would be considered a match since the violation markers fall within the same grid segment. The matching is done based on the grid segment origin coordinates inside which the violation marker(s) reside.

The downside of this approach is that multiple different rules might be violated in the same grid position (for example minimum width and minimum spacing) and they would be indistinguishable on given grid location. However, a particular rule is expected to get violated in multiple different grid locations. The probability of the same set of rules violating in all of the exact same grid locations is extremely low. Besides, to further negate the impact of this, we use an autoencoder based approach to get rid of violating grid locations that have too many rule violations and hence, might not provide any useful information. We explain this next.

*4) Using Autoencoders to extract useful information from the violation list:* Once the design rule check is done using all rules in the respective DRMs for the target layers, a binary results vector is generated for each rule before conducting similarity check. The length of the vector is equal to the total number of unique violation locations after the design rule checks combined across both sets of DRMs. Each bit in the vector represents a particular grid location that has at least one violation. If a particular rule gets violated in a certain location, the bit corresponding to that location in the result vector of that rule is equal to '1'. The results vectors for all violating rules are constructed accordingly. Each violating rule in DRM1 is compared against every violating rule in DRM2 by computing the cosine distance between the two vectors. For two non-zero result vectors $v_1$ and $v_2$, their cosine distance is calculated as $1 - (v_1 \cdot v_2)/(||v_1||||v_2||)$, where a smaller distance means higher similarity and a distance of 0 means completely matching. The rule(s) in DRM2 with the minimum cosine distance that is below a certain fixed threshold value is considered a match with the target rule in DRM1.

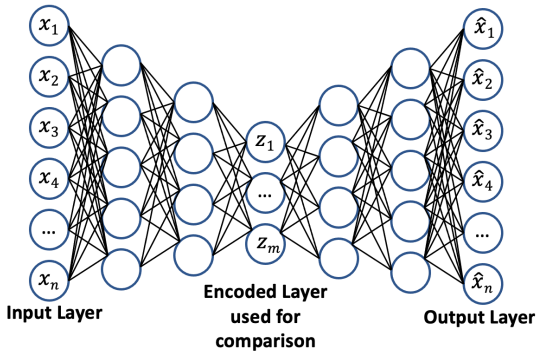


Figure 6: Variational autoencoder used in RLG and RLP.

While some shapes in the layout violate a single or a few rules in the DRM, there will be some random shapes that would violate a lot of rules. The bit positions corresponding to these locations in the results vector would be '1' for a large number of rules. As a result, these locations only increase the results vector length and add to the computation complexity when matching violations without providing much useful information. In order to remove such violation locations, denoise data and reduce dimensionality before matching, we use autoencoder [8]. Autoencoder is an unsupervised neural network that compresses the data to lower dimension and and then reconstructs the input back. Compared to traditional dimensionality reduction techniques like principal component analysis (PCA), multi-layer autoencoders allow non-linear mapping between the original and encoded space and achieve better reduction performance. Autoencoders are trained so that the difference between input and output vectors is minimized, thereby maximally preserving the input information given a small encoded vector. Since our approach aims to minimize

manual efforts during the matching process, autoencoder is a good choice to reduce dimensionality of data without knowing the matching information. Autoencoders can be of different types. Since in our case we do not have any spatial information to extract from the generated violations, we use a multi-layer fully connected variational autoencoder as shown in Figure 6. The autoencoder is trained every time in the flow using input vectors corresponding to each violating rule. If there are 'n' unique violations in the flow and there are 5 iterations, each input vector is of length '5n' and the output vector is also of the same length. Post autoencoder training, every input vector is passed through the first stages of the autoencoder to obtain the reduced vector z. In our experiments, we used a 30x10 grid size. Thus, the maximum number of violating locations could be 300. We also generated 10 layouts for each set of layers (explained in the next section). Thus, the maximum number of unique violating locations possible is 3000. Every layer is halved and the final latent space, z, is 8x compressed. The final cosine similarity is computed using the compressed z vectors. Using a simple filtering based approach to simply remove violating grid locations that have a large number of violations comes with a much higher probability of getting rid of useful information that some of these locations might have.

*5) Overall Flow:* The overall random layout generation flow is shown in Figure 7. The overall runtime is dominated by the layout generation and the design rule check part of the flow. For a set of three layers (Metal1-Via1-Metal2), if there are 10 layouts generated (10 iterations of steps 2-6 in the flow), the overall runtime for a pair of commercial PDKs with ∼15-20 rules per layer is ∼15-20 minutes. If larger number of layers are compared together, this runtime increases significantly. For a set of 7 layers, the runtime for the same number of iterations increases to almost an hour. Hence, we limit RLG to checking upto 4 layers simultaneously. Per iteration, per layer, the number of grid segments was 30x10 (PxQ). At each grid location, a 3x3 (N=3) tile-based shape is generated and placed. The total size of the layout depends on the maximum length and width of each rectangular tile used to generate the shape. These dimensions, in turn, depend on the technology node and the target layer(s). For example, for the 45nm DRM, the total size of the layout generated per iteration was $\sim 1.2mm \times 1mm$ as the maximum width and length of each of the rectangular tile was set at $13\mu m$ and $33\mu m$ respectively. For each set of layers, steps 3-6 in Figure 7 are repeated 10 times. Based on commercial DRMs, the number of rules per layer roughly varies between 40-80. Thus, DRC per iteration is performed on ∼40-320 rules per DRM depending on the number of target layers.

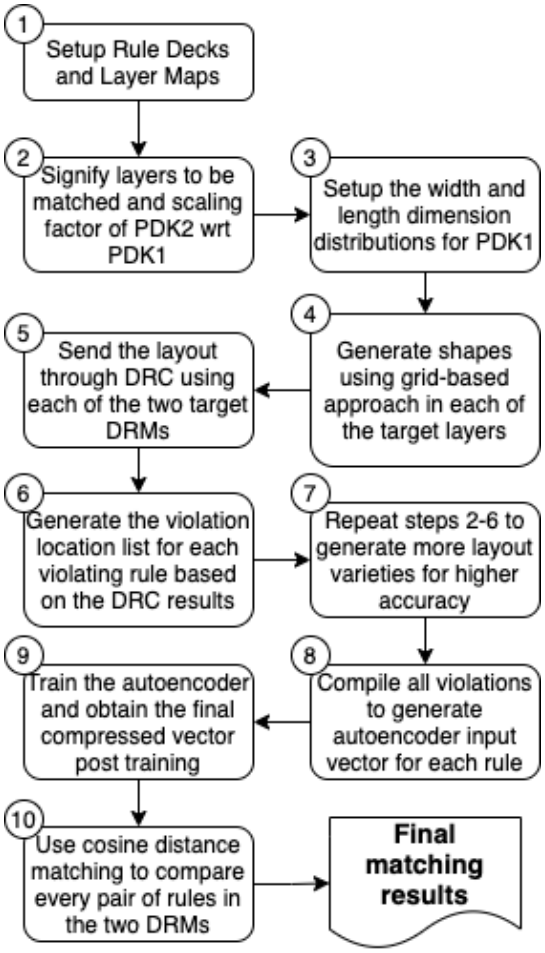


Figure 7: Overall workflow of random layout generation (RLG) based design rule matching. This is shown for a single scaling factor. If multiple scaling factors are used for the same set of layers, repeat steps 1-9 for each scaling factor and perform cosine distance matching (step 10) on the final combined result across all scaling factors.

### *B. Rule Language Processing Based Approach*

While random layout generation allows matching rules based on violations each rule identifies on generated layouts, it is not perfect. Some rules may cause no violation with the generated layouts. E.g.: density rules may never be violated if the generation algorithm does not generate layouts with very high densities. As a result, some other means of layout matching is necessary. This leads us to rule language processing (RLP). Even without parsing the rule files through a DRC tool and relying on rule violations, the individual rules written in DRC tool language (for e.g. Standard Verification Rule Format or SVRF) and the accompanying descriptions should include enough information for rule matching. Figure 2 shows examples rules written in the SVRF format. The description of a rule (beginning with @) is typically written in plain English and can be processed using traditional natural language processing (NLP) [9] techniques. However, the rule descriptions are usually not standardized. So training of the language embedding and/or classifier is needed for every set of PDKs that we want to compare in order to achieve good performance. However, these PDKs do not contain enough rule descriptions to train an NLP classifier with good generalizability. Besides, labeling matching pairs manually for training the network defeats the purpose of an automatic design rule matching tool. In our experiments, matching rules based on SVRF keywords has $> 10\%$ higher matching accuracy compared to NLP based on rule descriptions. Hence, we use keyword

matching approach in this work and we describe it in detail in the subsequent subsections.

*1) Data processing:* For the rest of the paper, we use SVRF as a placeholder for any DRC tool language. The SVRF keywords are first embedded into one-hot vectors using a dictionary for ease of processing, with each location in the vector representing a unique SVRF keyword. The SVRF keywords can be divided into two types, inherent SVRF commands and others. The inherent SVRF commands are operators on variables, and the number of unique commands is the same regardless of the PDKs being compared. As a result, the dictionary used for embedding initially contains a fixed-length portion containing all inherent SVRF commands, and equivalent commands are mapped to the same vector. An example of this is the "EXTERNAL" and "EXT" commands, which are different versions of the same command. Other SVRF keywords include layer names and values. Since they differ between different PDKs, the dictionary size varies depending on the PDKs being compared. To deal with this issue, we allow the dictionary to expand when a keyword that has not been seen before appears. For example, if the dictionary already contains $k$ keywords and a new keyword appears, it will be mapped to $\{(k$ 0's), 1$\}$. If the two PDKs being matched contain $n$ unique keywords and values other than the inherent SVRF commands, the total size of the of the dictionary becomes $n + n_0$, where $n_0$ is the number of inherent SVRF commands. Once all rules are processed in the two PDKs, all keyword embeddings shorter than $n + n_0$ are padded with 0's to the same $n+n_0$ length for easier processing. The embedding of a rule is then taken as the sum of the all the keyword embeddings in the rule.

*2) Initial matching with embedded data:* Once all rules are embedded, they're first grouped based on the layer they belong to, which can typically be found in the name of the rule. A rule belonging to a particular layer like metal 1 or via 1 will only be matched to rules in the same layer in the other PDK, therefore simplifying the matching procedure. Even though it's possible for a rule to include variables from different physical layers, like the enclosure of nearby via layers on a metal layer, all rules we've encountered are named based on the main layer it's concerned with, and no rule has been found to match with rules of another layer. Within the same layer, rules are matched by computing the cosine distance between pairs of rules, similar to the method in Sec. III-A. Pairs with cosine distance below a fixed threshold are considered matches. This allows finding multiple matches to the same rule, while also increasing the chance of false matches.

*3) Using Autoencoders to compress information:* The simple scheme mentioned above can find too many false matches or miss matching rules depending on the characteristic of the PDKs. Since variable names and SVRF commands are weighted equally, two rules can be considered matched if they differ in a single variable but use the same commands. Conversely, even if two rules achieve the same effect, they can be considered unmatched if one of them uses a different way of writing the rule. To compress the unnecessary parts of the vector and focus on the salient parts, we use a variational autoencoder (VAE) similar to the one in Figure 6 to compress

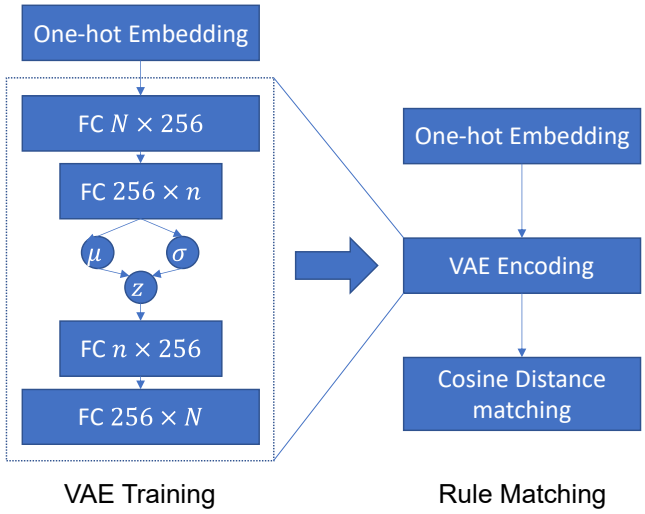


Figure 8: Overall workflow of rule language processing and the associated neural network architecture.

and then decompress the embeddings of all rules in the two PDKs. The autoencoder is trained to minimize the reconstruction loss, and the latent vector generated by the encoder is used for matching.

*4) Overall flow:* Figure 8 shows the overall flow of keyword-based rule matching. FC stands for fully-connected layers, and the two numbers following it represent the number of input and output neurons of each layer. $N$ is the number of keywords after processing all the rules from both PDKs, which equates to 3000-5000 depending on the PDKs being compared. $n$ is chosen to be 128 in all experiments, so the autoencoder achieves a compression ratio between 2% and 4%. $z = \mu + \sigma \odot \epsilon$ where $\epsilon \sim N(0, I)$. This reparameterization step is used to constrain the $z$ vector to a normal distribution. Doing so helps with classification after VAE training in our experiments. Most of the time is spent on the initial data embedding and training of the autoencoder model. Overall run time depends on the size of the PDKs, but is typically $< 5$ minutes when training the VAE on a 14-core CPU using PyTorch.

## IV. Results

For evaluating our approaches we compared the following sets of design rules manuals:

- Alpha and mature DRMs belonging to a commercial process design kit (PDK) for the same technology node (180nm)
- Two different commercial DRMs for the same technology node (180nm)
- Two commercial DRMs for different technology nodes (40/45nm vs. 65nm)

Unless specified otherwise, results are obtained after processing using Autoencoders, and the threshold used for matching is set to be 0.05 for both RLG and RLP. 800-1200 rules from 15-20 layers are used for matching from each DRM. While DRDebug can provide one-to-many correspondence between the rules, for the sake of simplicity, we use the top matching

pairs (pairs with the lowest distance below the set threshold) when evaluating accuracy.

### A. Overall Matching Accuracy - RLG and RLP

To evaluate Random Layout Generation technique, we compared multiple sets of design rule manuals as listed previously. Due to computation complexity, for RLG, we limited ourselves to rules that span across at most 4 layers. For each iteration of the flow, we first generated 10 different random layouts for the same set of layers that we are comparing. Each layout is sent through design rule check using the two sets of DRMs. The list of violations for each of the 10 layouts are compiled and the results vector is either generated (for the first layout) or appended to the already existing vector (for the remaining layouts). The final results vector obtained after running DRC on all 10 layouts is used for training the autoencoder. The encoded vector, post autoencoder training, is taken and the cosine distance is computed for each pair of vectors across the two target DRMs being compared. Lower the cosine distance, higher is the matching confidence. In Rule Language Processing (RLP), both DRMs are sent through the flow to match all rules across all layers. The final RLG and RLP results are provided in Table I. The layers are broadly classified into two groups: (1) metal/via layers, (2) non-metal/non-via (other) layers. The results are categorized as correct, false and missed matches. Given that there is no industrial standard on matching design rules, matches are determined by hand, where pairs of rules that are sufficiently similar are considered matches. False matches include the rules that were incorrectly matched while missed matches include rules that should have been matched but were not matched by either RLG or RLP. Our manual inspection was spot-checked by an expert design rule deck developer at our collaborating foundry. The missed matches in RLG happen when the generated layouts do not violate certain rules and hence, they cannot be matched. In RLP, missed matches happen when similar rules in two DRMs use different variable names or rule definitions.

*1) Alpha and mature DRMs for same technology node:* We first compared two versions (alpha and mature) of the DRM belonging to the same technology node from the same commercial PDK. As provided in Table I, for metal and via layers, we saw that 73.20% of the rules were violated. the remaining 26.8% of the missed matches include all density related rules. All violating rules matched correctly, with high confidence (more than 98% of the rules had cosine distance of zero). The generated layouts had polygons placed at every grid location, hence, none of the 10 generated layouts violated any of the density rules. Generating appropriate layout to violate density rules is extremely hard and needs to be done separately from the rest of rules. On the other hand, density rules are mostly defined similarly between design rule manuals and, hence, RLP based approach does exceptionally well in matching these rules, with only 8% missed matches. The missed matches in RLP typically differ in the values or constraints used for the rule. Since all keywords are treated equally, the algorithm sometimes gets confused when slightly different values are used in the two PDKs or when new constraints are added for the mature DRM. For the non-metal layers, RLG could only create shapes in four layers in a single iteration of the flow. Thus, during DRC, only rules spanning four or lesser layers could be violated. For matching more complex rules involving more than four layers, we use RLP based approach. For layers such as poly, active, n-well, n/p-implant and their derived layers, RLP correctly matched 82% of the violated rules. The remaining 18% of the rules had false matches, where two rules were incorrectly shown as matches. However, all of these matches had non-zero cosine distance. If only matches with cosine distance of zero is considered for these layers, then all false matches get filtered out. However, 17% of the correct matches also now get filtered out and get considered as rules that have no equivalent matches. If the cosine distance is set to a non-zero threshold value (for this pair of DRMs we considered a threshold value of 0.05), then all correct matches are retained while 80% of the false matches get filtered out.

*2) Two commercial DRMs for same technology node:* We then compared two different commercial design rule manuals (DRMs) for the same technology node. The results were similar. All metal and via rules matched correctly except the density related rules as the generated layouts did not violate any of the density rules. When a cosine distance threshold of 0.05 is set, all metal and via rules in DRM-1 that does not have matching rules in DRM-2 are correctly flagged as no corresponding matches are found. This shows that for the metal and via rules, RLG not only correctly points out the matching rules, it also highlights the rules that are missing in the decks, which is required during the verification process. For non-metal and non-via layers, efficacy of RLG is limited to rules that span across 4 or lesser layers. For such rules, the correct matches are flagged as high-confidence (low cosine distance) matches. Some of the false matches can be successfully eliminated by choosing the right cosine distance threshold. Without any threshold, $\sim$16% of the rules are incorrectly flagged as matches. With a threshold of 0.05, this significantly reduces to less than 2.5%. But, with the threshold set, $\sim$2% of correct matches now get flagged as rules that have no corresponding matches. This result will change depending on the threshold value that is set. For a threshold value of 0, the number of false matches become 0, while 6.7% of the correct matches get incorrectly tagged as rules with no matches. RLP performance in non-metal and non-via layers remain similar to the performance in Sec. IV-A1

*3) Two commercial DRMs for two different technology nodes:* For the final set of results, we wanted to check how the techniques perform when comparing DRMs for two different technology nodes. Hence, we used one 65nm commercial PDK and another 40nm LP commercial PDK. Since these two PDKs were for two different nodes, the layers, rule definitions and the rule dimensions were very different. As a result, matching the rules for each layer was more challenging. For RLG we first compared the metal and via layers. We used scaling factors from 1.25 to 1.65 in increments of 0.1. While the overall matching accuracy is 65.30% using RLG, the performance varies depending on the type of rule. All (four in total) density rules and five spacing rules related

Table I: Overall results when using Random Layout Generation and Rule Language Processing. Autoencoders are used.

| | **Random Layout Generation** | | | | | | **Rule Language Processing** | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | Alpha and Mature DRMs (180nm) | | Two commercial DRMs (180nm) | | Commercial DRMs (40/45nm vs. 65nm) | | Alpha and Mature DRMs (180nm) | | Two commercial DRMs (180nm) | | Commercial DRMs (40/45nm vs. 65nm) | |
| **Layers** | Metal & Via | Other | Metal & Via | Other | Metal & Via | Other | Metal & Via | Other | Metal & Via | Other | Metal & Via | Other |
| **Correct Matches** | 73.20% | 69.10% | 89.50% | 67.80% | 65.30% | 42.96% | 87.10% | 81.17% | 88.17% | 82.21% | 58.95% | 47.87% |
| **False Matches** | 0.00% | 3.60% | 0.00% | 4.70% | 14.70% | 17.04% | 5.38% | 1.95% | 6.45% | 3.02% | 0.00% | 0.00% |
| **Missed Matches** | 26.80% | 27.30% | 10.50% | 26.60% | 20.00% | 40.00% | 7.53% | 16.88% | 5.38% | 14.77% | 41.05% | 52.13% |

to nets connected to different voltage domains could not be matched as they did not generate any violations. For such rules, RLP performs better than RLG as actual layouts with different voltage domains and densities are not required in RLP. For the rest of the metal layer rules, across all scaling factors, 71.4% of the violating rules were matched correctly. The remaining 28.6% of the rules that resulted in false matches are parallel run-length spacing rules. Due to the difference in spacing and length dimensions between the two rule decks, it is challenging to correctly categorize these rules as correct and incorrect matches. If the exact dimensions are ignored and they are all broadly categorized under the same parallel run-length spacing rule bucket, then these 28.6% rules all match correctly. Since the rule decks are from the same foundry, the rules have similar names. If they are matched by names, then 2 out of 6 rules match correctly and the rest end up with false matches. For the via layers, 1 out of 18 rules per layer did not generate a violation as it checks for connections between via layer and neighboring metal dummy layers. The number of violating rules show that the current pattern library used in RLG is comprehensive as it manages to violate most of the rules. On an average, across all via layers, 81.25% of the violating rules match correctly. The remaining rules are spacing rules which, similar to the metal layers, match with other spacing rules having different rule names. So they can be classified as correct matches if all spacing rules are categorized under the same bucket, and incorrect if matched using the exact rule names. We then compared poly, implant, well and contact layers. In spite of the rule dimension mismatches between the two technology nodes, these layers had an accuracy of 71.6%. However, the fraction of violating rules in these layer is lower than that of metal and via layers. On an average, across all the non-metal/non-via layers tested, $\sim$60% of the rules can be successfully violated using RLG. The ones that do not get violated are either density rules, rules for different voltage domains or rules interacting across multiple layers. These rules are matched using RLP. Table II compares the performance of RLG and RLP across different layer types for these two commercial DRMs. RLP is able to match some rules that cannot be violated using RLG, but RLG achieves higher accuracy for violating rules.

Table II: Accuracy breakdown for commercial DRMs (40/45nm vs 65nm).

| | **Non-violating Rules** | **Violating Rules** |
|---|---|---|
| | Metal Layers | |
| **RLG** | 0.00% | 71.43% |
| **RLP** | 44.44% | 64.26% |
| | Via Layers | |
| **RLG** | 0.00% | 81.25% |
| **RLP** | 0.00% | 42.86% |
| | Other Layers | |
| **RLG** | 0.00% | 71.60% |
| **RLP** | 48.53% | 46.15% |

### *B. Interesting Result Highlights: RLG vs. RLP*

The two proposed techniques are complementary in nature. RLP helps to highlight subtle changes in rule definitions which might have a big impact on the correctness of the rule deck. When comparing the alpha and mature versions of rule decks belonging to the same node we found instances where the alpha version had values (for instance min. width dimension) assigned to variables and these variables were used in the rule definitions. In the mature version, the variables were still there but they were not used in the rule definition. Instead the values were hard-coded in the definition. This can lead to potential problems if the engineer changes the value assigned to the variable only. The change will not get reflected in the final rule check. RLP helped to highlight these subtle differences that do not get caught when using RLG.

RLG, on the other hand, helps to establish correspondence in the case of complex rules where the rule definitions might widely vary between the two design rule manuals. For such rules, RLP fails to match them but RLG correctly points out the correct matches since it does not depend on the exact SVRF rule definition. This is especially true when comparing rules from DRMs belonging to two different foundries. However, RLG requires layout generations that would result in rule violations. If rules do not get violated, RLG will not be able to match them. RLG also requires generating grids that can represent the rule. Spatially expansive rules, either spanning multiple layers or covering a large area, demand correspondingly large grids, which hurts matching performance or takes more time to generate the grids and perform DRC checks. RLP does not require that and hence, works better for density rules and rules involving multiple layers (more than 4/5 layers).

To combine the benefits of both approaches, we set a rule where a pair of rules is considered matched if either RLG or RLP consider it to be matched. The results are shown in Tab. III. Combining the two approaches improves the average percentage of correct matches by 11.6%. Even with the combined approach, matching accuracy is generally lower for the two commercial DRMs for different technologies, but we believe the missed and false matches are also valuable. The missed matches highlight the differences between the two DRMs that may need further examination. The false matches highlight pairs of rules that should not be matched but appear similar in terms of function or syntax. The accuracy is also limited by the training data available, since DRDebug only uses data from the two DRMs being compared. Better

generalization performance is possible by first training a larger autoencoder model using violation patterns and matching pairs from other similar DRMs, and then only fine-tuning the model with the target DRMs.

Table III: Combined results using both RLG and RLP.

| | Alpha and Mature DRMs (180nm) | | Two commercial DRMs (180nm) | | Commercial DRMs (40/45nm vs. 65nm) | |
|---|---|---|---|---|---|---|
| **Layers** | Metal & Via | Other | Metal & Via | Other | Metal & Via | Other |
| **Correct Matches** | 88.17% | 82.32% | 88.24% | 83.39% | 77.32% | 57.84% |
| **False Matches** | 5.38% | 2.89% | 6.95% | 3.99% | 2.06% | 5.88% |
| **Missed Matches** | 6.45% | 14.79% | 4.81% | 12.62% | 20.62% | 36.27% |

### C. Benefit of Using Autoencoders

Autoencoders are used both for RLG and RLP to improve matching performance. Autoencoders are trained from scratch for different pairs of PDKs being compared, since different PDKs can have different violation patterns and styles of SVRF rules. Given the small size of the models used, training time of the autoencoders constitute a small portion of the the runtime for both RLG and RLP even when training on a CPU. The performance with and without the autoencoder is compared in Table IV. We compared the two commercial DRMs for different technology nodes to measure the benefit of autoencoders since we found this to be the most difficult scenario because of significant differences in rule definitions and rule dimensions. For RLG, using autoencoders improves matching accuracy by 11.7% points. Also, the cosine distance difference between correct and false matches increases significantly. As a result, it becomes easier to set a distance threshold that helps to separate false and correct matches. For RLP, using autoencoders improves accuracy by 10.8% points. Most of the additional matches from using autoencoders are width rules with parallel length constraints. With an autoencoder, some of the unnecessary value and layer name differences are compressed, and thus more matches can be achieved. Compared to using a more traditional dimensionality reduction technique like principal component analysis (PCA), using an Autoencoder improves matching performance by 5.18 percentage points for RLG and 1.4 percentage points for RLP.

Table IV: Accuracy comparison between matching with and without autoencoders. The results are for matching DRMs of two different technology nodes (40/45nm vs 65nm) and using the aggregated results of all layers.

| | RLG | RLP |
|---|---|---|
| Without autoencoder | 43.7% | 42.6% |
| With PCA | 50.2% | 52.0% |
| With autoencoder | 55.38% | 53.4% |

## V. Conclusion

Design Rule Checking (DRC) is an important step in the physical design flow. Creating and verifying the design rule deck is one of the most challenging tasks during technology development. In this work, we develop two complementary techniques for comparing process design rule decks and automatically establishing a one-to-one correspondence between rules from two different Process Design Kits (PDKs). Doing so helps to verify the rule decks and can catch any missing rules that can lead to yield loss. The first approach, Random Layout Generation (RLG), generates a set of random layouts and sends them through design rule check using both rule decks. The violation locations are matched intelligently to generate the final list of matched rules. The second approach, based on rule language processing (RLP), matches rules based on rule commands. While RLG is agnostic of the format and syntax of rule commands unlike RLP, it heavily depends on the polygon dimensions and shapes. RLP, on the other hand, is best suited for density/coverage rules and rules that span across multiple layers as it does not require running design rule check and is not limited by the generated layouts or incur high runtimes for rules spanning across multiple layers. The two techniques combined can correctly match more than 80% of the rules and does exceptionally well even for different technology nodes.

## References


[1] Y. Zhang, J. Cobb, A. Yang, J. Li, K. Lucas, and S. Sethi, "32nm design rule and process exploration flow," in *Photomask Technology 2008*, H. Kawahira and L. S. Zurbrick, Eds., vol. 7122, International Society for Optics and Photonics. SPIE, 2008, pp. 1250 – 1261. [Online]. Available: https://doi.org/10.1117/12.801593

[2] V. Joshi, B. Cline, D. Sylvester, D. Blaauw, and K. Agarwal, "Leakage power reduction using stress-enhanced layouts," in *2008 45th ACM/IEEE Design Automation Conference*, 2008, pp. 912–917.

[3] M. Tantawy, R. Guindi, M. Dessouky, and M. Al-Imam, "Parameterized test patterns methodology for layout design rule checking verification," in *2015 IEEE International Conference on Electronics, Circuits, and Systems (ICECS)*, 2015, pp. 588–591.

[4] survey results by Mentor Graphics consulting division using the data provided by 3 Asian fabs, "Process development and QA," Jan 2014.

[5] Y. Lee, J. Park, M. A. Elsayed, M. E. Gadallah, and M. Alimam, "Drc code coverage test a novel qa methodology," in *2018 International Conference on IC Design Technology (ICICDT)*, 2018, pp. 93–96.

[6] R. S. Ghaida and P. Gupta, "A framework for early and systematic evaluation of design rules," in *2009 IEEE/ACM International Conference on Computer-Aided Design - Digest of Technical Papers*, 2009, pp. 615–622.

[7] R. Ghaida and P. Gupta, "DRE: A Framework for Early Co-Evaluation of Design Rules, Technology Choices, and Layout Methodologies," in *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 31, no. 9, 2012, pp. 1379–1392.

[8] D. P. Kingma and M. Welling, "An introduction to variational autoencoders," *arXiv preprint arXiv:1906.02691*, 2019.

[9] G. G. Chowdhury, "Natural language processing," *Annual review of information science and technology*, vol. 37, no. 1, pp. 51–89, 2003.