Efficient Design Rule Checking Script Generation via Key Information Extraction

Binwu Zhu CUHK
Xinyun Zhang CUHK
Yibo Lin Peking Univ.
Bei Yu CUHK
Martin Wong CUHK

ABSTRACT

Design rule checking (DRC) is a critical step in integrated circuit design. DRC requires formatted scripts as the input to the design rule checker. However, these scripts are always generated manually in the foundry, and such a generation process is extremely inefficient, especially when encountering a large number of design rules. To mitigate this issue, we first propose a deep learning-based key information extractor to automatically identify the essential arguments of the scripts from rules. Then, a script translator is designed to organize the extracted arguments into executable DRC scripts. In addition, we incorporate three specific design rule generation techniques to improve the performance of our extractor. Experimental results demonstrate that our proposed method can significantly reduce the cost of script generation and show remarkable superiority over other baselines.

CCS CONCEPTS

• **Hardware → Methodologies for EDA**.

KEYWORDS

Design Rule Checking, Natural Language Processingm Key Information Extraction

ACM Reference Format:
Binwu Zhu, Xinyun Zhang, Yibo Lin, Bei Yu, and Martin Wong. 2022. Efficient Design Rule Checking Script Generation via Key Information Extraction . In *Proceedings of the 2022 ACM/IEEE Workshop on Machine Learning for CAD (MLCAD '22), September 12–13, 2022, Snowbird, UT, USA.* ACM, New York, NY, USA, 6 pages. https://doi.org/10.1145/3551901.3556494

1 INTRODUCTION

Design rule checking (DRC) is an important step in electronic design automation (EDA) flow to check whether the layout conforms to a set of design rules. Rules usually specify certain geometric and connectivity restrictions to ensure sufficient margins for variability in semiconductor manufacturing processes, so as to ensure the proper function and reliability of layout designs. As shown in Figure 1, the completed DRC process includes two phases. (1) Rule making: manufacturers first specify the essential design rules based on their manufacturing capability and then convert them into the executable DRC scripts manually, which is illustrated in the first phase of Figure 1. (2) Rule checking: these scripts are provided to the designer and will be input into a design rule checker, such as KLayout [1], to verify the correctness of the layout design.

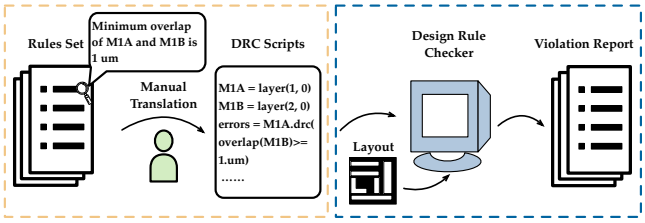

Figure 1: Entire design rule checking flow.

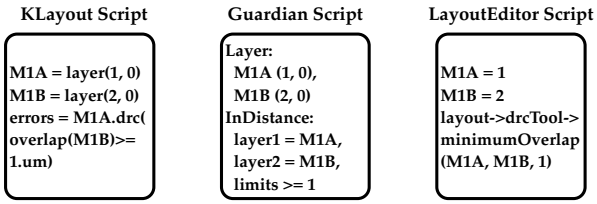

Figure 2: Script languages vary in different checkers.

Despite the importance of DRC, the whole process is always time-consuming and error-prone. There are mainly three aspects of reasons. Firstly, with the rapid development of semiconductor technology and the shrinking size of integrated circuits, the number of rules has grown from a few hundred in 65nm nodes to thousands of rules in 7nm nodes. Secondly, as shown in Figure 2, different checkers require different script languages, which means all scripts must be re-implemented when transferring to other checkers. Thirdly, some design rules can be very complicated, i.e., with complex logic, which may easily lead to misunderstanding.

In the last few years, advanced deep learning techniques have spawned many frameworks for effectively and efficiently solving EDA problems, including physical design [2–4], mask synthesis [5–7] and testing [8–10], etc. DRC, which demands highly efficient solutions, also greatly benefits from the development of deep learning. Due to the slow execution of the design rule checker, deep learning-based methods try to replace it with an artificial neural network, showing satisfactory efficiency and acceptable accuracy. For example, Tabrizi *et al.* [11] propose a neural network to extract features from a placed netlist and then detect detailed routing short violations. Islam *et al.* [12] develop the ensemble random forest algorithm to predict DRC violations before global routing, which is always the most time-consuming procedure in VLSI design flow.

These previous works for rule violations detection try to overcome the low-efficiency drawback of DRC by accelerating the rule checking process. But considering the real-world scenarios where industrial designs still rely on checkers along with the executable

MLCAD '22, September 12–13, 2022, Snowbird, UT, USA.

ACM ISBN 978-1-4503-9486-4/22/09…$15.00
https://doi.org/10.1145/3551901.3556494

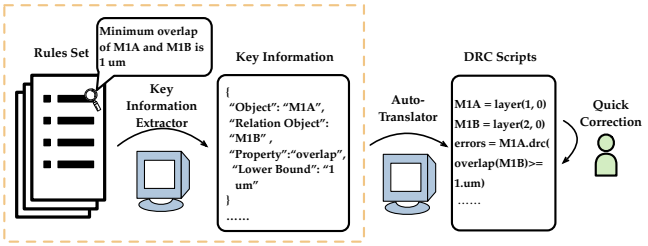

Figure 3: Our Proposed Automatic DRC Scripts Generation Flow. This is the optimization aiming at the rule making phase in Figure 1.

scripts, we consider optimizing the rule making process, which is also extremely slow when encountering a vast number of rules. Some research works [8–10] from the testing area try to adopt language models to generate assertion statements from specifications for functional verification of the fabricated chips. Inspired by these works, we consider that the DRC script generation process can also achieve automation to substantially ease the manual workload, achieving higher efficiency than the traditional manual flow, the first phase in Figure 1. Compared with testing specifications, design rules are more complicated and detailed, so they require well-designed methods to process.

We customize a brand new script generation flow depicted in Figure 3, which is totally different from natural language processing methods in [8–10]. In our flow, we design a deep learning-based key information extractor to automatically identify the essential arguments of the scripts, which can be regarded as a key information extraction process. The following script translator will organize the extracted arguments into the final scripts. Dividing the script generation flow into two stages makes it applicable to various checkers. The reason is as follows: The script translator in our flow conducts rule-based tasks, i.e., deciding which function to call and how to pass the extracted key information to the function. When switching to other checkers, we can keep the extractor unchanged and only need to modify the translator, which is convenient to implement once the script grammar is determined.

In addition, manual work is essential to correct potential errors in final scripts, since deep learning-based key information extractor in the generation flow can not guarantee absolute accurate results. But it should be noted that such manual correction work is much more relaxed in contrast to the traditional flow. The correction workload is mainly determined by the accuracy of the extracted key information, so the focus of our work is to design a powerful extractor.

To the best of our knowledge, our work is the first to investigate methods for automatic script generation to achieve DRC acceleration. The main contributions are listed as follows:

- We propose an efficient DRC script generation flow and design dedicated deep learning techniques based on the state-of-the-art natural language processing model to accurately extract key information from design rules.
- We develop data generation techniques based on the special language structures of design rules to expand the dataset, overcoming the dilemma of lacking design rule data for academic research.

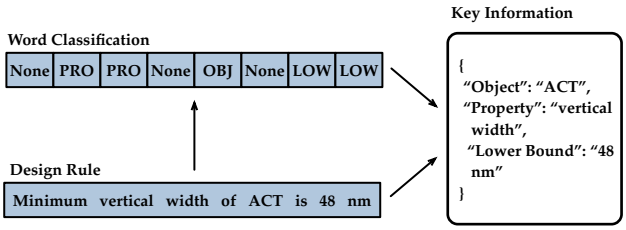

Figure 4: Key information extraction process (PRO means Property, OBJ means Object and LOW means Lower Bound).

- Experimental results on 7nm technology node demonstrate that our extractor achieves 85.3% precision and 88.0% recall on the key information extraction task.
- It only takes 5.46ms on average for our flow to generate the script of a single design rule.

2 PRELIMINARIES

2.1 Design Rule Key Information Extraction

Extracting key information from natural language design rules can be converted to such a problem that a specific word should be classified into a particular category, termed as a semantic role, such as the property to be checked or a specified minimum value. Hence, the problem can be considered as a word classification problem. We provide an example as shown in Figure 4 to illustrate the whole process and all specified semantic roles will be further explained in Section 3.1. After finishing the word classification task, the categories and related words can be paired and then stored into a data structure, which is exactly the key information extracted from design rules. The following script translator simply organizes the extracted information into the final scripts; therefore, the accuracy of the generated scripts mainly depends on the extractor performance.

To quantitively evaluate the performance of the extractor, we adopt three metrics, precision, recall, and F1 score. In our task, given a semantic role "S", words that actually belong to this category are represented as $\mathbb{G}$ and words predicted by our model as "S" are marked as $\mathbb{H}$. Then precision is defined as the ratio between $\#(\mathbb{G} \cap \mathbb{H})$ and $\#(\mathbb{H})$, and recall is calculated as the ratio between $\#(\mathbb{G} \cap \mathbb{H})$ and $\#(\mathbb{G})$, where $\#(\cdot)$ computes the number of collection elements. F1 score is the harmonic mean of precision and recall. An optimal key information extractor should get $\mathbb{H}$ close to $\mathbb{G}$ for each category and achieve high performance on all evaluation metrics.

2.2 Transformer and BERT

Recently, Transformer [13] has made much progress in sequence-to-sequence tasks [14–16]. Transformer consists of two parts, encoder and decoder. BERT [15] is one of the most famous models built with the Transformer Encoder and has been widely used as a backbone to extract features from sentences to solve many NLP problems such as Question Answering [17], Machine Translation [18], etc. To illustrate BERT [15], we first introduce the structure of Transformer Encoder.

As shown in Figure 5(a), Transformer Encoder consists of multiple layers, of which the most important one is the multi-head self-attention, allowing the model to attend to information at different positions globally [13]. In Transformer Encoder, given the input representation of a sequence $\{\boldsymbol{x}_1, \boldsymbol{x}_2, \ldots, \boldsymbol{x}_n\}$ and packed together as a matrix $\boldsymbol{X} \in \mathbb{R}^{n \times d_m}$, where d_m is the dimension for each element

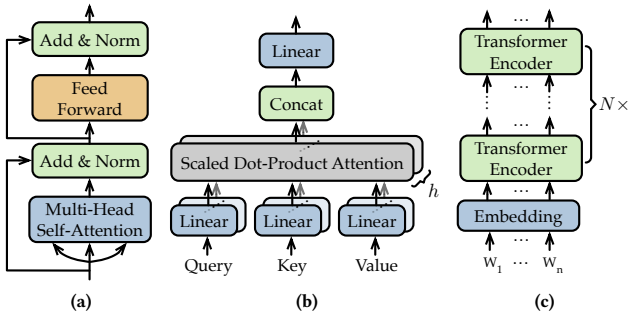

Figure 5: (a) Transformer Encoder; (b) Multi-Head Self-Attention; (c) BERT.

$\boldsymbol{x}_i$, the multi-head self-attention layer first projects the input matrix $\boldsymbol{X}$ onto three subspaces, which can be represented as:

$$\{\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}\} = \{\boldsymbol{X}\boldsymbol{W}^Q, \boldsymbol{X}\boldsymbol{W}^K, \boldsymbol{X}\boldsymbol{W}^V\}, \tag{1}$$

where $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ are called Query, Key and Value as named in Transformer [13], and $\boldsymbol{W}^Q, \boldsymbol{W}^K, \boldsymbol{W}^V \in \mathbb{R}^{d_m \times d_m}$. Then the output of multi-head self-attention layer can be formulated as:

$$\text{MultiHead}(\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}) = \text{Concat}\left(\boldsymbol{H}_1, \ldots, \boldsymbol{H}_h\right) \boldsymbol{W}^O, \tag{2}$$

where $\boldsymbol{H}_i, i \in \{1, 2, \ldots, h\}$ is the output of a single scaled dot-product attention head as shown in Figure 5(b) and h is the number of heads. To illustrate the dimension of $\boldsymbol{H}_i$ and $\boldsymbol{W}^O$, we first give the formulation of $\boldsymbol{H}_i$ as follows:

$$\begin{aligned} \boldsymbol{H}_i &= \text{Attention}\left(\boldsymbol{Q}\boldsymbol{W}_i^Q, \boldsymbol{K}\boldsymbol{W}_i^K, \boldsymbol{V}\boldsymbol{W}_i^V\right) \\ &= \text{softmax}\left[\boldsymbol{Q}\boldsymbol{W}_i^Q \left(\boldsymbol{K}\boldsymbol{W}_i^K\right)^\top / \sqrt{d_k}\right] \boldsymbol{V}\boldsymbol{W}_i^V. \end{aligned} \tag{3}$$

For each attention head, the original input $\boldsymbol{Q}, \boldsymbol{K}, \boldsymbol{V}$ are further projected onto different subspaces via projection matrices $\boldsymbol{W}_i^Q, \boldsymbol{W}_i^K \in \mathbb{R}^{d_m \times d_k}, \boldsymbol{W}_i^V \in \mathbb{R}^{d_m \times d_v}$ so that different heads deal with different input to learn richer information [13]. The attention head then computes the similarity between projected Query and Key via scaled dot-product and a softmax function is applied to obtain the weights on projected Value. As represented in Equation (2), the multi-head self-attention concatenates all the outputs $\boldsymbol{H}_i \in \mathbb{R}^{n \times d_v}, i \in \{1, 2, \ldots, h\}$ from different heads and then reduces the high dimension feature to low dimension via another matrix $\boldsymbol{W}^O \in \mathbb{R}^{hd_v \times d_m}$.

As for BERT, the architecture in Figure 5(c) is based on stacked Transformer Encoder blocks [13] and hence incorporates the superiority of multi-head self-attention. In addition, another significant advantage of BERT [15] is that it has been fully pretrained by two complex tasks, Cloze and Next Sentence Prediction. Since these two tasks do not require any manual annotations, the model can be trained on two huge datasets, BooksCorpus (800M words) [19] and English Wikipedia (2500M words). As a result, the pertrained model has been equipped with strong language representation ability and can be easily fine-tuned for other language tasks.

3 METHODOLOGIES

3.1 Semantic Roles

As illustrated in Section 2.1, extracting key information from design rules is inherently a word classification problem. In our task, categories of words are determined based on their semantic roles in sentences. Design rule data for training our deep learning-based model is from an open-source design kit, FreePDK15 [20]. To clearly classify different words, we first clarify all essential semantic roles for rules in FreePDK15 [20].

Some prior works for semantic role labeling studies [21, 22] have defined roles for universal natural languages. However, semantic roles to be considered are relatively different for rule sentences in EDA. For example, numerical expressions are less frequent in these studies and usually not attributed to a separate category. In contrast, they exist in most design rules and are core components of the extracted key information. Moreover, semantic roles of numerical expressions are supposed to be further divided into three categories, i.e., "Lower Bound", "Upper Bound" and "Exact Value", in order to flexibly adapt to different design rules and avoid confusion. We list all customized semantic roles for our task with their meanings and examples in Table 1.

3.2 Rule Data Generation

Open-source design rules for academic research are relatively rare. Our training dataset, FreePDK15 [20], only includes around 130 rules. To help the key information extractor avoid overfitting as well as generalize better on those unseen rule data, we are supposed to expand the dataset before training.

However, rule generation for our task is heavily restricted. On one hand, since the extractor receives the design rule sentences, we are supposed to guarantee that all generated rules are both syntactically and semantically correct. On the other hand, as our task is a classification task, semantic role labels need to be assigned to each word, which is extremely expensive. Considering these drawbacks, we propose three customized generation techniques in this section.

Word Order Adjustment. Inspired by the rotation technique for image data augmentation, we propose to change the word order of a rule without modifying its meaning. For example, we can settle the conditional adverbial clauses at the start or the end of the sentence. For the human, the reordered sentence can be regarded the same as the original one. However, from the perspective of the extractor, the input rule is a sequence $[w_1, w_2, \ldots, w_n]$ where w_i stands for a word. If the word order is changed, the input will be totally different. Moreover, adjusting the order will not affect the semantic role labels of words, and thus no extra annotations need to be done.

Paraphrasing. In contrast to Word Order Adjustment, paraphrasing can produce a new rule that does not change the meaning but has different expressions. To paraphrase design rules, we can replace some words with synonyms or change the sentence structure, e.g., from passive to active voice. Although paraphrasing will modify some words, which requires extra annotations, there are still many words not replaced, whose semantic roles also stay unchanged. Hence, the annotation workload can be reduced remarkably.

Template Filling. The generated design rules from the previous two methods are still confined to the meaning of original ones, making it challenging to generate enough training data. To take a

Table 1: Explanations of all semantic roles defined in our work. The bold parts belong to the roles defined in their rows.

Semantic Roles	Meanings	Examples
Object	Target layer of checking rules.	Minimum vertical width of **ACT** is 48 nm.
Relation Object	Additional layer that have relationships with target layer	Minimum extension of GATEAB past **ACT** is 38 nm.
Property	Property to be checked of the target layer.	Minimum **vertical width** of ACT is 48 nm.
Condition	Logical conditions for particular layers.	GATEC shape bottom or top must be aligned if **distance is less than 192 nm**.
Restriction	Geometric restrictions that layers should follow.	GIL may **not bend**.
Lower Bound	Minimum value of the property to be checked.	Minimum vertical width of ACT is **48** nm.
Upper Bound	Maximum value of the property to be checked.	Maximum distance of GATEAB to neighboring shape is **236** nm.
Exact Value	Exact value of the property to be checked.	Exact horizontal spacing of ACT is **80** nm.

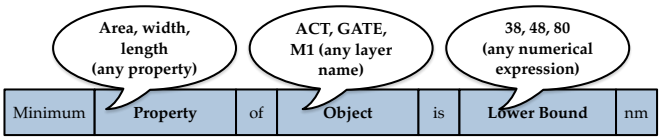

Figure 6: Design Rule Template.

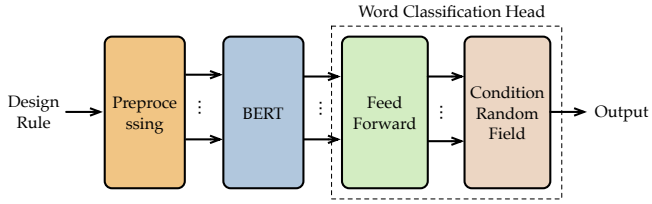

Figure 7: Key Information Extractor.

further step, we propose the third method, Template Filling. After applying the previous two generation methods, we can obtain a series of design rules with diverse sentence structures, from which multiple templates with different structures can be acquired. A typical template is given as shown in Figure 6, where the bold parts are semantic roles specified in Section 3.1. With such a template, any suitable words associated with these prescribed roles, whether from previous rules or not, can be filled. By filling in the designed templates via different combinations, a large amount of design rules can be collected for training, which benefits the generalization ability of the extractor dramatically. More importantly, we do not need to annotate the data manually since the semantic roles have been specified in advance.

3.3 Key Information Extractor

To classify all words from design rules into their corresponding semantic roles, we build up a deep learning-based language model. The overall architecture of our framework is illustrated in Figure 7.

Input Preprocessing Module. Before feeding the design rules into our extractor, some preprocessing operations need to be conducted. The first one is to split the rule into a list of words for the later word classification task. Besides, since different rules vary in sentence length, we extend the word list length to L by padding a special word "[PAD]". The whole procedure is formulated as:

$$[w_1, w_2, \ldots, w_{len(\boldsymbol{r})}, \underbrace{[\text{PAD}], \ldots, [\text{PAD}]}_{L-len(\boldsymbol{r})}] = \text{Preprocess}(\boldsymbol{r}), \quad (4)$$

where $\boldsymbol{r}$ is the input design rule. $w_i, i \in \{1, 2, \ldots, len(\boldsymbol{r})\}$ represents each word of $\boldsymbol{r}$ and $len(\boldsymbol{r})$ is its sentence length.

Backbone. Following the design paradigm of the deep learning model, we first need a backbone module to obtain a good feature representation from the input information. Determining the semantic role of each word is closely related to its sentence, and one word may have different semantic roles in different rules like "ACT" in the first and second example in Table 1. Therefore, the backbone should have strong abilities to capture the context information.

Instead of designing a backbone from scratch, we adopt a powerful language model, BERT [15], as the feature extractor, which proves to have prominent feature extraction ability according to many works like [17, 18]. As explained in Section 2.2, based on the self-attention mechanism, BERT is able to model interactions between any two different words in a sequence; therefore, the extracted feature of each word is closely correlated with the contexts. Besides, BERT has been fully pretrained, and thus we can fine-tune it from the pretrained parameters, which can notably speed up the training procedure.

Given the word list after preprocessing, the backbone will first encode words into vectors and then feed them into stacked Transformer Encoder layers. The output feature is represented as $\boldsymbol{F^o} \in \mathbb{R}^{L \times d_b}$, where d_b is the dimension of the extracted feature of each word.

Word Classification Head. With the purpose of classifying each word, we feed $\boldsymbol{F^o}$ into a word classifier, which is a simple feed-forward neural network composed of two fully connected layers. The output is represented as $\boldsymbol{P^{wc}} \in \mathbb{R}^{L \times N_{wc}}$, where N_{wc} is the number of categories, and the element $\boldsymbol{P}^{wc}_{i,j}$ stands for the score of the word w_i belonging to label j.

However, such a prediction head does not take the relationships between different labels into consideration. We can further force the word classification head to effectively avoid those impossible prediction sequences. For example, according to the common natural language expression habits, "Relation Object" is impossible to directly follow "Object", since there must exist some conjunctions between them. As a result, the extractor performance can be improved by evaluating the rationality of the entire prediction sequence. To achieve this, we build a probability model, condition random field (CRF) [23], on top of the word classifier, whose parameters are a label transition matrix, represented as $\boldsymbol{K} \in \mathbb{R}^{(N_{wc}+2)\times(N_{wc}+2)}$. The element of the label transition matrix $\boldsymbol{K}_{i,j}$ describes the score of transitioning from label i to j. Two additional states included in $\boldsymbol{K}$ stand for the "start" and "end" of the sequence. In such case, given a

design rule $\boldsymbol{r}$, the probability of a prediction sequence $\boldsymbol{y}$ is calculated from softmax function as:

$$p(\boldsymbol{y}|\boldsymbol{r}) = \frac{\exp S(\boldsymbol{r}, \boldsymbol{y})}{\sum_{\hat{\boldsymbol{y}} \in Y_{\boldsymbol{r}}} \exp S(\boldsymbol{r}, \hat{\boldsymbol{y}})}, \tag{5}$$

where $Y_{\boldsymbol{r}}$ represents all prediction sequence results given the rule $\boldsymbol{r}$. $S(\boldsymbol{r}, \boldsymbol{y})$ is used to measure the score of prediction $\boldsymbol{y}$, which can be formulated as:

$$S(\boldsymbol{r}, \boldsymbol{y}) = (\boldsymbol{K}_{start, \boldsymbol{y}_1} + \sum_{i=1}^{L-1} \boldsymbol{K}_{\boldsymbol{y}_i, \boldsymbol{y}_{i+1}} + \boldsymbol{K}_{\boldsymbol{y}_L, end}) + \sum_{i=1}^{L} \boldsymbol{P}^{wc}_{i, \boldsymbol{y}_i}. \tag{6}$$

In this way, $S(\boldsymbol{r}, \boldsymbol{y})$ is able to measure the reasonableness of the label sequence itself.

Model Training. After constructing the whole architecture, we need to specify the loss function to train our key information extractor. In the CRF module, we should maximize the groundtruth probability $p(\boldsymbol{g}|\boldsymbol{r})$, where $\boldsymbol{g}$ is the actual label sequence for the rule $\boldsymbol{r}$. Based on this maximization objective, the loss function can be formulated in the negative log-likelihood format as follows:

$$\mathcal{L}_{crf} = -\log p(\boldsymbol{g}|\boldsymbol{r}) = -S(\boldsymbol{r}, \boldsymbol{g}) + \log(\sum_{\hat{\boldsymbol{y}} \in Y_{\boldsymbol{r}}} \exp S(\boldsymbol{r}, \hat{\boldsymbol{y}})). \tag{7}$$

The objective of training is to minimize the loss calculated in Equation (7), which is successfully solved by Adam [24] optimizer, a widely-used gradient descent optimization algorithm.

3.4 Script Translator

The script translator in the second stage of our proposed flow is used to translate the extracted key information into the DRC scripts. When transferring to other checkers, we can preserve the extractor and simply replace the translator. The translator design is similar and simple for different checkers, and here we take the Guardian checker [25] as an example.

DRC script is composed of function calling statements. When given the key information, the functions to be called mainly depend on the checking properties. To conveniently search the required function, we can pair the properties and functions together, as shown in Figure 8. In addition, to automatically pass the key information to the function, we also need to connect the parameters of different functions with our semantic roles. As we clearly define the fine-grained semantic roles in Table 1, the relationships can be easily established. For example, parameters that receive the layer name correspond to "Object" or "Relation Object", and parameters that receive the checking value correspond to "Lower Bound" or "Upper Bound" or "Exact Value".

To better illustrate the entire script translation process, we take the translation process of an overlap rule as an example, which is shown in Figure 8. `Layer` is a regular function for each Guardian script and receives the layer names along with their identifiers, which depend on the specific layout design. `InDistance` function receives two layer names and the value to check the overlap. It can be observed that all the required arguments passed to these two functions can be easily obtained from the extracted information. By automatically filling them into the corresponding placeholders, we obtain the final script for overlap checking. It can be seen that the entire translation process is very efficient.

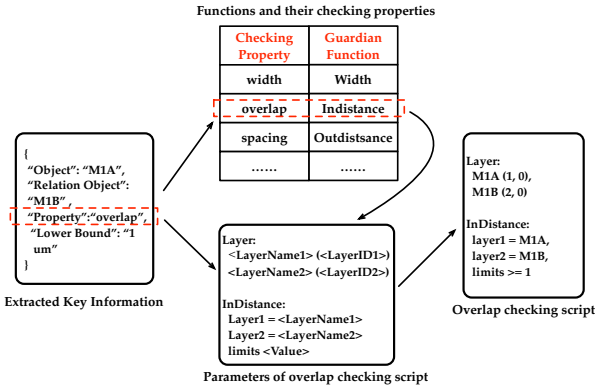

Figure 8: Script translator.

4 EXPERIMENT RESULTS

We implement our entire framework in Python and test it on a platform with the Xeon Silver 4114 CPU processor and NVIDIA TITAN Xp Graphic card. The dataset used for training our key information extractor contains 2970 design rules, 2840 of which are obtained via our proposed rule generation methods, and the rest are the original data from FreePDK15 [20]. To evaluate the performance, another design kit, ASAP7 [26], including 200 design rules on the 7nm node, acts as the test set. Due to the advanced technology node, rules in ASAP7 are more complex compared with our rules on the 15nm node for training. Therefore, the evaluation performance on ASAP7 will convincingly reflect the generalization ability of our framework. We summarize the statistics of the datasets in Table 2. It can also be observed from Table 2 that the "None" category words account for nearly 40 percent, which further demonstrates that a key information extractor can filter a lot of unnecessary information and contribute to the design rules script generation.

Due to the various structure of scripts for different rules, it is not convenient to directly measure the accuracy of the generated scripts. In Section 2.1, we discuss that the script accuracy can be reflected by the extractor performance, and we also explain that the key information extraction task is essentially a word classification task. To evaluate the comprehensive performance, we test the word classification accuracy, inference time of the whole generation process, and robustness ability of the extractor.

Word Classification Results. Table 3 shows the word classification performance of our extractor and two other language models on the test set. Since our work is the first one to investigate key information extraction methods from design rules, no other state-of-the-art work in DRC area can be referred to for comparison. Therefore, we implement two baseline models, bidirectional RNN (Bi-RNN) and bidirectional LSTM (Bi-LSTM). Similar to BERT, Bi-RNN [27] and Bi-LSTM [28] are commonly used for learning word features combined with context information and output the category prediction results, which match the objective of our extractor.

The comparison results illustrate that our customized extractor achieves the best performance on each category. It averagely outperforms Bi-RNN with 20.4% and 12.1% improvement in precision and recall and 17.8% rise in F1 score. Moreover, it surpasses Bi-LSTM with an average precision, recall and F1 score of 15.0%, 12.4% and 14.4%.

Table 2: Semantic roles distribution of dataset

Semantic Roles	Training Set		Test Set	
	#	Percent (%)	#	Percent (%)
Object	6054	15.68	491	14.74
Relation Object	2561	6.63	181	5.43
Property	4705	12.18	300	9.01
Condition	5061	13.10	596	17.89
Restriction	1404	3.64	159	4.77
Lower Bound	2482	6.43	326	9.79
Upper Bound	1144	2.96	8	0.24
Exact Value	1430	3.70	40	1.20
None	13778	35.68	1230	36.93
Total	38619	100	3331	100

Table 3: Comparison with two other language models

Categories	Bi-RNN [27]			Bi-LSTM [28]			Ours		
	Precision	Recall	F1	Precision	Recall	F1	Precision	Recall	F1
Object	0.609	0.666	0.636	0.774	0.662	0.714	0.853	0.804	0.828
Relation Object	0.436	0.674	0.529	0.422	0.674	0.519	0.849	0.896	0.872
Property	0.879	0.970	0.922	0.894	0.953	0.923	0.892	0.900	0.896
Condition	0.786	0.767	0.776	0.669	0.757	0.710	0.818	0.838	0.828
Restriction	0.389	0.453	0.419	0.538	0.403	0.460	0.789	0.704	0.744
Lower Bound	0.947	0.871	0.907	0.960	0.883	0.920	0.967	0.907	0.936
Upper Bound	0.500	1.000	0.667	0.429	0.750	0.545	0.889	1.000	0.941
Exact Value	0.371	0.650	0.473	0.750	0.900	0.818	0.741	1.000	0.851
None	0.874	0.714	0.775	0.893	0.825	0.858	0.892	0.894	0.893
Average	0.649	0.759	0.685	0.703	0.756	0.719	**0.853**	**0.880**	**0.863**
Ratio	0.761	0.863	0.794	0.824	0.859	0.833	**1.000**	**1.000**	**1.000**

Inference Time. In addition to the satisfactory accuracy, our flow also shows superior efficiency. We first test the inference time of the extractor on ASAP7 dataset, which contains 200 design rules. The total runtime result is 1.0s, from which we can calculate that our model averagely takes only 5.0ms to process a single rule. As for the translator, since it is simply responsible for deciding which function to call and passing the extracted key information to the function, the translation process is also high-efficient. According to our measurement, the script translator spends around 0.46ms processing one item of key information. In conclusion, by combining the extractor and translator, our framework can generate a single script in 5.46ms on average, indicating that our proposed DRC script generation flow is extremely efficient.

5 CONCLUSION

In this paper, we propose an automatic DRC scripts generation flow. We first build up a deep learning-based extractor that efficiently recognizes the key information from design rules and then utilize a script translator to organize the extracted information into the scripts. To implement a high-performance extractor, we propose three rule generation methods to expand the volume of rule data, which is beneficial for improving the generalization ability of our extractor. As for the architecture design, we utilize the pretrained powerful language model BERT as our backbone to extract effective representations from the input rules. Experimental results have confirmed the satisfactory performance on both accuracy and efficiency of our framework. We hope that our work can provide some preliminary solutions for automating the generation of DRC scripts and help reduce the manual workload.

ACKNOWLEDGMENTS

This work is supported in part by HiSilicon and The Research Grants Council of Hong Kong SAR (Project No. CUHK14208021).

REFERENCES

[1] "Klayout." [Online]. Available: https://www.klayout.de/doc/manual/drc.html
[2] Y. Lin, S. Dhar, W. Li, H. Ren, B. Khailany, and D. Z. Pan, "DREAMPlace: Deep learning toolkit-enabled GPU acceleration for modern VLSI placement," in *Proc. DAC*, 2019.
[3] Z. Xie, Y.-H. Huang, G.-Q. Fang, H. Ren, S.-Y. Fang, Y. Chen, and J. Hu, "Routenet: Routability prediction for mixed-size designs using convolutional neural network," in *Proc. ICCAD*, 2018, pp. 1–8.
[4] D. Hyun, Y. Fan, and Y. Shin, "Accurate wirelength prediction for placement-aware synthesis through machine learning," in *Proc. DATE*, 2019, pp. 324–327.
[5] Y. Ma, J.-R. Gao, J. Kuang, J. Miao, and B. Yu, "A unified framework for simultaneous layout decomposition and mask optimization," in *Proc. ICCAD*, 2017, pp. 81–88.
[6] H. Yang, S. Li, Z. Deng, Y. Ma, B. Yu, and E. F. Y. Young, "GAN-OPC: Mask optimization with lithography-guided generative adversarial nets," *IEEE TCAD*, 2020.
[7] G. Chen, W. Chen, Y. Ma, H. Yang, and B. Yu, "DAMO: Deep agile mask optimization for full chip scale," in *Proc. ICCAD*, 2020.
[8] C. B. Harris and I. G. Harris, "Glast: Learning formal grammars to translate natural language specifications into hardware assertions," in *Proc. DATE*, 2016, pp. 966–971.
[9] J. Zhao and I. G. Harris, "Automatic assertion generation from natural language specifications using subtree analysis," in *Proc. DATE*, 2019, pp. 598–601.
[10] R. Krishnamurthy and M. S. Hsiao, "Transforming natural language specifications to logical forms for hardware verification," in *Proc. ICCD*, 2020, pp. 393–396.
[11] A. F. Tabrizi, L. Rakai, N. K. Darav, I. Bustany, L. Behjat, S. Xu, and A. Kennings, "A machine learning framework to identify detailed routing short violations from a placed netlist," in *Proc. DAC*, 2018, pp. 1–6.
[12] R. Islam and M. A. Shahjalal, "Late breaking results: Predicting drc violations using ensemble random forest algorithm," in *Proc. DAC*, 2019, pp. 1–2.
[13] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, Ł. Kaiser, and I. Polosukhin, "Attention is all you need," in *Proc. NIPS*, 2017, pp. 5998–6008.
[14] D. Cer, Y. Yang, S.-y. Kong, N. Hua, N. Limtiaco, R. S. John, N. Constant, M. Guajardo-Céspedes, S. Yuan, C. Tar *et al.*, "Universal sentence encoder," *arXiv preprint arXiv:1803.11175*, 2018.
[15] J. Devlin, M.-W. Chang, K. Lee, and K. Toutanova, "BERT: Pre-training of deep bidirectional transformers for language understanding," *arXiv preprint arXiv:1810.04805*, 2018.
[16] P. J. Liu, M. Saleh, E. Pot, B. Goodrich, R. Sepassi, L. Kaiser, and N. Shazeer, "Generating wikipedia by summarizing long sequences," in *Proc. ICLR*, 2018.
[17] W. Yang, Y. Xie, A. Lin, X. Li, L. Tan, K. Xiong, M. Li, and J. Lin, "End-to-End Open-Domain Question Answering with BERTserini," *NAACL HLT 2019*, p. 72, 2019.
[18] J. Zhu, Y. Xia, L. Wu, D. He, T. Qin, W. Zhou, H. Li, and T. Liu, "Incorporating BERT into Neural Machine Translation," in *Proc. ICLR*, 2019.
[19] Y. Zhu, R. Kiros, R. Zemel, R. Salakhutdinov, R. Urtasun, A. Torralba, and S. Fidler, "Aligning books and movies: Towards story-like visual explanations by watching movies and reading books," in *Proc. ICCV*, 2015, pp. 19–27.
[20] K. Bhanushali, "Design Rule Development for FreePDK15: An Open Source Predictive Process Design Kit for 15nm FinFET Devices," Ph.D. dissertation, 05 2014.
[21] P. R. Kingsbury and M. Palmer, "From treebank to propbank." in *LREC*, 2002, pp. 1989–1993.
[22] C. F. Baker, C. J. Fillmore, and J. B. Lowe, "The Berkeley framenet project," in *Proc. ACL*, 1998, pp. 86–90.
[23] J. Lafferty, A. McCallum, and F. C. Pereira, "Conditional random fields: Probabilistic models for segmenting and labeling sequence data," in *Proc. ICML*, 2001.
[24] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.
[25] "Guardian." [Online]. Available: https://silvaco.com/wp-content/uploads/product/pdf/guardian_brief.pdf
[26] L. T. Clark, V. Vashishtha, L. Shifren, A. Gujja, S. Sinha, B. Cline, C. Ramamurthy, and G. Yeric, "ASAP7: A 7-nm finFET predictive process design kit," *Microelectronics Journal*, vol. 53, pp. 105–115, 2016.
[27] J. Zhou and W. Xu, "End-to-end learning of semantic role labeling using recurrent neural networks," in *Proc. ACL*, 2015, pp. 1127–1137.
[28] M. E. Peters, M. Neumann, M. Iyyer, M. Gardner, C. Clark, K. Lee, and L. Zettlemoyer, "Deep contextualized word representations," in *Proceedings of NAACL-HLT*, 2018, pp. 2227–2237.