Frontiers of Information Technology & Electronic Engineering
www.jzus.zju.edu.cn; engineering.cae.cn; www.springerlink.com
ISSN 2095-9184 (print); ISSN 2095-9230 (online)
E-mail: jzus@zju.edu.cn

UI layers merger: merging UI layers via visual learning and boundary prior*

Yunnong CHEN[1,5], Yankun ZHEN[4], Chuning SHI[2], Jiazhi LI[2], Liuqing CHEN[†‡2,3,5], Zejian LI[1,3,5], Lingyun SUN[2,3,5], Tingting ZHOU[4], Yanfang CHANG[4]

[1]*School of Software Technology, Zhejiang University, Hangzhou 310027, China*
[2]*College of Computer Science and Technology, Zhejiang University, Hangzhou 310027, China*
[3]*Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies, Hangzhou 310027, China*
[4]*Alibaba Group, Hangzhou 311121, China*
[5]*Zhejiang-Singapore Innovation and AI Joint Research Lab, Hangzhou 310027, China*
[†]E-mail: chenlq@zju.edu.cn

Received Mar. 15, 2022; Revision accepted Aug. 31, 2022; Crosschecked Feb. 22, 2023

Abstract: With the fast-growing graphical user interface (GUI) development workload in the Internet industry, some work attempted to generate maintainable front-end code from GUI screenshots. It can be more suitable for using user interface (UI) design drafts that contain UI metadata. However, fragmented layers inevitably appear in the UI design drafts, which greatly reduces the quality of the generated code. None of the existing automated GUI techniques detects and merges the fragmented layers to improve the accessibility of generated code. In this paper, we propose UI layers merger (UILM), a vision-based method that can automatically detect and merge fragmented layers into UI components. Our UILM contains the merging area detector (MAD) and a layer merging algorithm. The MAD incorporates the boundary prior knowledge to accurately detect the boundaries of UI components. Then, the layer merging algorithm can search for the associated layers within the components' boundaries and merge them into a whole. We present a dynamic data augmentation approach to boost the performance of MAD. We also construct a large-scale UI dataset for training the MAD and testing the performance of UILM. Experimental results show that the proposed method outperforms the best baseline regarding merging area detection and achieves decent layer merging accuracy. A user study on a real application also confirms the effectiveness of our UILM.

Key words: User interface (UI) to code; UI design lint; UI layer merging; Object detection
https://doi.org/10.1631/FITEE.2200099 **CLC number:** TP39

1 Introduction

The graphical user interface (GUI) is an important visual communication tool that can play an essential role in an app's success. A user interface (UI) design draft is a high-fidelity GUI prototype, and it has a view hierarchy representing both how UI components are constructed and how they are arranged. One of the main jobs of a front-end engineer is to implement the code. With today's increasing development in the Internet industry, there is a huge demand for front-end code development. To relieve front-end developers from tedious and repetitive work, some previous research has adopted automatic methods to generate maintainable code from UI screenshots (Behrang et al., 2018; Beltramelli, 2018), but it can be more suitable to generate code

‡ Corresponding author

* Project supported by the National Key R&D Program of China (No. 2018AAA0100703), the National Natural Science Foundation of China (Nos. 62006208 and 62107035), the Ng Teng Fong Charitable Foundation in the form of ZJU-SUTD IDEA Grant, and the Alibaba-Zhejiang University Joint Research Institute of Frontier Technologies

ORCID: Liuqing CHEN, https://orcid.org/0000-0002-9049-0394

from UI design drafts that contain the UI metadata. Imgcook (https://www.imgcook.com/) is such a tool that can automatically generate front-end code from UI design drafts.

To generate high-quality and maintainable front-end code with automatic code generation tools, the UI design drafts need a concise and structured view hierarchy. In practice, designers usually produce a UI design draft by facilitating the overlay of layers that represent basic shapes and visual elements, with design software such as Sketch (https://www.sketch.com/) and Figma (https://www.figma.com/). These fragmented layers without structured grouping inevitably increase the difficulty of understanding the semantics of UI components and also impair the maintainability of the generated code.

In a code generation process, the associated fragmented layers need to be merged into UI components to avoid increasing the complexity of the hierarchical structure and even reduce the quality of generated code. As shown in Fig. 1, the dashed rectangular boxes represent the fragmented layers. Note that some fragmented layers together represent one UI component. As shown in Fig. 2, the view hierarchy without merging fragmented layers in the UI icon is complicated and redundant. After merging, the layout structure is simplified greatly because we can use a single container to represent the component. However, due to the substantial number of layers and complex design patterns in a UI design draft, it is time-consuming for designers or developers to locate and merge multiple fragmented layers manually. In this study, we try to locate all the UI components that have fragmented layers that need to be merged. Then we can merge these layers into UI components. Therefore, the quality of the generated code can be improved because the complexity of the view hierarchy is greatly reduced.

As described above, the challenge is to determine the location and number of UI components in a design draft. The number of UI components in a given design draft is diverse. The UI components have various types, such as icon, atmosphere UI, and background UI. Multiple types result in different sizes and aspect ratios. For example, there is a significant size difference between the background UI (top-left) and the WiFi icon (top-right) as shown in Fig. 1. Another challenge is that we need to search for all the associated fragmented layers in the region of UI components when their locations are found. Fig. 1 shows that the atmosphere UI (middle-left) consists of six fragmented layers and we have to find all of them accurately. Intuitively, the more accurate the detected UI components' boundaries are, the more accurately we can find the associated layers.

In particular, our method solves the problem in two steps: merging area detection and layer merging, which are implemented by the merging area detector (MAD) and the layer merging algorithm, respectively. The MAD can automatically detect UI component areas. The layer merging algorithm uses the located merging area to merge the fragmented

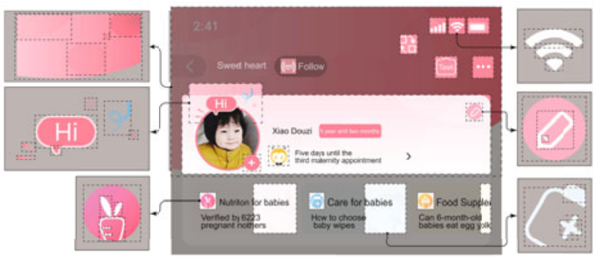
2:41
Sweet heart
Follow
Hi
Xiao Douzi
Five days until the third maternity appointment
Nutrition for babies
Verified by 6223 pregnant mothers
Care for babies
How to choose baby wipes
Food Supple
Can 6-month-old babies eat egg yolk

Fig. 1 Examples of fragmented layers in user interface (UI) components (the dashed rectangles represent the fragmented layers). These fragmented layers form UI components with semantic information

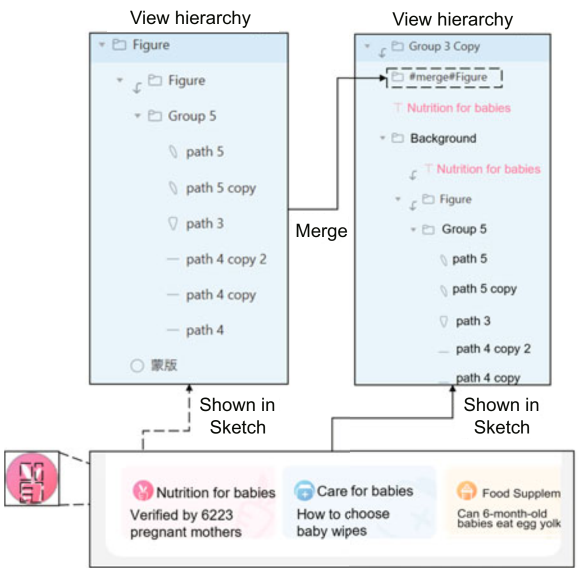

Fig. 2 An example of merging fragmented layers. The dashed boxes in the "carrot-like" icon represent fragmented layers which are shown in the view hierarchy on the top-left side under the "Figure" folder. After merging these fragmented layers, the view hierarchy on the top-right side is simplified as one single "#merge#" container

layers into UI components. Before feeding data into the MAD, we construct a preprocessing pipeline that parses the UI design drafts to obtain the screenshots with view hierarchy. The layer boundary information in the view hierarchy can be incorporated to the MAD as prior knowledge. Additionally, a novel data augmentation and spatial fusion (SF) strategy is introduced to boost the performance of our MAD. To train our MAD and evaluate the effectiveness of our UI layers merger (UILM), we collect and construct a UI dataset with modern diverse UI design drafts.

We summarize the contributions of this study as follows:

1. We construct a large UI dataset (https://github.com/zju-d3/UILM) consisting of UI design drafts. A dynamic data augmentation and SF strategy is introduced to boost the performance of our method.

2. This is the first work to solve the fragmented layer issue by proposing a method called UILM, which contains an MAD and a layer merging algorithm.

3. Our MAD outperforms the best merging area detection baseline. The experimental results based on three specific development conditions show that our method can successfully support automatic code generation. The user study on a real application also confirms the effectiveness of our UILM.

2 Related works

2.1 UI code generation

To free the developer from tedious and repetitive work, recent research on automatically generating code from various design stages has been conducted, including hand-drawn sketches (Aşıroğlu et al., 2019; Jain et al., 2019; Suleri et al., 2019), wireframes (Halbe and Joshi, 2015; Ge, 2019), and GUI screenshots (Nguyen and Csallner, 2015; Beltramelli, 2018; Chen CY et al., 2018; Moran et al., 2020; Feng et al., 2021). Suleri et al. (2019) proposed an approach for web code generation from hand-drawn mock-ups using computer vision techniques and deep learning methods. Sketch2Code (Jain et al., 2019) employs deep neural networks (DNNs) to detect GUI elements in sketches. Its output is a platform-independent UI representation object used by a GUI parser to create code for different platforms. However, the generation from sketches or wireframes to code still requires manually modifying the generated code. To improve the usability of the generated code, pix2code (Beltramelli, 2018), which is based on convolutional neural networks (CNNs) and recurrent neural networks (RNNs), can generate code from a GUI screenshot. To generate more appropriately structured code, Chen CY et al. (2018) presented a neural machine translator that combines computer vision and machine translation techniques for converting a UI design image to a GUI skeleton.

The above-mentioned UI code generation techniques do not fully use UI metadata, such as view hierarchies or accessibility tags, and thus some work is needed to improve the quality of generated code by taking the UI design drafts to generate more useful and maintainable front-end code. Yotako (https://yotako.io/) and Imgcook are tools that take UI design drafts made by designers as input for modern design software. The resulting code from these tools describes the original UI using only relative constraints, allowing it to be responsive to different front-end platforms, e.g., Vue, React, and Angular. However, the existence of fragmented layers misleads the machine, so there are many fragmented containers in the generated code. The generated code often does not satisfy developers' requirements due to the lack of maintainability. We seek to address this limitation and advance the body of work on code generation with our approach.

2.2 UI detection and dataset

The first step of our method is to use UI screenshots and raw view hierarchies to detect the fragmented layer regions. Here we briefly survey existing GUI element or component detecting techniques. Some works (Chen S et al., 2019; Chen JS et al., 2020; Moran et al., 2020) first use traditional image processing methods, such as edge detection, to locate UI elements, and then apply the CNN model to identify the semantics of UI elements (e.g., UI type). Liu et al. (2020) presented OwlEye to classify and detect GUI display issues, such as missing images and text overlap, to guide developers to fix these bugs. They further developed a fully automated approach, Nighthawk (Liu et al., 2023), which is based on the faster region-based convolutional neural network (Faster RCNN) model, to detect GUIs with display issues and locate the detailed region of the issue to guide developers to fix bugs. Gallery D.C. (Chen CY et al., 2019) uses Faster RCNN to detect UI components and automatically creates a gallery of GUI design components. White et al. (2019) used YOLOv2 to identify GUI widgets in UI screenshots to improve GUI testing. Zhang et al. (2021) collected and annotated a large UI dataset from iPhone apps to train a robust and efficient on-device model to detect UI elements.

To train a model for detecting and merging fragmented layers in UI design drafts, a large-scale dataset consisting of UI design drafts is mandatory. Recently, Rico (Deka et al., 2017), a large-scale dataset of Android apps, is generally used as the data source for related research on UI. Different from the traditional image-based dataset, Rico consists of about 72 000 UI examples from 9722 Android apps. Each example is associated with a screenshot of a particular UI design, the corresponding view hierarchy, and the user interaction information. It marks 24 UI component types, 197 text button concepts, and 97 icon types on these view hierarchies. However, it has some limitations, such as broken hierarchies, inaccurate bounding box boundaries, and inconsistencies in the class labels of similar objects (Bunian et al., 2021). Although Li et al. (2022) developed tools for improving the quality of the Rico dataset, more complex UI layouts and UI design specifications have been a challenge for UI research. Zhang et al. (2021) attempted to fill this gap by collecting and annotating a large UI dataset from modern iPhone apps. Hence, it is important to have a well-structured dataset that can provide more accurate and more diverse UI design to support UI research. Also, to the best of our knowledge, there is no public dataset for the task of detecting fragmented layers. Therefore, we collect UI drafts and construct a UI dataset with modern diverse UI components that enables us to train a fragmented layer MAD.

In this study, we propose a novel object-detection-based method to detect the regions of the fragmented layers in UI design drafts. Given that UI design drafts have view hierarchies, we enrich visual features by incorporating the layers' boundary information.

3 Methodology

In this section, we introduce the details of the proposed method. As illustrated in Fig. 3, we present a dynamic data augmentation approach, and design the MAD. The segmentation map encodes the boundary information and the location of the UI components. By adding the segmentation map to the UI images, MAD can use the spatial information of layers to condition the proposed bounding features. The layer merging algorithm can merge fragmented layers into UI components. After merging the layers, they can be transformed into one single image-type layer, which can significantly reduce the complexity of layers in the UI design drafts and improve the quality of generated code.

3.1 Dataset construction

To investigate our automatic approach for merging fragmented layers in design drafts, it is essential to have a large-scale, carefully annotated dataset of UI design drafts. Thus, we create the UILM dataset,

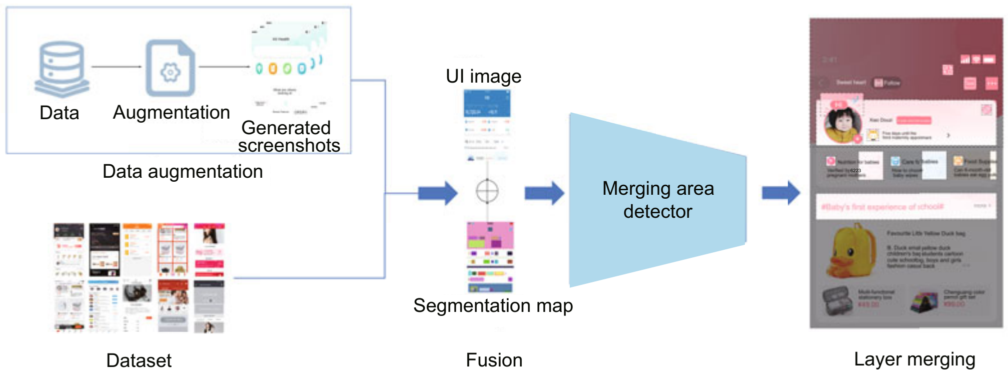

Fig. 3 Overview of the proposed method

a novel well-annotated dataset of >600 Sketch files. Each Sketch file consists of several design drafts. Each design draft has a view hierarchy for organizing layers. A view hierarchy is a tree structure where each layer in the tree corresponds to an element in the UI. Different from Rico (Deka et al., 2017), our layers are the indivisible elements that make up UI objects in design software. In fact, in design drafts, it is very common for multiple fragmented layers to form a UI component, such as icons and background UI.

When constructing the dataset, we recruit UI designers to perform data annotation. They are asked to read a document that details the steps for data annotation. Then they are provided with an example set of annotated UI design drafts where all the fragmented layers are clustered into several groups and associated with the "#merge#" label. This provides a better understanding of how to make a correct "#merge#" annotation. Eventually, they use Sketch to manually locate fragmented layers for given UI design drafts and manually merge these layers into groups. Consequently, these groups are named starting with "#merge#."

In the data preprocessing stage, given an annotated UI design draft, we parse its artboard to a JSON file and generate the train labels using the COCO format (Lin et al., 2014). The parsing pipeline is shown in Fig. 4. The Sketch tools save the corresponding screenshots. The layers in an artboard can be regarded as a view tree. We traverse the tree from bottom to top and obtain the spatial information of all layers. The information of each layer includes position, size, type, fill, etc. When we traverse to the layer group containing the "#merge#" label, we create the training data $[x, y, w, h]$ based on the previously obtained position and size information ((x, y) represents the coordinates of the top-left corner, and w and h denote the width and height information respectively). To highlight the boundary information, we generate segmentation maps by filling the layers with colors in traversing order. In summary, with UI drafts, we generate UI screenshots, corresponding segmentation maps, and training data containing groups of "#merge#" labels and layer information.

3.2 Merging area detection

We need to locate the merging area of the UI component and determine reliable boundary for the area. It is difficult to merge layers with high accuracy if the predicted boundary is not accurate enough. As illustrated in Fig. 5, to solve this issue, we adopt multi-stage adaptive convolution (Vu et al., 2019) to learn an anchor with an adaptive shape to improve the boundaries' alignment accuracy in the context of complex UI layouts.

Specifically, given a feature map x, we calculate each location p on the output feature y in the adaptive convolution as follows:

$$y[p] = \sum_{O \in \varOmega} \omega[O] \cdot x[p + O], \tag{1}$$

where $\varOmega$ is the offset field and offset O can be

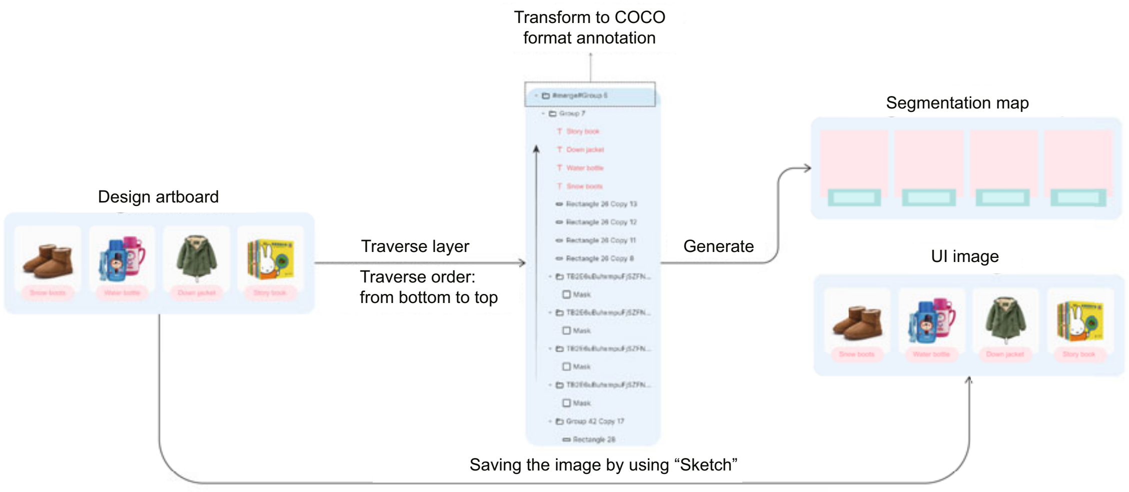

Fig. 4 Parsing pipeline of user interface (UI) draft preprocessing

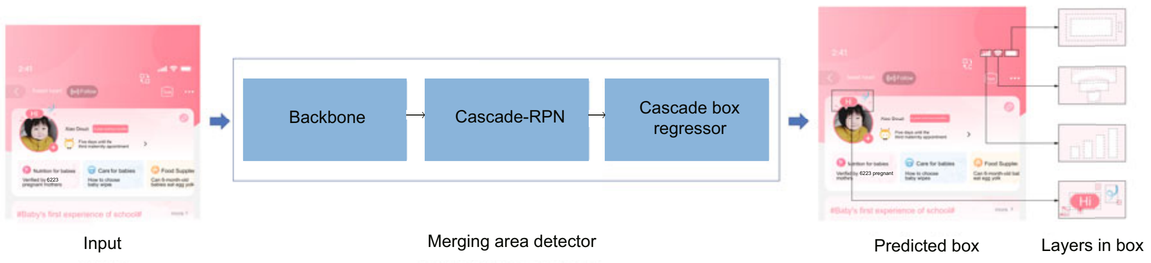

Fig. 5 Architecture of the merging area detector (RPN: region proposal network)

decomposed into center offset and shape offset, which can be obtained by

$$O = O_{\text{ctr}} + O_{\text{shp}}, \tag{2}$$

where $O_{\text{ctr}} = (\bar{a}_x - p_x, \bar{a}_y - p_y)$, O_{shp} is defined by anchor shape and kernel size, and $\bar{a}$ denotes the projection of anchor a onto the feature map.

At the region proposal network, to select sufficient positive samples and avoid establishing a loose requirement for positive samples, we use two stages by starting out with an anchor-free metric followed by an anchor-based metric in the second stage. In the first stage, the adaptive convolution is set to perform dilated convolution because anchor center offsets are zeros. Its function is to enlarge the feature map perception field. In the second stage, we compute the anchor offset O^τ and feed it into the regressor f^τ to produce the regression prediction $\hat{\delta}^\tau$. The prediction $\hat{\delta}^\tau$ is used to produce regressed anchors $a^{\tau+1}$. The objectness scores are then produced from the classifier, followed by non-maximum suppression (NMS) to produce the region proposals.

At the box regressor, improving the layer boundary alignment rate depends on the box regression performance of the detector. In Faster RCNN (Ren et al., 2017), a fixed threshold, intersection over union (IoU), is set, typically $u = 0.5$, which establishes quite a loose requirement for positives. This makes it difficult to train a detector to achieve the optimal performance. If the IoU is set higher, typically u=0.7, it leads to a few positive samples after filtering and causes the model to be over-fitting. In this study we follow the suggestions of Cai and Vasconcelos (2018) that a detector optimized at a single IoU level is not necessarily optimal at other levels. In other words, each IoU level can learn an optimal bounding box regressor independently. We adopt the cascade box regressor, which improves performance by multi-stage expansion while avoiding the over-fitting problem.

3.3 Layer merging algorithm

Our ultimate goal is to find redundant layers from the layer list and merge them into UI components. Therefore, after detecting the merging areas from screenshots, we need to trace back to the corresponding layers from UI drafts in the detected merging area. We propose the layer merging algorithm, which merges the relevant layers while filtering the irrelevant layers in the detected area.

The complete merging process is shown in Algorithm 1. With the JSON file and the predicted bounding box list as the input, we first flatten the

Algorithm 1 Layer merging

```
Input: $N$ predicted bounding boxes $\{\text{bb}_i\}_{i=1}^N$ and their
    JSON files
Output: result group res
 1: $T_i \leftarrow$ pre-determined threshold of the intersection
 2: Traverse JSON files to obtain the flatten layer list
    $\{\text{fl}_j\}_{j=1}^M$ and obtain the indices of the layers
 3: Arrange $\{\text{bb}_i\}_{i=1}^N$ in ascending order
 4: for all $\text{bb}_i$ in bb do
 5:   for all $\text{fl}_j$ in fl do
 6:     if $\text{fl}_j \cap \text{bb}_i > T_i$ then
 7:       Save the layer $\text{fl}_j$ to the filtered list $\{\text{fi}_k\}_{k=1}^K$
 8:     end if
 9:   end for
10:   Compute the distance threshold $T_{\text{d}}$
11:   for all $\text{fi}_k$ in fi do
12:     if $\text{fi}_{k+1}$.index $-$ $\text{fi}_k$.index $< T_{\text{d}}$ then
13:       Save the layer to result group res
14:     end if
15:   end for
16:   Remove the layers in res from flatten list fl and
      update fl
17: end for
18: return res
```

hierarchical layers to a layer list, because it is assumed that the tree hierarchy from UI designers is not reliable. We then index the layer list and arrange the predicted bounding boxes in ascending order.

Given a predicted bounding box, we traverse the layers from the flattened list "fl." The area of layers can be calculated. We calculate the intersection area of the given predicted bounding box and all layers. The layers that exceed the pre-determined area threshold T_i are saved to the filtered list "fi." We index the layers by their absolute positions in the list. We can calculate the distance between any two layers by using the index. The mean distance of the filtered layers is used as the distance threshold T_{d}. If the distance between adjacent layers is below the threshold T_{d}, we save the layer to the result group "res." Finally, the saved layers are removed from the flattened layer list, and the flattened layer list is updated. After processing all predicted bounding boxes, we find all the fragmented layers belonging to particular UI components.

3.4 Feature fusion

Most of the UI design drafts in our research are from mobile online shopping platforms, which have pretty rich semantic information, such as components, icons, and backgrounds. In addition, a variety of UI components in e-commerce scenarios lead to complex boundary information in constructed layers, which makes it difficult to learn the bounding spatial features. We therefore propose a fusion strategy to use the spatial information as prior knowledge.

There are two strategies for using the spatial features based on the segmentation map. As shown in Fig. 6, we first generate the segmentation map which contains only boundary information. In the spatial fusion (SF) strategy, we stack the segmentation map to the original UI image to produce the fusion image. The fusion image contains the specific boundary of the UI component. The fusion images are fed into the CNN backbone to produce a feature map. In the feature fusion (FF) strategy, inspired by Xu et al. (2017), we use the high-dimensional features of spatial information as prior knowledge. Specifically, two high-dimensional feature maps are extracted from the original image and the corresponding segmentation map respectively by the CNN backbone, and are then concatenated as a fused feature map. For hard examples, the fusion strategy can help the backbone learn its boundary information more easily.

3.5 Dynamic data augmentation

Training an effective object detection model for visual understanding requires massive high-quality data. Traditional methods of data augmentation for object detection are mainly image processing, including random flipping, random cropping, and resizing. These methods often cannot be dynamically adjusted to the data. Static data augmentation does not work well for small or large-aspect-ratio objects in the UI design scenario. Furthermore, training the MAD requires a large number of annotated UI design drafts. However, there is so far no such type of open dataset, and collecting such UI design drafts requires great effort and time.

In this study, we propose the dynamic data augmentation method to generate training data using existing Sketch files. Because the size of UI layers ranges widely, our data augmentation approach focuses on small components and large-aspect-ratio components and we present the dynamic data augmentation algorithm. Given the screenshot and associated JSON file, we traverse the layer list and randomly remove the layers that should not be merged according to a predefined ratio. We then generate a new JSON file and corresponding screenshot. Note that the layers are randomly deleted at each training epoch, so we have a very diverse training sample. As shown in Fig. 7, the meaning of "dynamic" is to keep the merging layers that are shown in the red solid-line box, and the other irrelevant layers shown in the red dashed box are randomly removed with a certain probability.

4 Experiments

4.1 Experimental setting

4.1.1 Dataset

The dataset contains screenshots of artboards and the corresponding JSON files with UI hierarchies. It is worth mentioning that large-scale artboards may exceed the GPU memory limit, so we split each artboard into several images with a fixed size based on the shorter side. Because the shorter side of these artboards is not fixed, we resize the images such that the shorter side has a maximum of

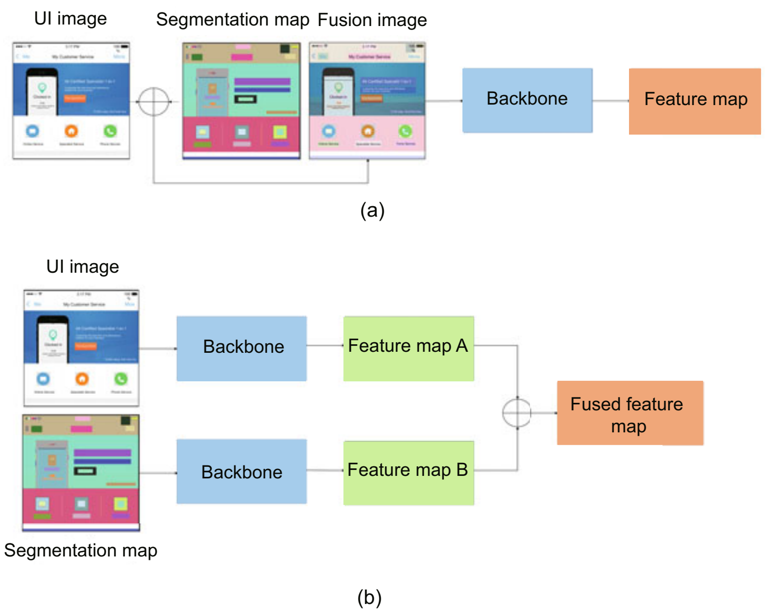

Fig. 6 Two strategies for feature fusion: (a) spatial fusion strategy; (b) feature fusion strategy

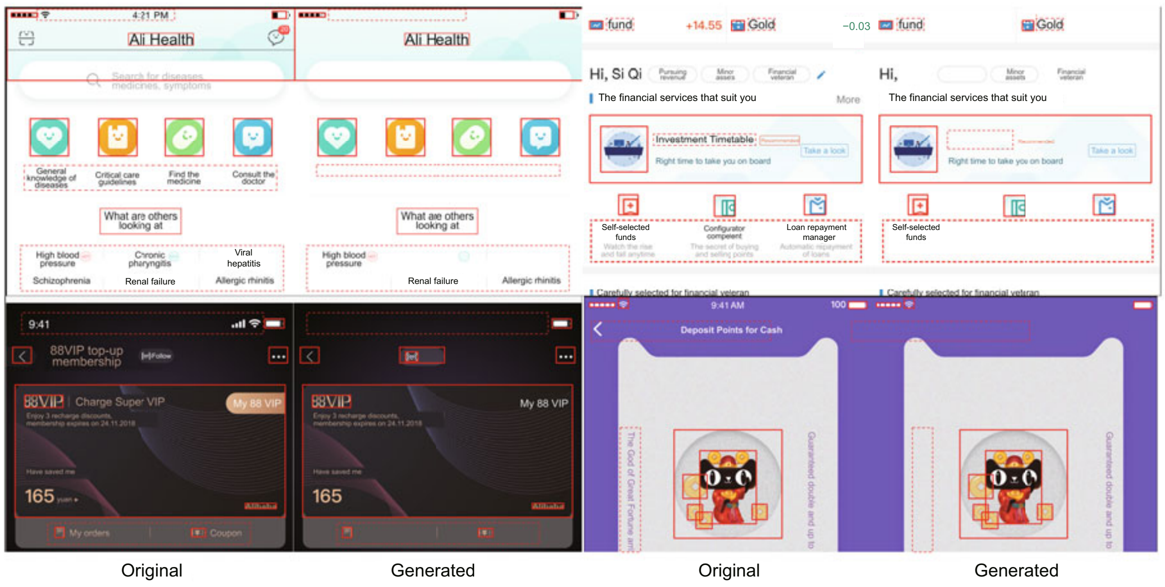

Fig. 7 Examples of data augmentation (solid box: preserved region; dashed box: removed region). References to color refer to the online version of this figure

800 pixels and the larger side has 1333 pixels. When dividing the dataset into the training set and test set, the images from the same artboard are divided into the same set to avoid introducing bias. Duplicated screenshots, which are produced from highly similar design drafts, are removed from the dataset. A total of 7399 screenshots are collected for experiments. To make the model focus on the hard examples, we take 30% of the collected data to augment small objects, which are defined as smaller than 32×32 pixels, and take 30% of the original data to augment large-aspect-ratio objects when the aspect ratio is >3. In total, 5981 screenshots are used as the training set and augmented screenshots are added to the training set. The number of augmented screenshots generated by dynamic data augmentation is 5448. The

test is performed on the remaining 1418 collected screenshots without data augmentation.

4.1.2 Implementation details

We use ResNet50 (He et al., 2016) pre-trained on ImageNet as the backbone network. The feature pyramid network (FPN) (Lin et al., 2017) is used to extract a pyramid of features. In the first stage of the adaptive region proposal network, the anchor-free metrics for sample discrimination with the thresholds of center-region σ_{ctr} and ignore-region σ_{ign} are 0.2 and 0.5 respectively. In the second stage, we use the anchor-based metric with the IoU threshold of 0.7. The multi-task loss is set with the stage-wise weight $\alpha_1 = \alpha_2 = 1$ and the trade-off λ=10. The NMS threshold is set to 0.7. In the cascade box regression network, there are three stages with $\text{IoU} = 0.5, 0.6, 0.7$. In the first stage, the input to the regressor is the adaptive region proposal from the cascade region proposal network. In the following stages, re-sampling is implemented by simply using the regressed outputs from the previous stage. We follow the standard settings as in Vu et al. (2019). We implement our method with the PyTorch and MMDetection codebase (Chen K et al., 2019). We train all models with a mini-batch of 8 for 12 epochs using the stochastic gradient descent (SGD) optimizer with a momentum update of 0.9 and a weight decay of 0.0001. The learning rate is initialized to 0.01 and divided by 10 after 8 and 11 epochs. It takes about 4 h for the models to converge on an NVIDIA GeForce RTX 3090 GPU.

4.1.3 Evaluation metrics

We report performance with the metrics used in the COCO detection evaluation criterion (Lin et al., 2014) and provide mean average precision (mAP) across various IoU thresholds (i.e., IoU $= \{0.50 : 0.05 : 0.95, 0.50, 0.75\}$) and various scales (small, medium, large). In the COCO evaluation, the IoU threshold ranges from 0.50 to 0.95 with a step size of 0.05 represented as AP. The APs at fixed IoUs such as IoU=0.50 and IoU=0.75 are written as AP50 and AP75, respectively.

4.2 Results

4.2.1 Detection performance comparison with baselines

We first present the merging area detection performance. Table 1 shows the performance comparison with the baselines. Note that our MAD uses the proposed SF strategy and dynamic data augmentation. It can be seen that our MAD is much better than the baselines. Specifically, the performance of our MAD is improved by 26.61% and 18.15% in terms of mAP compared to RetinaNet and Faster-RCNN, respectively. It is effective to incorporate boundary prior knowledge to condition the bounding features. Our approach also outperforms competing methods such as Cascade-RCNN and GA-Faster-RCNN by 12.75% and 15.58% respectively, in terms of mAP. This shows that the feature alignment using adaptive anchors and the progressive refinement of boundaries can contribute to a performance boost. Compared to other methods, MAD enhances feature representation by fusing prior knowledge of layers' boundary information. This also implies that MAD is especially good at detecting UI layout.

4.2.2 Comparison of two fusion strategies

We conduct experiments to evaluate the effectiveness of the two fusion strategies separately. Table 2 shows that the SF strategy that fuses layers' boundary information into the original UI image is a better choice. A reasonable explanation is that

Table 1 Performance comparison with baselines

Method	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
RetinaNet	0.545	0.708	0.602	0.523	0.634	0.395
Faster-RCNN	0.584	0.726	0.647	0.588	0.627	0.430
GA-Faster-RCNN	0.597	0.739	0.654	0.605	0.637	0.429
Cascade-RCNN	0.612	0.734	0.665	0.612	0.657	0.456
CRPN-Faster-RCNN	0.638	0.766	0.696	0.638	0.687	0.487
MAD	**0.690**	**0.801**	**0.753**	**0.677**	**0.752**	**0.536**

The best results are in bold. The subscripts "50" and "75" represent that the IoU thresholds are 0.50 and 0.75, respectively. The subscripts "S," "M," and "L" represent area$\leq 32 \times 32$, $32 \times 32 <$ area $\leq 96 \times 96$, and area$> 96 \times 96$, respectively

Table 2 Comparison of two fusion strategies

Method	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
RetinaNet+FF	0.556	0.745	0.633	0.553	0.636	0.272
RetinaNet+SF	**0.602**	**0.754**	**0.679**	**0.580**	**0.682**	**0.435**
Faster-RCNN+FF	0.495	0.690	0.583	0.505	0.563	0.170
Faster-RCNN+SF	**0.622**	**0.757**	**0.697**	**0.614**	**0.683**	**0.442**
MAD+FF	0.607	0.763	0.680	0.611	0.659	0.378
MAD+SF	**0.674**	**0.797**	**0.741**	**0.661**	**0.736**	**0.524**

The better results are in bold. The subscripts "50" and "75" represent that the IoU thresholds are 0.50 and 0.75, respectively. The subscripts "S," "M," and "L" represent area$\leq 32 \times 32$, $32 \times 32 <$ area $\leq 96 \times 96$, and area$> 96 \times 96$, respectively. FF: feature fusion; SF: spatial fusion

the semantic information in the segmentation map is poor, so it is difficult to enrich the features extracted from UI images. The fused features may even corrupt the original UI image feature representation, degrading model performance. The SF strategy enriches the original image with spatial features from layer boundaries at the pixel level, which appears to be more efficient for the backbone to extract the semantic features. The results show that the SF strategy on all three models improves mAP by at least 8.27% over the FF strategy.

4.2.3 Analysis of failure cases

Our MAD does not perform well for complex background components and some complicated UI component shapes. Fig. 8 shows the typical examples that make the detection of merging layers very challenging. For UI components with complex shapes, MAD cannot accurately determine the component's boundaries. For example, the predicted bounding box and the ground truth have a very significant gap, such as the left side of Fig. 8. Therefore, the merging algorithm cannot merge all fragmented layers into the correct UI component. Another challenge is that the model has difficulty in learning salient visual features in complex design scenarios. For example, if the designer designs a background UI component like the right side of Fig. 8, MAD cannot accurately determine the boundaries that belong to the background.

4.2.4 Layer merging performance

To evaluate the layer merging result, we define a metric called mean layer IoU, which is similar to IoU. It is a score from 0 to 1 that specifies the intersection of a prediction layer and the ground-truth layer in the

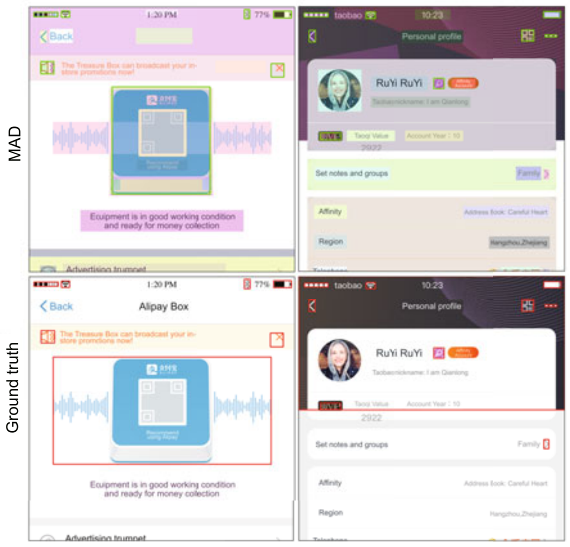

Fig. 8 Failure cases (ground truth is shown in red boxes, while the predictions are shown in green boxes). References to color refer to the online version of this figure

UI component. For example, if there are three layers in a predicted group and four layers in ground truth, then the layer IoU is 0.75. We average the sum of the IoU of all layer groups, and perform operations on all labeled data to find the layers that should be merged. Our model achieves 87.70% in terms of mean layer IoU. This implies that the layer merging algorithm can achieve decent accuracy.

4.3 Ablation study

4.3.1 Data augmentation

We investigate the contribution of the proposed data augmentation approach. "MAD+DataAug" denotes the adoption of the data augmentation approach in MAD. "MAD+SF+DataAug" denotes the adoption of both the data augmentation approach and the SF fusion strategy in MAD. From Table 3 we can observe that "MAD+SF+DataAug" improves mAP by 1.6% from "MAD+SF." This indicates the effectiveness of data augmentation for merging area detection. The training set with the augmented data has more diverse samples for small and large-aspect-ratio objects, which increases the scalability of the model.

4.3.2 Spatial fusion

We conduct experiments with or without the fusion strategy. Note that we use the SF strategy according to the previous experimental results. Table 3 indicates that after encoding the boundary information, MAD has 2.3% mAP improvement. The obvious gain brought by the SF strategy suggests the necessity of the bounding spatial prior knowledge which enriches feature representation. In addition, applying the fusion strategy to Faster-RCNN and RetinaNet yields 5.7% and 3.8% mAP improvement, respectively. This also demonstrates that the prior knowledge of boundary information can boost the performance of UI component detection models.

Fig. 9 presents the advantages of SF with examples. As shown on the right side of Fig. 9, the SF strategy adds a clear boundary to distinguish different UI component areas. Thus, the model can visually identify the region as a combination of multiple layers more easily, and increase the prediction performance of layer merging. Furthermore, in a real design scenario, the designer may have already merged some layers like the left side of Fig. 9. Therefore, there is no need to detect the digital icons, but there is no information here to prompt the model. This could cause the model to wrongly recognize the UI component region. The results show that MAD with the SF strategy detects the UI component successfully, because the model can learn visual boundary features to avoid detecting UI components that have been merged.

5 User study and application

The goal of this work is to automatically merge fragmented layers in UI design drafts to reduce the time developers spend on understanding and modifying code, and improve development efficiency. The extensive experiments above demonstrate that our model outperforms the baselines with decent advance. However, the satisfaction of the generated code might be subjective depending on different users or developers. In this section, we build a complete pipeline for merging fragmented layers called UILM. Specifically, a screenshot with view hierarchy is fed into a parsing pipeline to produce a

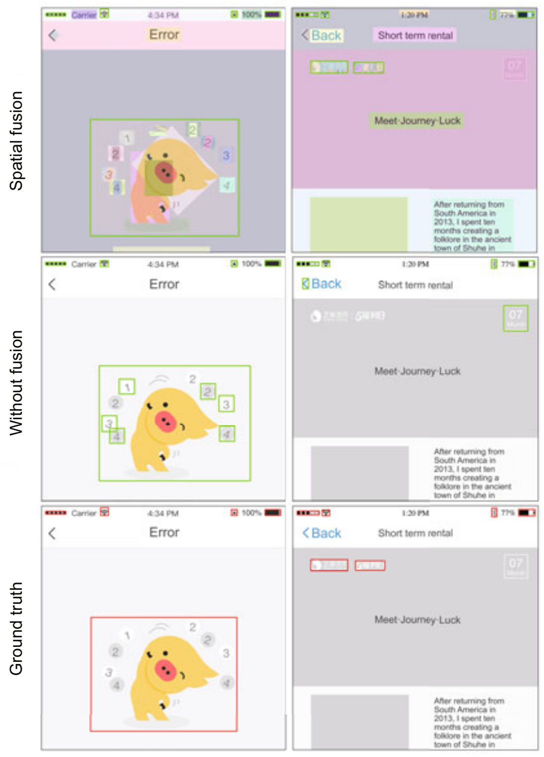
Spatial fusion
Without fusion
Ground truth
Error
Back
Short term rental
Meet·Journey·Luck
After returning from South America in 2013, I spent ten months creating a folklore in the ancient town of Shuhe in

Fig. 9 Examples of spatial fusion (ground truth is shown in red boxes, and the predictions are shown in green boxes). References to color refer to the online version of this figure

Table 3 Ablation studies

Method	AP	AP_{50}	AP_{75}	AP_S	AP_M	AP_L
MAD	0.651	0.778	0.705	0.658	0.688	0.512
MAD+SF	0.674	0.797	0.741	0.661	0.736	0.524
MAD+DataAug	0.659	0.775	0.706	0.666	0.704	0.519
MAD+SF+DataAug	**0.690**	**0.801**	**0.753**	**0.677**	**0.752**	**0.536**

The best results are in bold. The subscripts "50" and "75" represent that the IoU thresholds are 0.50 and 0.75, respectively. The subscripts "S," "M," and "L" represent area$\leq 32 \times 32$, $32 \times 32 <$ area $\leq 96 \times 96$, and area$> 96 \times 96$, respectively segmentation map. Then the screenshot and corresponding segmentation map are fused and fed into our MAD to predict the bounding boxes as detected merging areas. Our layer merging algorithm takes the merging areas and searches for the associated layers to be merged. To better evaluate the usefulness of UILM, we conduct a user study to investigate feedback from developers. We also apply UILM in Taobao's front-end development process.

5.1 Evaluation metrics

There are no existing evaluation metrics for code generation from design drafts in the literature. Inspired by the GUI design evaluation (Zhao et al., 2021), we propose two novel objective metrics for participants to estimate the quality of code generation by considering the characteristics of front-end code implementation: code availability and code modification time. Also, to confirm if UILM improves code readability and maintainability, we propose two subjective metrics by asking participants to rate their experience. The four metrics are quantified by scores ranging from one to five. Code availability evaluates how much generated code is available for production. We use git service (https://git-scm.com/) to record the lines of code changes. The calculation approach is as follows:

$$\text{availability} = 1 - \frac{\text{number of lines of code changes}}{\text{total number of lines of code}}. \tag{3}$$

For data statistics, the metric score is defined as shown in Table 4. A score of one to five corresponds to code availability between 0 and 1.0. Code modification time evaluates the time required to adjust the code to actual production standards. The participants record the time of modifying the generated code. We use 2 min as an interval. A modification time of $\geq$10 min scores one point, and within 4 min scores five points. The scores of readability and maintainability are given by participants from one to five subjectively, representing the quality of code from low to high.

5.2 Procedures

In this study, we define three categories of common UI components that contain fragmented layers, which are icon, atmosphere UI, and background UI. We fetch the corresponding components from design drafts and then generate the code using Imgcook. For each category, we have 10 samples. Note that we do not remove the components that are overlapped on the background UI to ensure generalization.

For code evaluation, we recruit frond-end engineers who have more than three years programming experience and at least two years front-end development experience using Vue (https://vuejs.org/), which is a progressive JavaScript framework. They are introduced to a detailed explanation of the code evaluation metrics. Then they are provided with generated code. We calculate scores for modification time and the number of code changes, and collect their ratings of code readability and maintainability. Note that they are not aware of which code is merged by our method, and all of them evaluate the generated code without any discussion. After the experiment, we ask the participants to leave some feedback about our UILM.

Table 4 Evaluation metrics for code generation

Score	Code availability	Code modification time (min)
1	$0 \leq a < 0.75$	$t \geq 10$
2	$0.75 \leq a < 0.80$	$8 \leq t < 10$
3	$0.80 \leq a < 0.85$	$6 \leq t < 8$
4	$0.85 \leq a < 0.90$	$4 \leq t < 6$
5	$0.90 \leq a \leq 1.00$	$t < 4$

5.3 Results

As shown in Table 5, the code generated after merging the fragmented layers using our UILM outperforms the code without merging in average code availability, modification time, readability, and maintainability. In addition to the average score, our method outperforms the method without merging, on four metrics for all three UI component categories. The results demonstrate the generalizability of our UILM. We analyze the experimental results in detail. The merged icon and atmosphere UI significantly decrease the code modification time, because our UILM is good at detecting icon and atmosphere UI. Also, the code readability and maintainability of the background UI are significantly improved by 70.27% and 48.28%, respectively. Regarding the background UI, UILM not only merges the fragmented layers of the detected background, but also merges the overlapping icons and atmosphere UI. Hence, the improvement is obvious and significant. This information proves that our method can help improve the quality of the generated code

To demonstrate the significance of UILM, we carry out the Mann–Whitney U test (Fay and Proschan, 2010), which is specifically designed for small samples. $p<0.05$ is typically considered to be statistically significant and $p<0.01$ is considered to be highly statistically significant. The results show that our UILM can contribute significantly to code generation in all four metrics except code availability in the background UI. In addition, the participants conclude that merging fragmented layers has a positive effect on improving generated code quality. They think that generated code without merging tends to have redundant containers, but most containers can be fixed after merging. One of the participants emphasizes that UILM has the potential to aid the automatic code generation process: "The key requirement is fast iteration. The generated code with a clean layout is very beneficial to maintain and modify."

5.4 Application

To understand the applicability of UILM from an industrial perspective, we apply our method to Taobao's front-end development process. We first invite Taobao's front-end engineers working on automated code generation development to provide some typical and challenging UI design drafts to be merged in practical application scenarios. To evaluate these representative samples, our model as a plugin in a code generation tool automatically merges fragmented layers in UI design drafts. We visualize some challenging detection results as shown in Fig. 10. The green box represents the components detected by MAD in the first column and the red box represents the layers to be merged in the second column. The results show that the proposed method detects all components successfully and merges all associated layers. It also demonstrates that our UILM can

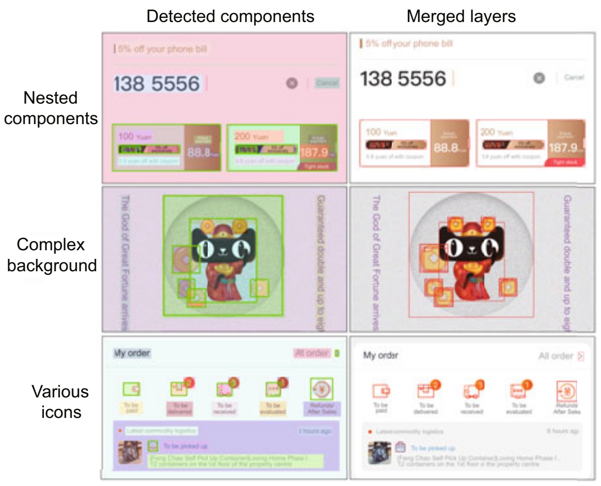

Fig. 10 Examples of associative layer merging results (the green box represents the detected components, and the red box represents the layers to be merged). References to color refer to the online version of this figure

Table 5 Performance of human evaluation

Category	Metric	Score		
		Non-merge	Merge	Increment (%)
Icon	Code availability	2.72	3.54**	30.15
	Code modification time	2.80	3.78*	35.0
	Readability	3.84	4.44*	15.63
	Maintainability	2.98	4.06*	36.24
Atmosphere UI	Code availability	2.72	3.50*	28.68
	Code modification time	2.62	3.68**	40.46
	Readability	3.14	4.20*	33.76
	Maintainability	2.82	3.82*	35.46
Background UI	Code availability	1.94	2.46	26.80
	Code modification time	2.24	2.98*	33.04
	Readability	2.22	3.78**	70.27
	Maintainability	2.32	3.44*	48.28
Average	Code availability	2.46	3.17	28.86
	Code modification time	2.55	3.48	36.47
	Readability	3.07	4.14	34.85
	Maintainability	2.71	3.77	39.11

$^{**}p<0.01$; $^{*}p<0.05$

process various UI components such as various icons and nested components. We also display a document object model (DOM) tree associated with generated front-end code. Fig. 11 shows that without merging fragmented layers, the "text-like" logo consists of 10 image containers in the generated DOM tree. Its nested structure and the redundant containers result in the poor readability and maintainability of generated code. Our UILM can merge these layers into one group with a "#merge#" tag for recognition by downstream code generation algorithms. When recognizing the annotation, an automatic code generation tool, such as Imgcook, can merge all layers into a group to produce a singe image container. After applying our UILM, the DOM tree generated by Imgcook is simplified a lot. The clean DOM tree significantly improves the quality of generated code.

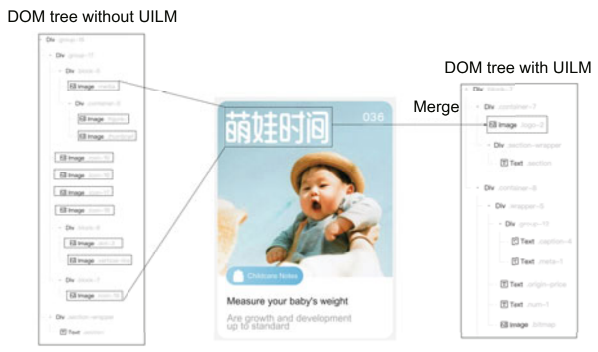
DOM tree without UILM
DOM tree with UILM
Merge
萌娃时间
036
Measure your baby's weight
Are growth and development up to standard

Fig. 11 An example of a generated DOM tree with and without UILM. In the DOM tree without UILM, the dashed boxes represent redundant containers. In the DOM tree with UILM, the single image container represents the "text-like" UI component (DOM: document object model; UI: user interface; UILM: UI layers merger)

6 Conclusions

In this paper we investigated a novel issue concerning layer merging in an automatic design draft to the UI view code process, which can decrease the quality of generated code. To solve this issue, we innovatively proposed UILM by detecting the areas of UI components and merging the fragmented layers into UI components. By incorporating boundary prior knowledge, MAD can achieve more than 5.6% boost in detection mAP compared with the best baseline. We also proposed a dynamic data augmentation approach to boost the performance of MAD. As the first work of merging fragmented layers in UI design drafts, we constructed a large well-annotated UI dataset to train our model and evaluated the effectiveness of our method. Furthermore, UILM was proven to be effective in practice by building a test pipeline.

Contributors

Yunnong CHEN, Liuqing CHEN, and Yankun ZHEN designed the research. Yunnong CHEN and Chuning SHI processed the data. Yunnong CHEN and Liuqing CHEN drafted the paper. Chuning SHI, Jiazhi LI, and Zejian LI helped organize the paper. Tingting ZHOU, Yanfang CHANG, and Lingyun SUN revised and finalized the paper.

Compliance with ethics guidelines

Yunnong CHEN, Yankun ZHEN, Chuning SHI, Jiazhi LI, Liuqing CHEN, Zejian LI, Lingyun SUN, Tingting ZHOU, and Yanfang CHANG declare that they have no conflict of interest.

Data availability

The data that support the findings of this study are openly available in Github at https://github.com/zju-d3/UILM.

References

Aşıroğlu B, Mete BR, Yıldız E, et al., 2019. Automatic HTML code generation from mock-up images using machine learning techniques. Scientific Meeting on Electrical-Electronics & Biomedical Engineering and Computer Science, p.1-4. https://doi.org/10.1109/EBBT.2019.8741736

Behrang F, Reiss SP, Orso A, 2018. GUIFetch: supporting app design and development through GUI search. Proc 5th Int Conf on Mobile Software Engineering and Systems, p.236-246. https://doi.org/10.1145/3197231.3197244

Beltramelli T, 2018. pix2code: generating code from a graphical user interface screenshot. ACM SIGCHI Symp on Engineering Interactive Computing Systems, Article 3. https://doi.org/10.1145/3220134.3220135

Bunian S, Li K, Jemmali C, et al., 2021. VINS: visual search for mobile user interface design. CHI Conf on Human Factors in Computing Systems, Article 423. https://doi.org/10.1145/3411764.3445762

Cai ZW, Vasconcelos N, 2018. Cascade R-CNN: delving into high quality object detection. IEEE/CVF Conf on Computer Vision and Pattern Recognition, p.6154-6162. https://doi.org/10.1109/CVPR.2018.00644

Chen CY, Su T, Meng GZ, et al., 2018. From UI design image to GUI skeleton: a neural machine translator to bootstrap mobile GUI implementation. Proc 40th Int Conf on Software Engineering, p.665-676. https://doi.org/10.1145/3180155.3180240

Chen CY, Feng SD, Xing ZC, et al., 2019. Gallery D.C.: design search and knowledge discovery through auto-created GUI component gallery. Proc ACM on Human-Computer Interaction, Article 180. https://doi.org/10.1145/3359282

Chen JS, Xie ML, Xing ZC, et al., 2020. Object detection for graphical user interface: old fashioned or deep learning or a combination? Proc 28th ACM Joint Meeting on European Software Engineering Conf and Symp on the Foundations of Software Engineering, p.1202-1214. https://doi.org/10.1145/3368089.3409691

Chen K, Wang JQ, Pang JM, et al., 2019. MMDetection: open MMLab detection toolbox and benchmark. https://arxiv.org/abs/1906.07155v1

Chen S, Fan LL, Su T, et al., 2019. Automated cross-platform GUI code generation for mobile apps. Proc IEEE 1st Int Workshop on Artificial Intelligence for Mobile, p.13-16. https://doi.org/10.1109/AI4Mobile.2019.8672718

Deka B, Huang ZF, Franzen C, et al., 2017. Rico: a mobile app dataset for building data-driven design applications. Proc 30th Annual ACM Symp on User Interface Software and Technology, p.845-854. https://doi.org/10.1145/3126594.3126651

Fay MP, Proschan MA, 2010. Wilcoxon-Mann-Whitney or t-test? On assumptions for hypothesis tests and multiple interpretations of decision rules. *Stat Surv*, 4:1-39. https://doi.org/10.1214/09-SS051

Feng SD, Ma SY, Yu JZ, et al., 2021. Auto-icon: an automated code generation tool for icon designs assisting in UI development. Proc 26th Int Conf on Intelligent User Interfaces, p.59-69. https://doi.org/10.1145/3397481.3450671

Ge XF, 2019. Android GUI search using hand-drawn sketches. Proc IEEE/ACM 41st Int Conf on Software Engineering: Companion Proc, p.141-143. https://doi.org/10.1109/ICSE-Companion.2019.00060

Halbe A, Joshi AR, 2015. A novel approach to HTML page creation using neural network. *Proc Comput Sci*, 45:197-204. https://doi.org/10.1016/j.procs.2015.03.122

He KM, Zhang XY, Ren SQ, et al., 2016. Deep residual learning for image recognition. IEEE Conf on Computer Vision and Pattern Recognition, p.770-778. https://doi.org/10.1109/CVPR.2016.90

Jain V, Agrawal P, Banga S, et al., 2019. Sketch2Code: transformation of sketches to UI in real-time using deep neural network. https://arxiv.org/abs/1910.08930

Li G, Baechler G, Tragut M, et al., 2022. Learning to denoise raw mobile UI layouts for improving datasets at scale. CHI Conf on Human Factors in Computing Systems, Article 67. https://doi.org/10.1145/3491102.3502042

Lin TY, Maire M, Belongie S, et al., 2014. Microsoft COCO: common objects in context. Proc 13th European Conf on Computer Vision, p.740-755. https://doi.org/10.1007/978-3-319-10602-1_48

Lin TY, Dollár P, Girshick R, et al., 2017. Feature pyramid networks for object detection. IEEE Conf on Computer Vision and Pattern Recognition, p.936-944. https://doi.org/10.1109/CVPR.2017.106

Liu Z, Chen CY, Wang JJ, et al., 2020. Owl eyes: spotting UI display issues via visual understanding. Proc 35th IEEE/ACM Int Conf on Automated Software Engineering, p.398-409. https://doi.org/10.1145/3324884.3416547

Liu Z, Chen CY, Wang JJ, et al., 2023. Nighthawk: fully automated localizing UI display issues via visual understanding. *IEEE Trans Soft Eng*, 49(1):403-418. https://doi.org/10.1109/TSE.2022.3150876

Moran K, Bernal-Cárdenas C, Curcio M, et al., 2020. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Trans Soft Eng*, 46(2):196-221. https://doi.org/10.1109/TSE.2018.2844788

Nguyen TA, Csallner C, 2015. Reverse engineering mobile application user interfaces with REMAUI. Proc 30th IEEE/ACM Int Conf on Automated Software Engineering, p.248-259. https://doi.org/10.1109/ASE.2015.32

Ren SQ, He KM, Girshick RB, et al., 2017. Faster R-CNN: towards real-time object detection with region proposal networks. *IEEE Trans Patt Anal Mach Intell*, 39(6):1137-1149. https://doi.org/10.1109/TPAMI.2016.2577031

Suleri S, Pandian VPS, Shishkovets S, et al., 2019. Eve: a sketch-based software prototyping workbench. CHI Conf on Human Factors in Computing Systems, Article LBW1410. https://doi.org/10.1145/3290607.3312994

Vu T, Jang H, Pham TX, et al., 2019. Cascade RPN: delving into high-quality region proposal network with adaptive convolution. Proc 33rd Conf on Neural Information Processing Systems, p.1430-1440.

White TD, Fraser G, Brown GJ, 2019. Improving random GUI testing with image-based widget detection. Proc 28th ACM SIGSOFT Int Symp on Software Testing and Analysis, p.307-317. https://doi.org/10.1145/3293882.3330551

Xu N, Price B, Cohen S, et al., 2017. Deep image matting. IEEE Conf on Computer Vision and Pattern Recognition, p.311-320. https://doi.org/10.1109/CVPR.2017.41

Zhang XY, de Greef L, Swearngin A, et al., 2021. Screen recognition: creating accessibility metadata for mobile applications from pixels. CHI Conf on Human Factors in Computing Systems, Article 275. https://doi.org/10.1145/3411764.3445186

Zhao TM, Chen CY, Liu YM, et al., 2021. GUIGAN: learning to generate GUI designs using generative adversarial networks. Proc 43rd IEEE/ACM Int Conf on Software Engineering, p.748-760. https://doi.org/10.1109/ICSE43902.2021.00074