A Model for Automatic Code Generation from High Fidelity Graphical User Interface Mockups using Deep Learning Techniques

Michel Samir, Ahmed Elsayed, Mohamed I. Marie
Department of Information Systems-Faculty of Computers and Artificial Intelligence, Helwan University, Cairo, Egypt

***Abstract*—Graphical user interface (GUI) is the most prevalent type of user interfaces (UI) due to its visual nature, which allows direct manipulation and interaction with the software. Mockup-based design is a frequently used workflow for constructing GUI. In this workflow, the anticipated UI design process typically progresses through multiple steps, culminating in the creation of a higher fidelity mockup and subsequent implementation of that mockup into code. The design process involves repeating those multiple steps because of the ongoing changes in requirements, which can make the process tedious and necessitate modifications to the GUI code. Additionally, the process of implementing and converting a design into GUI code itself is laborious and time-consuming task that can prevent developers from dedicating the bulk of their time implementing the software's functionality and logic, making it a costly endeavor. Automating the code generation process using GUI design images can be a solution to mitigate these issues and allow more time to be allocated towards building the application's functionality. In this research paper, deep learning object detectors are employed to detect the predominant UI elements and their spatial arrangement in a high-fidelity UI mockup image. This approach generates an intermediate representation, including the layout hierarchy of the user interface leading to the automation of the front-end code generation process for the mockup. The proposed approach demonstrates its effectiveness through experimental results, achieving a recognition mean average precision (mAP) of 91.37% for atomic elements and 87.40% for container elements in the mockup image. Additionally, similarity metrics are employed to assess the visual resemblance between the generated mockups and the original ones.**

Keywords*—Code generation; graphical user interfaces; deep learning; computer vision; mockups

I. Introduction

In an interactive software, there are user interfaces (UIs) which are used by users to communicate with the system and to operate the system's functionalities. The most popular form of UI is graphical user interface (GUI) because of its visual nature which allows direct manipulation of the software. The development of GUIs for apps is often a manual and time-consuming task. Based on a survey [1] conducted among over 5,700 developers, around 51% reported working on app UI design tasks on a daily basis, more than other development tasks, which they tended to perform every few days. Another study revealed that an average of 45% of the code size of software is relevant to the user interface and that the average time spent on the user interface portion is nearly 50% during the implementation phase [2].

A common workflow for building user interfaces is mockup-based design [3]. In this approach, a graphic designer creates a rough illustration of the anticipated UI design. Ideally, design process need to go through several steps. It often starts as a digital or sketched wireframe [4]. A wireframe is a document which outlines the basic structure of the application. A wireframe does not define specific details such as colors. After a wireframe is created, it is refined and more detail is added i.e. it becomes a higher fidelity mockup [5]. After finalizing the design, the implementation of that design starts. Finally, that prototype should be evaluated to check its usability and to discover design problems. Those steps are repeated until the prototype considered satisfactory. With continuous changes in the requirements, this whole design process becomes monotonous and the GUI code needs to be modified accordingly.

This process of implementing client-side software based on a GUI mockup created by designers is the responsibility of developers. Implementing and converting a design into GUI code is time-consuming for the developer and prevent developers from dedicating the majority of their time implementing the actual functionality and logic of the software and therefore costly. Moreover, considering the complexity of UI, generating the GUI code from mockups requires extensive experience as extracting visible elements and their relationship, selecting proper widgets from diversity of UI components, and generating source code are error-prone task. One more problem associated with generating front-end code from GUI image is that computer languages used to implement such GUIs are specific to each target runtime system; thus resulting in tedious and repetitive work when the software being built is expected to run on multiple platforms using native technologies [10].

To cut down these problems, and to invest time in building the actual functionality of the application, front-end code automation is required. Basically, developers have to visually realize UI elements and their spatial layout in the image, and then translate this knowledge into proper GUI components and their compositions. Automating this visual understanding and translation would be beneficial for bootstrapping GUI implementation. However, it is a challenging task due to the diversity of UI designs and the complexity of GUI code to generate. Understanding mockups in the form of images by a machine is a problem of Computer Vision since it entails a

machine making deductions from mockups, understanding them and extracting logical information from them. Computer Vision has made surpassing progress since its beginning. Deep learning methods may be applicable to this task. Deep Neural Networks (DNN) has been extremely popular with the introduction of Convolutional Neural Networks (CNN) and has shown considerable success over classical techniques when applied to other domains, particularly in vision problems [6, 7, 8].

Detection of objects in UI screenshots is an unusual visual recognition task that requires a distinct solution. In this research paper, a novel approach is introduced for identifying UI elements in high fidelity GUI mockups through the utilization of Deep Learning, as well as generating code automatically. To accomplish this, YOLOv7 [9] object detector models are employed in order to detect atomic and container elements within a UI screenshot. These detectors are trained using a specifically curated dataset of UI mockup images. Subsequently, UI representation object and layout hierarchy are constructed to assist generating cross platform code.

This study makes two primary contributions. Firstly, it proposes a unique approach that separates atomic and container UI elements into distinct models, resulting in enhanced detection accuracy. Secondly, it involves the creation of a data preprocessing pipeline specifically designed to overcome the limitations found in the semantic dataset. This research paper sticks to mockups rather than hand-drawn wireframes as there is no universally agreed-upon standard for wireframe symbols and they may not provide the level of precision and consistency required for complex UI designs.

The rest of the paper is organized as follows. The background is illustrated in Section II, followed by the related works in Section III. The dataset and data preprocessing pipeline are discussed in Section IV, followed by the research methodology in Section V. The evaluation is illustrated in Section VI. Section VII provides a discussion that compares the results with existing studies. Section VIII sketches out the future work. Finally, Section IX concludes the paper.

II. BACKGROUND

There is a misunderstanding regarding the meanings of wireframes, mockups, and how they differ from each other. It is important to provide an accurate explanation and distinguish these concepts from one another. The design process can be divided into three stages sequentially, namely wireframes, mockups, and prototypes. While the aforementioned sequence is prevalent and commonly used, it is possible for the design process not to go through all the stages or have minor variations depending on the designer, team, and project. For the purpose of this discussion, the focus will be on wireframes and mockups.

A. Wireframes

A wireframe also known as screen blueprint is a document which outlines the basic structure and layout of a page or screen when referred to applications that demonstrates what interface elements will exist on key pages. A wireframe is regarded as a low fidelity design document due to its simplicity and lack of visual styles and branding elements. Additionally, it does not provide specific details, such as colors, images or even right content. Furthermore, its purpose is to offer a basic visual understanding of a page at the beginning of a project to obtain approval from stakeholders and the team before commencing the creative phase.

Wireframes can be classified into two categories: digital or hand-drawn wireframes. Hand-drawn wireframe, also known as sketch, is useful for early design stages and rapid iterations. It helps designers to quickly visualize rough ideas, create an initial model for the overall layout in a basic format. On the other hand, digital wireframe is more detailed but yet simple. It is usually created using digital wireframing tools. While it still does not include specific components like images or full text, it provides much more detail than its Hand-drawn counterpart as shown in Fig. 1.

Despite the availability of digital wireframing tools, most designers tend to begin by sketching on paper with a pen (Hand-drawn wireframe). This is because designers usually possess an art background and may feel limited by digital tools. Although there is no universally agreed-upon standard, wireframe sketches generally use a similar group of symbols that have commonly understood meanings. Fig. 2 illustrates some of these elements.

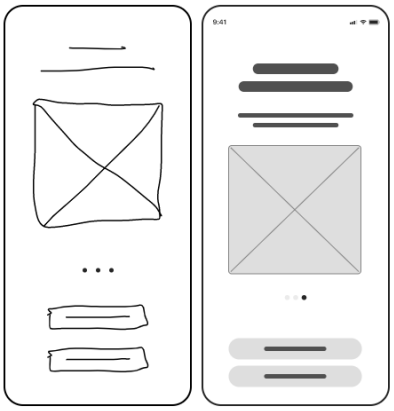

Fig. 1. The difference between Hand-drawn wireframe (a) and Digital wireframe (b).

Fig. 2. Examples of elements commonly used to represent UI elements in wireframes.

B. Mockups

A mockup is a high fidelity design document that is the most detailed and closest to the actual end product design and similar in nature to an app GUI screenshot. It proposes the final look of the design and is usually built between wireframing and prototyping. Wireframes are designed to represent the structure and functional requirements, which are then featured in mockups. Therefore, mockups are essentially wireframes with visual design, such as images, colors, and typography. Fig. 3 shows the difference between wireframe and mockup.

Additionally, those mentioned concepts can be classified in another way. They can be classified into three different levels: (1) low-fidelity, which resembles hand-drawn wireframes and outlines the basic structure of a page, (2) mid-fidelity, which resembles digital wireframes and is the start of mocking up the actual interface, and (3) high-fidelity, essentially mockups with high-quality visuals and contents.

When it comes to wireframes and mockups, designers have different practices and preferences, they can: (1) start with hand-drawn wireframes and then immediately craft mockups, (2) start with digital wireframes and then convert them into mockups, or (3) start with hand-drawn wireframes, convert them to a digital format and then to mockups. After completing the final design document, designers pass their work on to front-end developers for implementing it into code. Implementing user interfaces involves re-creating in code what the designers created graphically in a software. Although developers typically prioritize implementing core functionalities, they often end up spending a significant amount of time coding user interfaces.

III. Related Work

Recently, there has been a growing interest in the use of deep learning and computer vision techniques to automatically generate UI code, which is a relatively new field of research. This section provides a review of the existing techniques and approaches that uses deep learning and computer vision to classify UI components in mockups presented as images. In this section, the attention will be directed towards the relevant studies that specifically concentrate on mockups and screenshots.

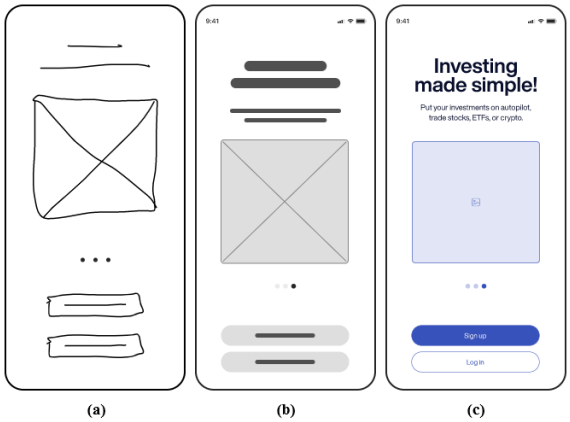

Fig. 3. The difference between Hand-drawn wireframe (a) and Digital wireframe (b) and Mockup (c).

Authors in [10] proposed an application, called Pix2code that transforms high-fidelity GUI screenshots created by designers into computer code. This application utilizes a Deep Learning framework to convert GUI images into their corresponding code for three different platforms, namely web-based, Android and iOS. The pix2code dataset is constructed by mapping bootstrap-based websites into Domain-specific language (DSL) consisting of 18 vocabulary tokens that describe websites layout and components. The dataset comprises 3,500 pairs of websites GUI images and their corresponding markup which is in DSL code. The main idea behind Pix2code is to train a model to learn the mapping between a GUI screenshot and the corresponding code that generates the GUI. The model relies on two main components. First, a Convolutional Neural Network is used to perform unsupervised feature learning on the GUI image. Second, a Recurrent Neural Network (RNN) is used to perform language modeling on the DSL code associated with the input GUI image.

In Pix2code, a three-step approach is required to solve the problem. First, a CNN-based image encoder is used to extract high-level visual features from GUI screenshot. These features are then passed through a fully connected layer to generate a fixed-length feature vector, which represents the input image. Second, long short-term memory (LSTM) network is used which is a type of RNN architecture. The LSTM network is trained to perform language modeling on the DSL code associated with the input GUI image. As a result of this training, the LSTM network gains an understanding of the syntax and semantics of the source code, which enables it to generate a language-encoded vector. This vector is a sequence of one-hot encoded tokens that correspond to the DSL code. Third, LSTM-based code decoder is used. Vectors from the previous two steps are concatenated and then fed into this decoder, which is able to generate high-quality code that accurately reflects the layout and components of the input GUI image. This LSTM decoder is trained to learn the relationship between objects present in the input GUI image and the associated tokens present in the DSL code.

While Pix2code performs well with simple datasets, it struggles with complex datasets containing numerous code tokens. To address this limitation, a novel front-end code generation approach is proposed [11], which utilizes multiple heads of attention to examine the feature vectors of GUI screenshots. This technique enables the analysis of the feature vectors, generation of code tokens, and seamless integration of the analysis and generation processes.

In the cited study [12], the approach is divided into three main components: (1) object detection, (2) text recognition, and (3) code generation. The process involves inputting a GUI image and running parallel modules for image processing, deep learning, and text detection and recognition. The GUI elements are detected using a fusion of deep neural network and traditional image processing techniques, followed by integrating the results from the text detection and recognition module. The detection results are then used to generate corresponding codes using a parser. Another study proposed in [13], employed a Deep Learning (DL) approach to design a system for generating GUI code for websites. A dataset

containing the coordinate, width, height, and type information of GUI objects is curated using 7500 webpages. This dataset is then utilized in the proposed system to detect objects within GUI images and generate DSL mark-up code.

Nguyen et al. [14] was the first to propose the technology of automatic reverse engineering of mobile application user interface (REMAUI). By analyzing screenshots of a mobile application's user interface, REMAUI detects the presence of different components, such as buttons, textboxes, and pictures, and generates their corresponding code. Their study was the first to utilize computer vision and optical character recognition techniques in addition to mobile specific heuristics to enable conversion of screen images into code for mobile platforms. This method not only translates the structure, but also the style (images, colors, fonts) of the designs. The REMAUI method works successfully, but its potential is limited by the time-consuming process needed to adapt techniques for identifying new elements.

Moran et al. [15] proposed ReDraw based on REMAUI. ReDraw is an algorithm that takes mockups of mobile application screens and generates structured XML code for them. The paper outlines a three-stage approach to automate the conversion of GUI designs to code, which involves the following steps: (1) Detection, (2) Classification, and (3) Assembly. The initial stage of their approach involves utilizing computer vision techniques to identify the individual components of the GUI. In the second stage, the identified components are classified based on their functionality, such as toggle-button, text-area, etc. This is achieved through the use of CNN. In the final stage, the XML code is generated by combining the results of the previous stages with the K-nearest neighbor (KNN) algorithm, which organizes the code based on web programming hierarchy. It is worth noting that the authors of this paper have also contributed to the development of a dataset. The dataset includes 14,382 GUI images with a total of 191,300 annotated GUI segments. These segments encompass 15 different classifications, including RadioButton, ProgressBar, Switch, Button, and Checkbox. The aforementioned CNN model relies on this dataset for training and evaluation purposes.

A framework proposed in [16] takes UI pages as input and generates the corresponding GUI code for Android or iOS as output. The authors first utilize traditional image processing techniques, such as edge detection, to identify the location of UI elements. They then employ CNN-based classification to determine the semantics of the UI elements, such as their type. The proposed framework consists of three phases, namely component identification, component type mapping, and GUI code generation. Component identification involves extracting components from the UI pages using image processing techniques, followed by identifying the component types (such as Button or TextView) using a deep learning algorithm based on CNN classification. Component type mapping maps the identified component types to their corresponding components in the target platform. GUI code generation generates the final GUI implementation code based on the component types and their attributes obtained from the previous two phases. The critical phase in this framework is the component type mapping, which employs a large map to generate the final code based on heuristic rules.

UIED is a GUI element detection toolkit [17] that was introduced in 2020. Using an image-based approach, it provides users with a platform for detecting GUI elements. The toolkit offers a web interface that enables users to upload their GUIs, and the system automatically detects and identifies the elements within them. In the approach proposed by [17], the detection task is split into two parts: (1) non-text element detection and (2) text detection. To extract non-text regions, traditional computer vision algorithms are utilized, while deep learning models are employed for classification and text detection. To detect non-text elements, the approach utilizes the Flood-Fill and Sklansky algorithms to identify potential layout blocks. The image is then subjected to edge detection and converted into a binary map form. The binary map is segmented into block segments based on the previously detected blocks, and the connected component labelling algorithm is used to detect GUI elements within each block. The detected elements are then classified using a ResNet-50, which was trained on a dataset of 90,000 GUI elements divided into 15 classes. To detect text, the approach utilizes the advanced EAST OCR, which is a deep learning-based scene text detector that can accurately identify text within the screenshot image.

Screen Recognition [18] is a system that generates metadata describing UI components from a single GUI image. This metadata is then forwarded to iOS VoiceOver, which enhances accessibility. The system is optimized for mobile devices, ensuring that it is both memory-efficient and fast. To achieve this, it utilizes deep learning techniques trained on an iPhone application dataset. The authors created a dataset of GUIs from thousands of iPhone applications by manually downloading the top 200 most popular applications from each of the 23 categories (excluding games). They then gathered screenshots of visited UIs and their metadata (tree structure, properties of UI elements), but the data was incomplete, so manual annotation was required. Ultimately, 40 individuals annotated all UI elements in the collected screenshots using bounding boxes and identifiers, resulting in a dataset of 77,637 annotated UI screens. The UI detection model is designed to extract elements from a GUI and classify them accordingly. To achieve this, the solution employs an SSD model with a MobileNetV1 backbone. After the inference, the output is post-processed to eliminate extraneous detections, and the built-in OCR service is utilized to identify any missing elements. However, since the detector generates separate bounding boxes for each element, the UI elements need to be grouped. This is accomplished using hard-coded heuristics that were empirically acquired from 300 randomly selected samples.

IV. Data Preprocessing Pipeline

Before presenting the proposed methodology and exploring it in details, a dataset is established that comprises clean UI annotations based on an existing mobile UI corpus. This section introduces a data preprocessing pipeline specifically designed to overcome challenges and problems associated with the UI corpus in order to produce a polished and clean dataset. This pipeline not only helps overcome UI corpus challenges

but also plays a crucial role in converting raw data instance within the dataset into a format that is compatible with the proposed methodology. In this section, a detailed description is provided of the dataset creation process and outlines the steps involved in the data preprocessing pipeline.

A. Mobile UI Corpus

The research experimental dataset is constructed by leveraging the open sourced Rico [19] dataset. The Rico dataset stands out as the most extensive public collection of mobile GUIs. It comprises 66,261 distinct GUI screens obtained from over 9.7k free Android applications spanning 27 diverse categories. Each example within this large-scale dataset consists of a screenshot and its corresponding view hierarchy metadata. A view hierarchy represents a tree structure of the UI layout wherein each node corresponds to an element within the UI. Each node encompasses a range of properties, including the UI element's position, its Android class, and various attributes that define the element.

Although the view hierarchy metadata provides specification for UI elements and their layout, a notable issue arises from the fact that the captured view hierarchies often contain enormous number of different element types. This abundance of different types poses challenges for training deep learning models and can potentially adversely affect their performance. Additionally, the view hierarchy metadata may include elements with overly generic types like View, WebView, as well as elements with custom types such as custom views or views from third-party packages. Consequently, this lack of specificity in element types hampers the conveyance of meaningful semantic information about the UI components displayed on the screen.

To address this, Liu et al. [20] suggest a method for generating semantic annotations where semantic types are assigned to the UI elements of the Rico view hierarchies. These annotations are applied on each screenshot in Rico dataset, enabling the identification of elements present in the UI along with their associated view hierarchy as a tree. 25 types of UI elements are defined in these sematic annotations, including TEXT, IMAGE, DRAWER, BUTTON, and more. However, the generated annotations are still noisy and not suitable for the purpose of comprehending GUIs. In this paper, these semantic annotations, which is in JSON format, and its corresponding screenshots obtained from the Rico dataset are referred to as the semantic dataset.

B. Semantic Dataset Limitations

In this section, the objective is to highlight the limitations identified in the semantic dataset, with the aim of obtaining a clean UI dataset that improves the performance of the proposed model. The primary concern lies within the UI elements themselves. One issue arises when the JSON annotation contains bounding boxes of an element that do not have visual correspondences on the corresponding screenshot. Another issue involves misaligned elements where bounding boxes partially cover other elements. An additional issue arises with elements that are extremely small, resulting in a zero area due to the element's boundary box having zero values for both width and height.

Another primary concern revolves around the incorrect semantic annotations assigned to UI elements. For instance, an ON-OFF SWITCH element being mistakenly labeled as an INPUT element. In addition to incorrect labeling of certain UI elements in the screenshots, there are cases where entire screenshots are inaccurately labeled, as if the annotations belong to an entirely different screenshot as shown in Fig. 4. Another significant concern emerges when the annotation JSON contains different semantic types that share the same bounding boxes. This creates a problem in determining which type among them is the correct one to consider for that particular boundary box.

Another observed issue is the presence of elements that are repeated multiple times in a screenshot, following a pattern such as items in a list or grid. While these elements may have similar shapes and structures, they are assigned different semantic types. For instance, in a list arrangement, some elements are labeled as ICON type while others are labeled as IMAGE type, despite all of them having the same shape. There is an additional concern regarding DRAWER and MODAL types, which are regarded as containers. The problem revolves around identifying UI elements that are contained within these types, as well as distinguishing elements that are not part of them, even if their boundary boxes overlap with both.

The most recent and significant issue observed is that certain UI elements are semantically labeled based on their functionality. However, visually, these elements should be categorized under a different UI type due to their shape and resemblance to that type. For instance, there are UI elements labeled as RADIO_BUTTON based on their functionality, but visually, they closely resemble the BUTTON type on the screenshot as shown in Fig. 5. This issue has the potential to significantly challenge the model and impact its performance. In order to construct the experimental dataset, limitations and issues highlighted above with the semantic dataset should be addressed through a data preprocessing pipeline.

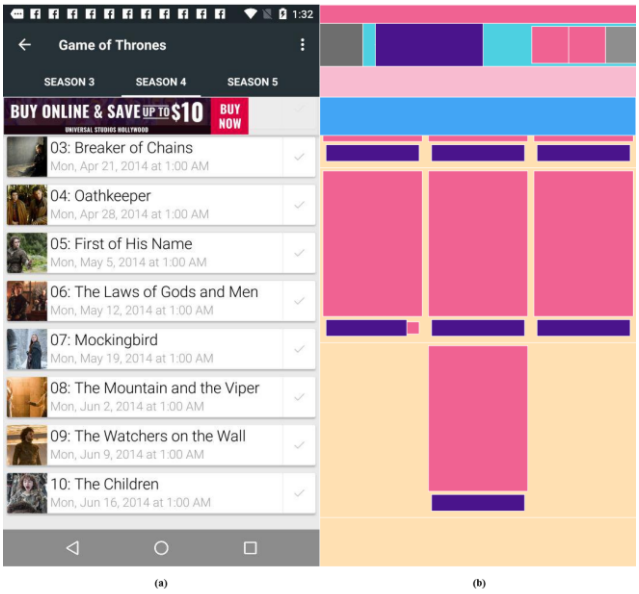

Fig. 4. Entire screenshot (a) are inaccurately labeled in semantic dataset (b).

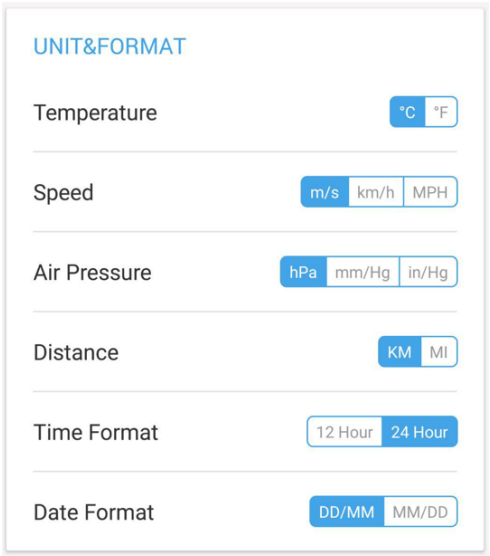

Fig. 5. Radio buttons that closely resemble the Button type.

C. Data Preprocessing Pipeline

The data preprocessing pipeline comprises four phases, as shown in Fig. 6: (1) neglecting phase, (2) extraction phase, (3) selection phase, and (4) formatting phase. In the initial stage, known as the neglecting phase, any incorrect data instances in the dataset are discarded. Data instances in the dataset are neglected if the entire screenshots have incorrect labels. Additionally, data instances are neglected if the screenshots do not come from an application and only consist of an Android launcher.

During the second phase, the objective is to extract all UI elements that are presented in the annotation JSON for each screenshot. To accomplish this, a Depth-First Search (DFS) algorithm is employed using recursion to traverse the annotation's tree for each screen. The outcome of this phase is the generation of a file for each screenshot, where each file contains a Python dictionary comprising all the extracted final UI elements. While executing this phase, UI elements with zero area are disregarded. In cases where multiple UI elements share the same boundary box, the last UI element visited during the pre-order traversal is retained and its semantic type is considered as the appropriate choice for that boundary box neglecting other elements that share the same boundary box.

During execution and when encountering a node in the JSON tree with the DRAWER or MODAL types, these types are treated as the parent node and added to a STACK. This signifies that all the visited children (UI elements), until reaching the parent node again, are contained within this type. Subsequently, the parent node's type is removed from the stack. All these UI elements associated with the parent node are stored in a separate list. Next, a check is performed to determine if there is any overlay (IOU) between any other UI element found in the annotation JSON and the parent type (DRAWER or MODAL). If the overlay exceeds 20%, the element is removed as it is considered to be hidden under the parent type (DRAWER or MODAL), as observed through Trial and error.

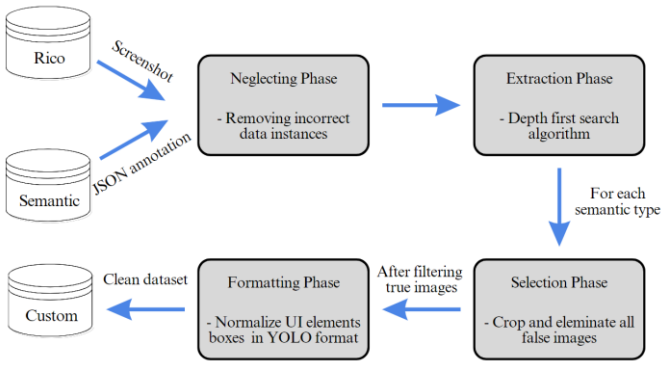

Fig. 6. Data preprocessing pipeline.

In the selection phase, the goal is to discern and filter the most suitable UI images for each semantic type, while excluding the incorrect ones. For every semantic type such as TOOLBAR, DRAWER, and others, a corresponding folder is generated to store all the images related to that semantic type. To accomplish this, the output of the preceding phase is utilized, which includes a generated file for each screenshot containing the extracted UI elements. For each file, each UI element is extracted based on its boundary box by cropping it from the image, and then place it in the folder that corresponds to its semantic type. The outcome is a set of folders, each named after one of the semantic types, and each folder contains images of UI element specific to that semantic type.

Two checks on the images are performed within each folder. Firstly, any image that has been labeled incorrectly is identified and should actually belong to a different type. Secondly, we prioritize retaining the standard shapes associated with each type, while eliminating UI elements that might have similar functionality but visually belong to a different semantic type. Based on these checks, all false images are eliminated/deleted, resulting in filtered folders that exclusively contain the visually best images of their respective UI elements.

The final phase is the formatting phase, where the boundary boxes of the UI elements are normalized. Moreover, each UI type is encoded with a predefined number to ensure compatibility with the YOLO format. The purpose of the formatting phase is to prepare the dataset in a suitable format to be used as input for the proposed model. To achieve this, the generated files obtained from the extraction phase are utilized. We iterate through each UI element in each file and verify if it is still present in the corresponding folder of its semantic type. If it is, the UI element is considered valid. Its boundary box, alongside its corresponding UI type, is saved in a text file using the YOLO format. Conversely, if the UI element is not found in the designated folder, it is neglected and excluded from further processing.

By employing the YOLO labeling format, this phase yields the creation of a text file for each screenshot, mirroring their respective names. Each text file contains separate lines, with each line presenting the details of a single UI element, including its boundary box and type. The boundary box and type for each UI element are described using specific representations. The bounding box is denoted by four values:

x_center, y_center, width, and height. The x_center and y_center represent the normalized coordinates of the bounding box's center. To achieve normalization, the pixel values of x and y, corresponding to the center of the bounding box on the x- and y-axis, are divided by the width and height of the image, respectively. The width and height values indicate the dimensions of the bounding box, and they are also normalized. In relation to the UI type, all semantic types are assigned numerical encodings. Each number corresponds to a specific UI type.

Finally, each UI element is represented in the YOLO format as a line, consisting of the encoded UI type known as class, normalized x_center, normalized y_center, normalized width, and normalized height. The outcome of this phase is our custom dataset in which each data instance includes a UI screenshot along with its corresponding text file, providing descriptions of the UI objects present in the screenshot in YOLO format.

D. Semantic Types

In this research, the primary emphasis lies on specific 20 semantic types, from those outlined by Liu et al. [20] in their semantic dataset. However, modifications are made by introducing new semantic types that are described below. The intention behind introducing these new types is to prioritize the visual aspects of the elements, enabling us to accurately translate these UI elements into corresponding code widgets.

Before introducing new semantic types, the WEB VIEW type is excluded because it does not qualify as a standalone semantic type. WEB VIEW refers to web content that is displayed within a mobile application, encompassing various UI elements that are not individually labeled. This research opted to exclude the VIDEO type and instead categorized them as IMAGE type since we consider them to be indistinguishable on static screenshots. Additionally, based on the same rationale, the ADVERTISEMENT type is excluded and classified as an IMAGE type. This decision is supported by the fact that in the code, the same image cannot be selected to be displayed as an advertisement, as it is a real-time process. We differentiate between the IMAGE and ICON types. IMAGE is reserved for real images that depict tangible objects, which can be captured by sensors. On the other hand, ICON is used to represent vector graphics images and logos.

In contrast, additional UI types are also introduced, including BOTTOM_SHEET, SPINNER, and PROGRESS_BAR. Within the RICO dataset, there are numerous screenshots that feature progress bars, even though they are not specifically classified as a type in the semantic dataset. To address this, an analysis was conducted by inspecting the nodes in the view hierarchy that contained an Android class named ProgressBar. In relation to BOTTOM_SHEET, the investigation of the DRAWER type revealed that bottom sheets are classified along with drawers. Drawers are side-bar menus that display an application's primary navigation options and can be toggled to open or close. On the other hand, bottom sheets are surfaces that contain supplementary content and are anchored to the bottom of the screen. We decided to categorize them separately because we perceived significant visual distinctions that warranted the creation of new class. Moreover, from a coding perspective, these elements require the implementation of entirely different widgets. Similarly, a similar situation was encountered with the SPINNER type. Initially, it was grouped under the MODAL type. However, as modals represent pop-up windows or dialogs, and spinners are drop-down menus, we decided to separate them due to the same rationale applied to the DRAWER and BOTTOM_SHEET types.

In total, a set of 23 semantic types has been established, encompassing BOTTOM_NAVIGATION, BUTTON_BAR, CARD, CHECKBOX, DATE_PICKER, DRAWER, ICON, IMAGE, INPUT, LIST_ITEM, MAP_VIEW, MODAL, MULTI-TAB, ON/OFF_SWITCH, PAGER_INDICATOR, RADIO_BUTTON, SLIDER, TEXT, BUTTON, TOOLBAR, SPINNER, PROGRESS_BAR, and BOTTOM_SHEET.

V. Research Methodology

Our methodology involves a five-phase pipeline that takes a high fidelity mockup image as input and generates a cross-platform application in real-time as the output. There are five phases involved in our methodology: (1) Model preparation, (2) Object detection, (3) Element post-processing, (4) Construction of the layout hierarchy, and (5) Code generation. The overall architecture of the proposed methodology is illustrated in Fig. 7. As depicted, the process utilizes pre-trained models to expedite training and enhance overall performance. Subsequently, our custom datasets are employed to fine-tune the pre-trained models and tailor them specifically to the desired domain. The training process for these custom models is a one-time occurrence. Once the custom models have been trained, they are employed solely for the purpose of detecting UI elements in the input mockup image.

This paper adopts a DNN approach for object detection. Object detection involves the classification and localization of various objects within an image. It encompasses the assignment of appropriate labels to each object and the creation of bounding boxes around them to enhance recognition. Object detection not only informs us about the presence of specific objects in an image, but also provides information about their spatial location. To locate the UI elements within the mockup images, the YOLOv7 real-time object detection model was utilized. YOLO, also known as You Only Look Once, is a deep learning model that has undergone several iterations to become a powerful solution for real-time object detection and localization. It falls under the category of one-stage detectors, offering fast inference speeds. In this section, the research paper will delve into the five-phase pipeline, providing a comprehensive and detailed explanation.

A. Model Preparation

To enhance the efficiency of YOLOv7, two aspects need to be tackled: (1) dataset-related concerns and (2) hyperparameters of the YOLO model. The first aspect involves addressing two areas: (1) balancing the dataset, and (2) improving dataset quality. The second aspect focuses on the selection of anchor boxes.

Starting with the first aspect, balancing a dataset is crucial because imbalanced datasets pose challenges for predictive

modeling. By achieving balance, we ensure that the model does not exhibit bias towards a particular class. To illustrate this point, let's consider the outcome of the selection phase. If the number of UI images in the TEXT folder is compared to the number in the DATE_PICKER folder, a significant class imbalance is observed, with a ratio of 1:1455. This stark contrast in the number of instances for each class highlights the pronounced imbalance within the semantic dataset.

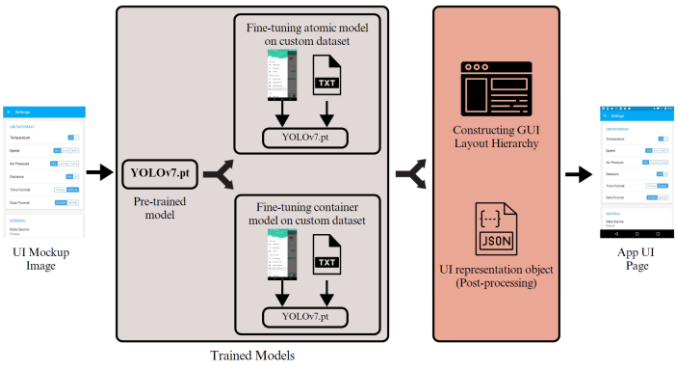

Fig. 7. The overall architecture of the proposed methodology.

In order to address this concern and achieve a balanced dataset, a technique that involves selecting a portion of our custom dataset has been applied in a manner that ensures a more even distribution of instances among all the classes. In this technique, a consistent quantity of screenshots will be allocated to each class (semantic type) in order to ensure that each class is represented in the dataset, particularly for classes with a small number of instances. It is essential that the screenshots selected for a specific class encompass instances that pertain to that class. Assigning a consistent number of screenshots to each class does not guarantee an equal number of instances, as a single screenshot may contain multiple instances of the same class and instances from other classes as well. By adopting this approach, we are able to regulate the quantity of screenshots chosen for each class and consequently the overall number of screenshots. This not only guarantees a minimum number of instances for each class but also sets an upper limit for classes with a large number of instances. As a result, it promotes a more balanced distribution of instances across the classes. This technique is utilized to create all the future datasets from the custom dataset specified in Section IV, which will subsequently be employed with YOLO models.

Furthermore, the utilization of class weights is a prevalent technique employed to tackle class imbalance within a dataset. These weights determine the relative significance of each class during the training process. In this proposed approach, we have incorporated YOLOv7's inverse class frequency weighting, which assigns higher weights to underrepresented classes and lower weights to overrepresented classes based on their inverse frequency within the dataset. As a result, this approach amplifies the importance of less prevalent classes during the training process.

Improving dataset quality is also a crucial concern in the process of training a YOLO model. One recurring issue observed in both the Rico dataset and the semantic dataset is the incomplete labeling of all visual elements present in the screenshot. For instance, while a button may be correctly annotated with the semantic type BUTTON, the accompanying text or icon within the button may not be labeled as shown in Fig. 8.

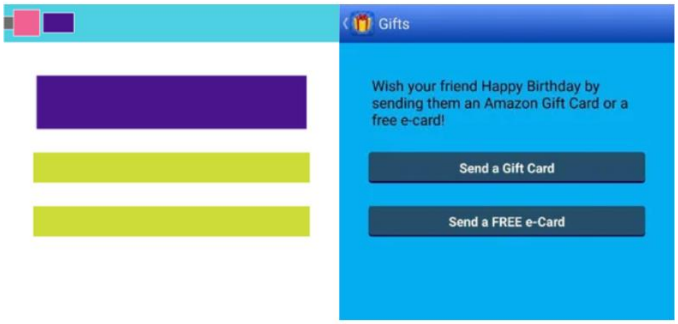

Fig. 8. Incomplete labeling of all visual elements present in the screenshot. The buttons lack proper labeling for their text.

Incomplete labeling for certain classes within the dataset may cause the proposed model to produce false negatives, leading to biased or suboptimal model performance, particularly for the classes with incomplete labeling. The model may struggle to accurately detect and classify instances of these classes resulting in reduced accuracy. In addition, incomplete labeling can lead to a problem in training models because the model might learn incorrect associations from the unlabeled instances of the class. This can result in poor performance when the model is used for prediction on new, unseen data. In order to mitigate this issue, ensuring complete labeling for all classes in the training dataset is essential. Consequently, the necessary step of manually verifying and adding annotations to the unlabeled objects in the screenshots was taken. To accomplish this, Labelimg [21] was utilized, a free, open-source software program written in Python for labeling images that enabled us to thoroughly check and annotate the previously unlabeled objects.

When it comes to the second aspect, which involves adjusting the hyperparameters of the YOLO model, the selection of anchor boxes can significantly enhance efficiency. YOLOv7 is categorized as an anchor-based model. Anchor boxes are predetermined bounding boxes with specific dimensions in terms of height and width. These boxes should be specifically designed to capture the object classes with the scale and aspect ratio that you aim to detect. The general idea is to generate numerous possible bounding boxes initially and then choose the most suitable ones to match the target objects. These selected boxes are then slightly adjusted in terms of position and size to achieve the optimal fit.

The choice of anchor boxes is crucial as YOLO predicts bounding boxes as offsets from these predefined anchors. By selecting optimal anchor boxes, the neural network's workload is reduced, resulting in higher model accuracy. To illustrate the optimal choice of anchor boxes, it is advisable to select anchor boxes that encompass a range of scales and aspect ratios. This ensures a better alignment with the size and shape of the objects being detected. Typically, anchor boxes are selected based on the object sizes found within your training datasets. To achieve this, K-Mean++ clustering algorithm is employed

to generate anchor boxes. This involves grouping the ground truth bounding boxes of UI elements in the training dataset into clusters and utilizing the centroids of these clusters as the anchor boxes, based on the number of anchor sizes that is needed. Fig. 9 illustrates the result of grouping boundary boxes of atomic elements into nine clusters based on their scale, where the centroids of these clusters act as anchor boxes.

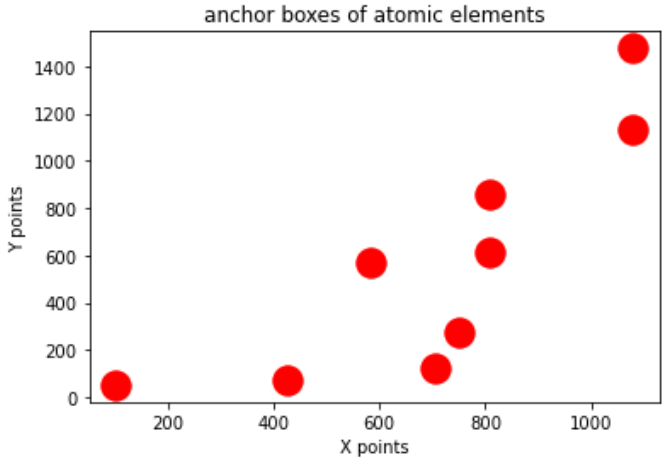

Fig. 9. The anchor boxes are represented by the centroids of atomic elements clusters.

B. Object Detection

Detecting GUI elements in the input mockup image is the essential phase of the proposed methodology. This particular phase consists of two modules, each with its own responsibility for detecting various elements in the mockup. The first module is designed to detect individual atomic elements, while the second module focuses on detecting container elements. Both modules take the mockup image as input and return the detected elements. Atomic elements are fundamental UI components that cannot be further divided and serve as the basic building blocks of an interface, such as checkbox or text elements. On the other hand, container elements are UI components that encompass and contain other UI elements, like toolbars and drawers. They act as visual boundaries or enclosures that primarily group and include atomic UI elements. Fig. 10 provides examples of both atomic and container elements.

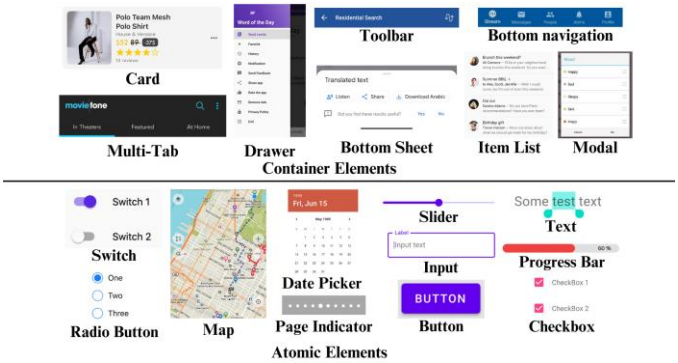

Fig. 10. Examples of both atomic and container elements.

The research paper has separated these types of UI elements into two modules for two reasons. Firstly, it is to address the challenge of handling different object scales. Object detection can be difficult when dealing with objects of varying sizes. One YOLO model may struggle with detecting small objects while performing well on larger ones, and vice versa. By utilizing two YOLO models specifically trained for different object scales, you can enhance the accuracy and reliability of detection. Secondly, by having separate models, you can ensure that training a new class in one model does not interfere with the previously learned classes in the other model. This approach allows you to expand the system's capabilities by adding new classes or new instances for a specific class without affecting the performance of the other model.

To detect atomic and container elements, a separate YOLO model was employed for each module. Each model was trained individually using distinct dataset derived from the custom dataset mentioned in Section IV. The previously mentioned dataset balancing technique was also applied to ensure the datasets were well-distributed. Both atomic and container models were trained for 400 epochs. For each module, the dataset is structured in the YOLO format. This includes a mockup image accompanied by a corresponding text file that describes the UI elements present in the mockup image. The text file contains information such as the object class, object coordinates, height, and width for each UI element. However, only the UI elements relevant to that specific module are retained in the dataset, while the other classes are removed. Initially, pre-trained YOLOv7 models were utilized that underwent training on the COCO dataset. Subsequently, for each module, fine-tuning on the YOLOv7 model was performed using its respective dataset.

In the atomic module, the dataset consists of a total of 1,400 examples. These examples are divided into training and validation sets, with 1,120 examples allocated for training and 280 examples for validation. Similarly, in the container module, the dataset also contains 1,200 examples. These examples are split into 960 for training and 240 for validation. The output of each module is a generated list that contains the detected elements found in mockup image, providing information such as their class labels and corresponding bounding boxes. Finally, the outputs of both modules are combined by concatenating them.

C. Element Post-processing

In order to convert the mockup to code, it is necessary to detect the visual properties of the UI elements such as their sizes, main colors, and more. The previous phase has generated a list of detected UI elements, but this additional phase is required to accurately identify and extract these visual properties. In addition to capturing the visual properties related to style, it is also important to capture the current state of certain UI elements including aspects such as the content displayed, the selection state (e.g., whether an element is selected or not), or the percentage state (e.g., progress or completion percentage).

By employing classical computer vision techniques, essential styling properties can be extracted for each UI element, such as width, height, main color, and background color. Additionally, for some UI elements, there would be other specific properties for them like border-radius for

buttons, number of page indicator circles, and whether a slider is 2-way slider or not and its selected range color. Furthermore, certain UI elements contain dynamic content such as TEXT elements. To extract text from these elements, Optical Character Recognition (OCR) techniques are employed. Specifically, we utilize the open-source Tesseract [22] OCR engine, accessed through the Pytesseract wrapper, which is implemented in Python. This allows us to recognize and extract textual values from regions identified as text by the YOLO model in the mockup image.

On the other hand, there are several UI types that have selection states, such as RADIO_BUTTON, CHECKBOX, PAGE_INDICATOR, and ON/OFF_SWITCH. The proposed dataset, which is used to train the UI detection YOLO model, includes examples of these UI types with their selection states. Some of these types, like radio buttons, checkboxes, and on/off switches, have a true or false selection state. The visual information of these UI elements is utilized to determine whether they are selected or not. The most common visual indicators include the color and the position of certain parts within the UI element itself. For instance, the selection state of a switch can be detected by analyzing the direction of its toggle. The other UI types such as MULTI-TAB, BOTTOM_NAVIGATION, and PAGER_INDICATOR have multiple selection states. Their visual information is utilized to identify the specific item that is selected. The most common visual indicators in this case are the color and size. For example, the selected tab is highlighted with a different color while the other tabs remain unhighlighted. Finally, when there is only one selection, "selected" property is assigned as true if the detected state is selected, otherwise false. In the case of multiple selections, we only record the index of the selected item.

There is another category of UI elements that exhibit a distinct behavior, which is the percentage state. Certain UI types, such as PROGRESS_BAR and SLIDER, always have a selected range. In this case, we have observed that the primary color and length serve as the most prevalent visual indicators. By analyzing the width of the detected element in relation to the length of the selected range, the percentage of the range that is selected can be determined.

Lastly, after completing the post-processing phase, all the determined outcome properties for each UI element are consolidated into a platform-independent UI representation object. This UI representation object is essentially a dictionary consisting of key-value pairs, which effectively represents the recognized UI elements along with their respective properties.

D. Constructing Layout Hierarchy

This phase holds great importance as its objective is to construct the UI layout by aligning the UI elements in a manner similar to the mockup. A novel approach was implemented and is referred to as "UI element grouping". In this approach, once the list of detected UI elements has been obtained from both atomic and container models during the object detection phase, we proceed to conduct an intersection test. This test involves comparing each atomic element with the container elements. If the intersection area between an atomic element and a container element exceeds 90%, the atomic element is considered to be inside the container element.

To initiate the UI element grouping approach, the atomic list and container list obtained from the atomic and container models are utilized as our input. Each UI element within these lists comprises attributes such as class_type_index, area, polygon, boundary_box [XLeft, YTop, XRight, YBottom], and visual properties from the UI representation object. As a result of the UI element grouping, an output in the form of a list is generated representing the layout hierarchy of the mockup. This hierarchy arranges the elements vertically, and we determine whether an element is displayed individually in a row or if it has neighboring elements arranged horizontally within the same row.

To identify the layout structure and determine the positioning of elements relative to each other, a well-defined sequence of steps is followed as outlined in Algorithms 1-5. At first, the YOLO models assign a unique index to each element class, starting from zero and going up to the number of classes minus one, for both atomic and container elements. To prevent any numbering conflicts between the models results, the indexes of atomic elements were adjusted to begin after the indexes of container elements. Subsequently, any inner elements were eliminated, whether they are atomic or container elements. Only the outer elements, which are not contained within any other element, will remain.

Algorithm 1: Adjust Indexes

Input:	List of detected UI elements from the atomic model (a_list) List of detected UI elements from the container model (c_list)
Output:	Indexes of a_list elements will begin after the indexes of c_list elements to avoid numbering conflict

```
1  procedure adjust_ indexes(a_list, c_list)
2      for each element in a_list do
3          class_type_index = class_type_index  + c_list length
4      end for
5      return a_list, c_list
6  end procedure
```

Algorithm 2: Eliminate Inner Elements

Input:	Output of Alg. 1 (a_list, c_list)
Output:	List comprises outer elements only

```
1  procedure eliminate_inner_elements(a_list, c_list)
2      outer_elements_list = a_list + c_list
3      for each element (A) in outer_elements_list do
4          for each other element (B) in outer_elements_list do
5              if area of A > area of B then
6                  if A polygon intersects B polygon > 90% then
7                      remove element B from outer_elements_list
8                  end if
9              end if
10             if area of B > area of A then
11                 if B polygon intersects A polygon > 90% then
12                     Remove element A from outer_elements_list
13                 end if
14             end if
15         end for
16     end for
```

Algorithm 2: Eliminate Inner Elements

```
17        return outer_elements_list
18    end procedure
```

Algorithm 3: Sort Element List

Input:	Output of Alg. 2 (outer_elements_list)
Output:	Sorted list (outer_elements_list)

```
1     procedure sort_list(outer_elements_list)
2         sort outer_elements_list by boundary_box[YTop]
3         return outer_elements_list
4     end procedure
```

Algorithm 4: Element Alignment

Input:	Output of Alg. 3 (outer_elements_list)
Output:	List comprises inner lists, with each inner list representing an element and indicating whether or not it has neighboring elements arranged horizontally within the same row.

```
1     procedure element_alignment(outer_elements_list)
2         let all_elements_adjacents_list as list
3         for each element (A) in outer_elements_list do
4             let adjacents_list as list for element (A)
5             for each other element (B) in outer_elements_list do
6                 if element B is adjacent to element A (Alg. 6) then
7                     add element A and B to adjacents_list if not exist
8                 end if
9             end for
10            add adjacents_list to all_elements_adjacents_list
11        end for
12        return all_elements_adjacents_list
13    end procedure
```

Algorithm 5: Remove Duplicates

Input:	Output of Alg. 4 (all_elements_adjacents_list)
Output:	If an element has no neighboring elements, its element_list will be empty. If it does have neighboring elements, each of those elements will also have their own element_list with the same elements. To avoid redundancy, duplicate lists are removed.

```
1     procedure remove_duplicates (all_elements_adjacents_list)
2         for each element_list in all_elements_adjacents_list do
3             if element_list is empty then
4                 This element has no adjacent elements
5                 Add only this element to element_list
6             end if
7             if element_list has elements (adjacent elements) then
8                 Each adjacent element have list with the same elements
9                 Remove those duplicate lists to that element_list
10            end if
11        end for
12        return all_elements_adjacents_list
13    end procedure
```

Afterward, the list of remaining elements, which includes both containers and atomic elements, is sorted in a top-to-bottom manner based on the Y-top point of each element. The YOLO detection boundary boxes do not ensure that the elements adjacent to each other will have their boundaries starting at the same horizontal line. Therefore, the purpose of sorting is not to arrange all elements vertically beneath each other. It is primarily aimed at detecting horizontal alignment by ordering the elements in a way that elements adjacent to each other appear consecutively in the list.

Afterwards, the alignment algorithm is utilized to identify elements that are positioned next to each other. In this alignment algorithm, every element in the list is compared to all other elements. The outcome is a separate list for each element, which includes any adjacent elements found for that specific element. We accomplish this by creating vertical lines from the y-top point to the y-bottom point for each element, and then comparing these lines with those of all other elements. If the alignment is approximately 90% horizontally, the elements are deemed to be in the same row and adjacent to each other, as illustrated in Algorithm 6. Following this criterion, if an element is compared to others and adjacent elements are found, its list will include these adjacent elements. Conversely, if an element is compared to others and no adjacent elements are found, its list will be empty, indicating that it is the sole element in that particular row.

As the final step of the algorithm, the list for each element is examined. If the list is empty, it indicates that the element has no adjacent elements. If there are adjacent elements present, each element's list will include the other elements. To avoid duplication, we remove any repeated elements, resulting in a single list that contains all the adjacent elements. These processes are repeated for every container element that contains atomic elements. This allows us to identify the structure of each container, even if it is an inner container. As a result, we are able to detect the hierarchical layout for the entire mockup.

E. Code Generation

The code generation phase is the final step in which the UI representation object, along with the layout hierarchy, is utilized to generate code that can be used across multiple platforms, including Android and iOS. Nowadays, there are several cross-platform solutions like Flutter and React Native, which aim to develop code once and run it seamlessly on both Android and iOS mobile systems. To generate cross-platform code, we made use of Flutter, an open-source UI software development kit developed by Google. We opted for Flutter over other alternatives because it eliminates the use of Platform Primitives. This ensures that the app visually appears almost identical across all platforms, without relying on native look components that may have variations. Algorithm 7 demonstrates the methodology employed for generating code.

Algorithm 6: Checking Alignment between UI Elements

Input:	Element (A) boundary_box [XLeft, YTop, XRight, YBottom] Element (B) boundary_box [XLeft, YTop, XRight, YBottom]
Output:	If the two elements are adjacent, the function returns true; otherwise, it returns false.

```
1     procedure calculate_vertical_overlap (box_A, box_B)
2         y1 = max(box_A[YTop], box_B[YTop])
3         y2 = min(box_A[YBottom], box_B[YBottom])
4         overlap = y2 - y1
5         return overlap
6     end procedure
7     procedure is_side_by_side(box_A, box_B):
8         overlap = calculate_vertical_overlap(box_A, box_B)
```

Algorithm 6: Checking Alignment between UI Elements

```
        height1 = box_A[YBottom] – box_A[YTop]
        height2 = box_B[YBottom] – box_B[YTop]
        return overlap >= 0.9 * min(height1, height2)
end procedure
```

Algorithm 7: Generating cross-platform code

Input:	Output of Alg. 5 (all_elements_adjacents_list). Each inner list (element_list) in all_elements_adjacents_list represents whether the current element has neighboring elements or not
Create:	Statefull widget class for each UI element type that receives visual properties as parameters. Each widget in a separate file.
Output:	Return generated front-end cross-platform code

```
procedure generate_code(all_elements_adjacents_list)
    let column_widget_list as list
    for each element_list in all_elements_adjacents_list do
        if element_list length equals 1 then
            Check element type and call its widget file
            Send element's visual properties as parameters
            Add element's widget to column_widget_list
        end if
        if element_list length >1 then
            let row_widget_list as list
            for each element in element_list do
                Check element type and call its widget file
                Send element's visual properties as parameters
                Add element widget to row_widget_list
            end for
            Add row_widget_list to column_widget_list
        end if
    end for
    return column_widget_list
end procedure
```

There are two primary concerns that require attention: (1) ensuring code readability and (2) implementing effective error handling. Ensuring code readability is a top priority for us. To achieve this, we embrace the concept of reusable components, similar to writing a function once and utilizing it multiple times. Each UI element is represented as a custom widget, which accepts parameters to describe its visual properties as described in the UI representation object. Each widget is organized into its own separate file. Eventually, the generated code files need to be compiled to run on the desired platform. Finally, a mechanism is implemented to handle errors that may appear in Flutter. Unlike HTML markup language, where errors may not disrupt the entire process, Dart (programming language used in Flutter) needs to be successfully compiled in order to run.

VI. Evaluation and Results

A diverse set of evaluation metrics and criteria are utilized in this type of research, indicating a lack of a clear standard for evaluation. To address this issue within this research, we aim to establish a clear standard for evaluation. The evaluation focuses on two main aspects: (1) the accuracy of object detection models and (2) the degree of similarity in user interfaces between the screens of the mockup and the screens generated from code. Furthermore, a comparison with existing systems was conducted.

The first aspect is to evaluate the accuracy of object detection models, commonly used evaluation metrics are utilized including Accuracy, Precision, and Recall, and others which are widely used in the field of object detection.

$$\text{Accuracy} = \frac{T_P + T_N}{\text{Total Predictions}} \quad (1)$$

Object detection models are commonly evaluated based on a metric called Intersection Over Union (IOU). This metric assesses the extent of overlap between two bounding boxes: the predicted bounding box and the ground truth bounding box. During the training stage, a target IOU threshold of 0.5 is typically sought, meaning that if the model predicts an object with a bounding box that overlaps the ground truth box by at least 50%, it is considered a valid prediction. Adjusting the IOU threshold can impact the values in the confusion matrix. This adjustment influences the number of true positives (TP), false positives (FP), and false negatives (FN), thereby impacting the overall performance metrics derived from the confusion matrix including precision, recall, and F1 score.

In order to demonstrate the efficacy of our proposed models in classifying UI components, three metrics measures were utilized: precision, recall, and F1 score. The calculation of these measurements is as follows:

$$\text{Precision} = \frac{T_P}{T_P + F_P} \quad (2)$$

$$\text{Recall} = \frac{T_P}{T_P + F_N} \quad (3)$$

$$\text{F1 score} = \frac{2\,x\,(precision\ x\ recall)}{precision + recall} \quad (4)$$

These metrics were assessed on the validation set for both atomic (A) and container (C) models. The measured metrics for each UI element are presented in Table I, showing the interesting results achieved by both trained models. Despite the presence of various styles within each component, it can accurately identify almost all of them. Fig. 11 shows sample of detection results by YOLOv7 on the validation set of both atomic and container models. In the realm of object detection, Average Precision (AP) and Mean Average Precision (mAP) have gained widespread popularity as the primary evaluation metrics in recent years. AP is a way to summarize the precision-recall curve into a single value representing the precision at different levels of recall. The average precision is computed by taking the mean of these precision values across all recall levels. A high average precision signifies strong performance in terms of both precision and recall, while a low average precision indicates lower values for either or both metrics. Typically, average precision is calculated separately for each class according to this equation:

$$\text{Average Precision (AP)} = \int_{r=0}^{1} p(r)dr \quad (5)$$

To evaluate the performance of object detection model across the different classes, the mAP is calculated by taking the average of the AP values for all the classes being considered as shown in this equation:

$$\text{Mean Average Precision (mAP)} = \frac{1}{k}\sum_{i}^{k} AP_i \quad (6)$$

The emphasis was placed on evaluating the overall performance of both YOLO models by prioritizing the measurement of mAP. Notably, the atomic model achieved an outstanding mAP score of 91.37%, while the container model achieved a respectable score of 87.40%. These results indicate a strong capability for detecting elements in both models, reflecting their strong performance in this regard. We selected mAP as our primary evaluation metric because it signifies a model's stability and consistency across different confidence thresholds. A high mAP suggests that the model performs well at various levels of confidence in its predictions. On the other hand, Precision, Recall, and F1 score are metrics used to assess the model's performance at a specific confidence threshold. When the mAP is good, it indicates that the model consistently achieves high precision, recall, and F1 score across different confidence thresholds. This consistency implies the model's stability and reliability in making accurate predictions.

The second aspect is to measure the similarity of images between the mockups and those produced by the generated code. Three commonly employed image similarity metrics, namely mean squared error (MSE), mean absolute error (MAE), and Structural Similarity Index (SSIM), are utilized to measure the similarity. The Mean Square Error calculates the average of the squared differences between the predicted and actual values. It serves as a metric to measure the disparity between the two images, where higher values signify a larger dissimilarity. On the other hand, the Mean Absolute Error calculates the average absolute difference between the predicted and actual values. Similar to MSE, a lower MAE indicates better model performance, as it means that, on average, the predictions are closer to the actual values.

TABLE I. PERFORMANCE OF THE TRAINED MODELS

UI Element	Precision	Recall	F1 score
Checkbox (A)	0.93	0.95	0.94
Date Picker (A)	0.95	0.97	0.96
Icon (A)	0.91	0.87	0.89
Image (A)	0.81	0.86	0.83
Input (A)	0.83	0.8	0.81
Map View (A)	0.89	0.82	0.85
On-Off Switch (A)	0.93	0.96	0.94
Page Indicator (A)	0.94	0.88	0.9
Radio Button (A)	0.86	0.93	0.89
Slider (A)	0.9	0.83	0.86
Text (A)	0.91	0.97	0.93
Progress Bar (A)	0.84	0.78	0.8
Button (C)	0.79	0.82	0.8
List Item (C)	0.92	0.86	0.89
Card (C)	0.78	0.76	0.77
Drawer (C)	0.89	0.94	0.91
Modal (C)	0.9	0.86	0.88
Multi-Tab (C)	0.88	0.86	0.87
Toolbar (C)	0.91	0.87	0.89
Bottom Sheet (C)	0.83	0.86	0.84
Spinner (C)	0.78	0.8	0.79
Button Bar (C)	0.91	0.93	0.92
Bottom Navigation (C)	0.81	0.82	0.81

Structural Similarity Index is a widely used metric in image processing that quantifies the structural similarity between two images. It takes into account three components: luminance, contrast, and structure. By comparing the SSIM index between the mockup image and the generated code image, we can assess how closely they resemble each other in terms of their structural characteristics. A higher SSIM index indicates a higher similarity between the two images. When evaluating on a testing set of 50 images, our results for MSE, MAE, and SSIM were 30%, 25.7%, and 83.3%, respectively.

Inference time is an important performance metric, as it refers to the measurement of the time it takes for a machine or deep learning model to make predictions or inferences on new data. In order to determine the inference time of YOLO models, the average time it took for inference on a testing set consisting of 50 images was calculated. The inference time varied between 45.6 ms and 85.0 ms, and we observed that as the number of elements in an image mockup increased, the inference time also increased as illustrated in Table II.

Fig. 11. Sample of detection results by YOLOv7 on the validation set of both atomic and container models.

TABLE II. INFERENCE TIME FOR ATOMIC MODEL

Number of UI elements detected	Inference time
4 Elements	46.9 ms
32 Elements	65.0 ms

VII. DISCUSSION

To validate the significance of the proposed approach, a comparative analysis was performed between the proposed system and other systems that specifically target high fidelity mockups using deep learning methods. Table III presents a comprehensive comparison encompassing multiple dimensions, such as system architecture, performance metrics, and the count of detected UI elements. It is important to highlight that each system utilizes a subset of metrics, and therefore, each metric will be compared with its corresponding counterpart in our set of metrics.

Table III presents three distinct categories of techniques: (1) end-to-end, (2) hybrid, and (3) object detection. The end-to-end approach (E) utilizes a comprehensive deep learning model to process mockups or wireframes and generate source code, which can then be transformed into a user interface. Hybrid techniques (H) typically employ traditional computer vision methods to extract the spatial information of UI elements, followed by CNN-based classification to determine their respective types or classes. Object detection (O) involves the identification, labelling, and precise delineation of objects within an image to improve their recognition. Based on this comparison, it is evident that the proposed approach exhibits an improvement in recognizing GUI mockup elements compared to the other systems, although it detects a larger number of elements. It is crucial to emphasize that a comprehensive study was conducted comparing an earlier version that encompassed all UI elements in a single YOLOv7 model. However, the performance of this one YOLOv7 model was not comparable to the two-model approach (atomic and container) due to challenges posed by the visual similarity between certain classes, such as CARD and BUTTON, and DATE_PICKER and MODAL.

TABLE III. COMPARISON BETWEEN THE PROPOSED APPROACH AND OTHER SYSTEMS

Criteria	[10]	[15]	[16]	[17]	[18]	Proposed
Number of UI elements	NA[a]	15	NA	15	12	23
Technique utilized	E	H	H	H	O	O
Text recognition (OCR)	No	Yes	NA	Yes	Yes	Yes
Training dataset	Custom	Custom	Custom	Rico	Custom (iOS)	Rico
Accuracy	77%	NA	85%	NA	NA	88.2%
F1 Score	NA	NA	NA	52%	NA	86.8%
Precision	NA	91.1%	NA	NA	NA	87.3%
mAP	NA	NA	NA	NA	87.5%	91.37%, 87.4% for (A), (C)

a. NA stands for Not Available

VIII. CONCLUSION

Converting mockup design images into front-end code presents a formidable challenge, as it necessitates a visual understanding of the images to detect the UI elements and their hierarchical structure. This paper introduces a novel approach that generates cross-platform front-end code from high fidelity mockup images. At the core of the proposed pipeline, YOLOv7 is utilized for the object detection phase. The approach utilizes YOLOv7 to accurately detect atomic and container UI elements, capturing their spatial location, and subsequently leverages this information to construct a comprehensive UI representation object that encompasses the layout hierarchy of elements within the mockup, showcasing its ability to effectively identify UI elements in mockups. Our second contribution entails the development of a data preprocessing pipeline aimed at addressing the limitations present in the semantic dataset. This pipeline enables us to construct custom datasets specifically tailored to the atomic and container models. The conducted technical evaluation showcases the promising nature of this approach and encompasses a broad spectrum of evaluation metrics, providing a foundation for future studies. This study ensures that deep learning techniques are well-suited for visual recognition tasks involving various types of GUI components.

IX. FUTURE WORK

Despite the comparatively small training datasets used, remarkable results are achieved. As a future work, it is imperative to augment the dataset by incorporating additional instances of elements and meticulously annotating them, thereby providing the models with a more diverse and comprehensive set of training data. Furthermore, there is a need for more extensive coverage of certain cases in the UI element grouping approach. This is particularly important when dealing with scenarios where multiple vertically arranged cards are aligned next to only one card.

REFERENCES

[1] IDC, "Mobile Trends Report," 2015. [Online]. Available: https://www.appcelerator.com/resource-center/research/2015-mobile-trends-report/ Accessed: 15 February 2018.

[2] B. A. Myers and M. B. Rosson, "Survey on user interface programming," in Proceedings of the SIGCHI conference on Human factors in computing systems - CHI '92, New York, New York, USA: ACM Press, 1992, pp. 195–202. doi: 10.1145/142750.142789.

[3] M. W. Newman and J. A. Landay, "Sitemaps, storyboards, and specifications," in Proceedings of the 3rd conference on Designing interactive systems: processes, practices, methods, and techniques, New York, NY, USA: ACM, Aug. 2000, pp. 263–274. doi: 10.1145/347642.347758.

[4] P. Campos and N. Nunes, "Practitioner Tools and Workstyles for User-Interface Design," IEEE Software, vol. 24, no. 1, pp. 73–80, Jan. 2007, doi: 10.1109/MS.2007.24.

[5] T. Silva da Silva, A. Martin, F. Maurer, and M. Silveira, "User-Centered Design and Agile Methods: A Systematic Review," in 2011 AGILE Conference, IEEE, Aug. 2011, pp. 77–86. doi: 10.1109/AGILE.2011.24.

[6] C. Dong, C. C. Loy, K. He, and X. Tang, "Image Super-Resolution Using Deep Convolutional Networks," Dec. 2014, [Online]. Available: http://arxiv.org/abs/1501.00092.

[7] B. Varadarajan, G. Toderici, S. Vijayanarasimhan, and A. Natsev, "Efficient Large Scale Video Classification," May 2015, [Online]. Available: http://arxiv.org/abs/1505.06250.

[8] A F M Saifuddin Saif, Trung Duong and Zachary Holden, "Computer Vision-based Efficient Segmentation Method for Left Ventricular Epicardium and Endocardium using Deep Learning" International Journal of Advanced Computer Science and Applications(IJACSA), 14(12), 2023. http://dx.doi.org/10.14569/IJACSA.2023.0141201.

[9] C.-Y. Wang, A. Bochkovskiy, and H.-Y. M. Liao, "YOLOv7: Trainable bag-of-freebies sets new state-of-the-art for real-time object detectors," Jul. 2022, [Online]. Available: https://arxiv.org/abs/2207.02696.

[10] T. Beltramelli, "pix2code," in Proceedings of the ACM SIGCHI Symposium on Engineering Interactive Computing Systems, New York, NY, USA: ACM, Jun. 2018, pp. 1–6. doi: 10.1145/3220134.3220135.

[11] Z. Zhang, Y. Ding, and C. Huang, "Automatic Front-end Code Generation from image Via Multi-Head Attention," in 2023 4th International Conference on Computer Engineering and Application (ICCEA), IEEE, Apr. 2023, pp. 869–872. doi: 10.1109/ICCEA58433.2023.10135462.

[12] B. Cai, J. Luo, and Z. Feng, "A novel code generator for graphical user interfaces," Sci Rep, vol. 13, no. 1, p. 20329, Nov. 2023, doi: 10.1038/s41598-023-46500-6.

[13] B. Asiroglu et al., "A Deep Learning Based Object Detection System for User Interface Code Generation," in 2022 International Congress on Human-Computer Interaction, Optimization and Robotic Applications (HORA), IEEE, Jun. 2022, pp. 1–5. doi: 10.1109/HORA55278.2022.9799941.

[14] T. A. Nguyen and C. Csallner, "Reverse Engineering Mobile Application User Interfaces with REMAUI (T)," in 2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE), IEEE, Nov. 2015, pp. 248–259. doi: 10.1109/ASE.2015.32.

[15] K. Moran, C. Bernal-Cardenas, M. Curcio, R. Bonett, and D. Poshyvanyk, "Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps," IEEE Transactions on Software Engineering, vol. 46, no. 2, pp. 196–221, Feb. 2020, doi: 10.1109/TSE.2018.2844788.

[16] S. Chen, L. Fan, T. Su, L. Ma, Y. Liu, and L. Xu, "Automated Cross-Platform GUI Code Generation for Mobile Apps," in 2019 IEEE 1st International Workshop on Artificial Intelligence for Mobile (AI4Mobile), IEEE, Feb. 2019, pp. 13–16. doi: 10.1109/AI4Mobile.2019.8672718.

[17] M. Xie, S. Feng, Z. Xing, J. Chen, and C. Chen, "UIED: a hybrid tool for GUI element detection," in Proceedings of the 28th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering, New York, NY, USA: ACM, Nov. 2020, pp. 1655–1659. doi: 10.1145/3368089.3417940.

[18] X. Zhang, L. De Greef, and S. White, "Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels," in Conference on Human Factors in Computing Systems - Proceedings, Association for Computing Machinery, May 2021. doi: 10.1145/3411764.3445186.

[19] B. Deka et al., "Rico," in Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology, New York, NY, USA: ACM, Oct. 2017, pp. 845–854. doi: 10.1145/3126594.3126651.

[20] T. F. Liu, M. Craft, J. Situ, E. Yumer, R. Mech, and R. Kumar, "Learning Design Semantics for Mobile Apps," in Proceedings of the 31st Annual ACM Symposium on User Interface Software and Technology, New York, NY, USA: ACM, Oct. 2018, pp. 569–579. doi: 10.1145/3242587.3242650.

[21] Tzutalin, LabelImg, 2015. [Online]. Available: https://github.com/tzutalin/labelImg. Accessed: Feb. 26, 2021.

[22] A. Kay, "Tesseract: an open-source optical character recognition engine," Linux Journal, vol. 2007, no. 159, p. 2, 2007.