Enhancing widget recognition for automated Android testing using Computer Vision techniques

Yadini Pérez López*
yadini.lopez@sidia.com
Sidia Research and Development Institute
Manaus, AM

Laís Dib Albuquerque
lais.albuquerque@sidia.com
Sidia Research and Development Institute
Manaus, AM

Gilmar J. F. Costa Júnior
gilmar.junior@sidia.com
Sidia Research and Development Institute
Manaus, AM

Daniel Lopes Xavier
daniel.xavier@sidia.com
Sidia Research and Development Institute
Manaus, AM

Leticia Balbi†
leticia.balbi@sidia.com
Sidia Research and Development Institute
Manaus, AM

Juan G. Colonna
juancolonna@icomp.ufam.edu.br
Federal University of Amazonas
Manaus, AM

Richand Degaki
rhd@icomp.ufam.edu.br
Federal University of Amazonas
Manaus, AM

ABSTRACT

Widget recognition is crucial for automated Android black box testing. Over the past ten years, different industrial tools and academic works have been available for identifying Graphical User Interface (GUI) components in Android screens. Traditional identification methods, like GUI hierarchy parsing, often struggle with dynamic content and complex structures. In contrast, Computer Vision (CV) techniques provide greater robustness and flexibility to adapt to different screen resolutions, design specifications, and patterns. However, the CV-based solutions available are still limited concerning the variety of widgets that can be recognized. Moreover, the current identification of GUI components mainly relies on classification, which can lead to ambiguous lists with repeated elements. In this paper, we combine different CV-based techniques to extract context-based descriptions for each widget, to enhance the identification process by going beyond class recognition for describing widgets. We also implemented two primary CV-based approaches for widget recognition: Object Detection combined with Classification, and a One-Stage Recognition method. We trained and evaluated the approaches on a custom **105** classes widget dataset. Moreover, we present a Computer Vision-based method for describing widgets using their contextual meaning on Android screen captures.

KEYWORDS

Automated Android Testing, Test Portability, Object Recognition, Object Classification, Optical Character Recognition, Graphical User Interface component, Widget Recognition

*Correspondence author.
†Current e-mail: leticiabalbisv@gmail.com.

In: Proceedings of the Brazilian Symposium on Multimedia and the Web (WebMedia'2024). Juiz de Fora, Brazil. Porto Alegre: Brazilian Computer Society, 2024.

ISSN 2966-2753

1 INTRODUCTION

Automated testing of Android applications is a critical component in ensuring the quality and reliability of software. One key aspect of this process is widget recognition, which involves identifying and interacting with various elements of a Graphical User Interface (GUI) [36]. Effective widget recognition allows testing frameworks to simulate user actions, verify the presence and state of User Interface (UI) components, and ensure that applications respond correctly to user inputs. This is essential for detecting and diagnosing issues that might not be apparent through manual testing alone, ultimately leading to higher-quality software [34].

Current approaches to widget recognition primarily fall into two categories: GUI hierarchy parsing and Computer Vision (CV) techniques. The first one involves extracting and analyzing the structural information of the application's UI, while CV techniques rely on image processing to recognize widgets directly from screenshots [4, 17, 31, 33, 40]. While GUI hierarchy parsing is intuitive and straightforward, it often struggles with dynamic content and complex structures, leading to incomplete or inaccurate results. On the other hand, CV-based methods offer greater flexibility and robustness, particularly in environments with frequent interface changes or non-standardized structures [37].

In this paper, we explore and compare two primary approaches to widget recognition: Object Detection combined with Classification, and a One-Stage Recognition method. We also address how we generated two datasets tailored for deep learning widget classification and recognition. Our data and recognition approaches focus on identifying a wide variety of widgets, up to 105 different classes, and generating UI context-based descriptions for each widget, attempting to expand the scope of automated black box Android testing [37].

The main contributions of our work are:

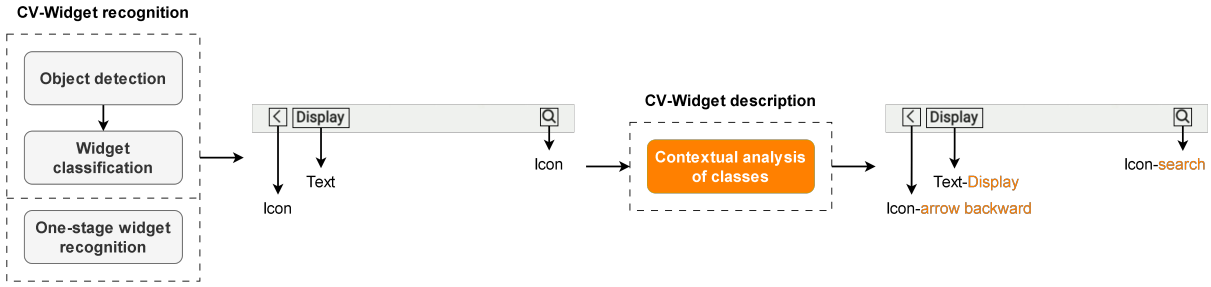


Figure 1: Two main approaches identified in the literature and industrial tools.

(1) We have trained two widget recognition approaches using a dataset of 105 classes of GUI components and compared their performance.
(2) We have trained and evaluated a widget identification method that is not limited to class recognition but generates context-based information for complementing the widget description.

To the best of our knowledge, this is the first widget identification approach (recognition + smart description) that focus on identifying a larger variety of widgets, up to 105 different classes, and generates a context-based description for each widget identified.

The rest of the paper is organized as follows: section 2 reviews the main knowledge and related works with this research. Section 3 explains our methodology, detailing each stage of the widget identification process, including recognition, smart description, and the datasets used. Section 4 presents the evaluation of our methods, comparing the two recognition techniques and assessing the performance of our smart descriptor method. We also address five research questions to demonstrate our contributions. Finally, sections 5 and 6 provide our final considerations and conclusions, respectively.

2 BACKGROUND AND RELATED WORKS

Widget recognition is crucial for Android testing automation tasks such as GUI and functional verification. It allows the testing framework to interact with and validate the behavior of the application's UI elements [53]. By recognizing widgets, automated tests can simulate user actions, verify the presence and state of UI components, and ensure that the app responds correctly to user inputs. This level of interaction is essential for issues detecting and diagnosing that might not be apparent through manual testing alone, leading to higher quality and more reliable software. Moreover, widget recognition facilitates the creation of more robust and maintainable test scripts, as the tests can adapt to cross-device-release changes in the UI layout or design without requiring extensive manual updates [16].

Furthermore, maximizing the number of widget types that a GUI recognizer can identify is paramount for automated Android testing, as it directly influences the comprehensiveness and accuracy of the testing process. Diverse widget recognition ensures the testing framework can interact with all elements of an application's interface, from basic buttons and text fields to complex custom widgets and dynamic content. Limitations in recognizing certain widget types can lead to incomplete testing and overlooked bugs [3]. By expanding the range of recognized widget types, developers can ensure that their automated tests cover a wider spectrum of UI elements, thereby improving the overall quality and stability of the application [49].

Nowadays, several industrial and academic solutions are available for automated GUI testing. These solutions were implemented following two main approaches: GUI hierarchy parsing and Computer Vision (CV) techniques for widget recognition. Moreover, some of these approaches extract the contextual meaning of the GUI elements, resulting in more detailed and complete identifiers.

2.1 GUI hierarchy parsing for widget identification

Component recognition through metadata, such as dump hierarchies, leverages the structural information of the application's UI to identify and interact with various widgets. This method relies on parsing the UI's XML or other hierarchical representations to extract widget properties and relationships, allowing precise targeting and manipulation during testing. This can be done using software [18, 21, 30, 32] or employed by frameworks and tools responsible for assisting in Android testing [2, 4–7, 13, 19, 20, 25, 26, 28, 33, 38, 40, 46, 48, 51, 54, 55, 58].

Although GUI hierarchy parsing for widget recognition is an intuitive and easy-to-implement solution, it has several notable limitations. First, it often struggles with dynamic content, and complex and deeply nested structures, leading to incomplete or inaccurate parsing results [36]. Moreover, this approach can be significantly hindered by variations in widget naming conventions and inconsistencies in hierarchical organization across different applications [34]. Additionally, it tends to perform poorly when encountering custom widgets or those that deviate from standard GUI frameworks, as these elements may not be accurately represented in the hierarchical data [36]. Furthermore, the reliance on a static representation of the GUI state can limit the technique's effectiveness in real-time or interactive environments where the GUI can change frequently. This can be translated into maintenance costs and poor portability of automation scripts. Finally, GUI hierarchy parsing can be computationally expensive, making it less suitable for applications that require real-time performance or those operating on resource-constrained devices [36].

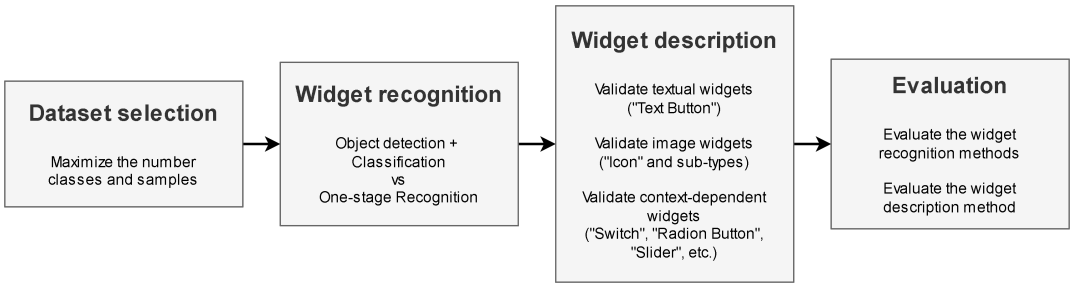


Figure 2: Research steps for *widget recognition* and *description* using Computer Vision techniques.

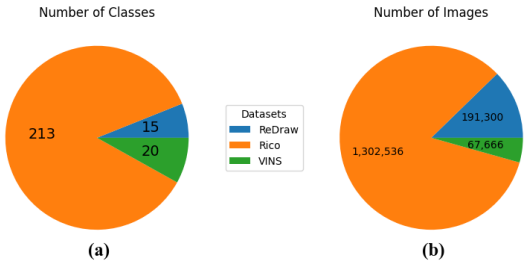


Figure 3: Number of classes (a) and samples (b) of Rico, Redraw, and VINS original datasets.

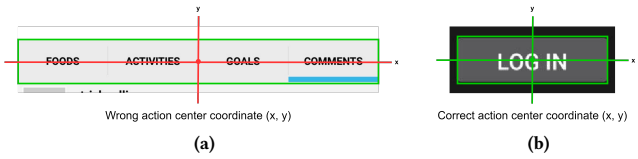


Figure 4: Example of "Multi-Tab" non-atomic widget (a) and "Text Button" atomic widget (b). Non-atomic widgets contain different atomic components, and their center coordinates (x,y) may have no effect during automated interactions (a).

2.2 Computer Vision-based widget recognition

CV-based solutions offer distinct advantages over GUI hierarchy parsing for widget recognition in software applications. Primarily, Computer Vision techniques provide a more robust and flexible approach, as they can recognize and adapt to a wide variety of visual styles, themes, and customizations that GUIs often employ [37]. This adaptability is crucial in environments where interfaces frequently change or lack standardized structures [34]. Furthermore, CV methods do not rely on the underlying code structure or access to the application's source code, making them applicable to a broader range of scenarios, including black-box applications [14]. They also enhance cross-platform compatibility since the visual elements remain consistent regardless of the operating system or the framework used to build the GUI [36].

CV-based widget identification have been addressed through two main stages in the academic and industrial solutions [43]. Figure 1 illustrates them: *widget recognition* and *description*. The first one aims to localize and classify widgets on screen capture images. For this recognition task, we identified two recurrent approaches: Object Detection + Widget Classification [27, 31, 43, 52, 59, 61] and One-Stage Recognition [1, 12, 14, 22, 44, 57, 60]. Sparsely, some works also included the second stage, the *widget description* stage, whose goal is to generate more detailed information about the GUI element [10, 11, 29, 35, 39, 41, 43, 56, 62]. In other words, this stage seeks to differentiate GUI elements of the same class according to their contextual meaning on the screen. Figure 1 shows a screen capture fragment of the "Settings" application that contains three GUI elements: an "Icon arrow backward", a text label "Display", and an "Icon search". The *widget recognition* stage localizes and classifies the elements according to a set of class names, in this example, "Icon", "Text", and "Icon". Then, with the *widget description* stage, the class information is complemented with the icons sub-category, "Icon-arrow backward" and "Icon search", and the actual content of the text element "Display".

In automated Android testing scenarios, like functionality tests, performing a stage one + stage two pipeline will generate better widget representations for maximizing the reproducibility of test steps. Moreover, GUI tests will improve the precision and also readability of bug reports, since problematic widgets will be fully described and easy to localize, which is desirable by Quality Assurance (QA) and Development teams [36].

It is important to highlight that according to our study, only a few solutions attempt to go beyond class-based widget identification [11, 39, 41]. Also, most of these works use the UI metadata information for locating and classifying the GUI elements, which is not scalable and leads to incompletely described widgets. Moreover, these works are limited concerning the variety of widgets that can be recognized, from 10 to 15 different classes [35, 43, 62].

To address the above limitations, we developed a methodology with two main goal: to expand the recognition capacity in terms of widget classes and to enhance the contextual description of the GUI components identified. Our solution is based on state-of-the-art Object Recognition techniques for widget localization and classification. Novelty, we present a combination of CV techniques, called Smart Descriptor, for contextual-based widget description.

3 METHODOLOGY

Figure 2 illustrates our research methodology. Firstly, we selected a dataset to work with Computer Vision techniques for widget recognition and description. Then, we implemented the most recurrent approaches in the literature and industrial solutions for

widget recognition: Object Detection + Classification, and One-Stage Widget Recognition. After that, we studied how to extract contextual information about text, images, and "context-dependent" widgets to generate descriptions for those GUI components. Finally, we evaluated the *widget recognition* and *description* methods using appropriate metrics for each technique.

3.1 Mobile GUI dataset

Rico [15, 24], ReDraw [42, 43] and VINS [8, 9] datasets are the most recurrently used collections for widget recognition, publicly available. Figure 3 shows the number of classes (a) and samples (b) for each one of the three datasets. Rico is the largest, with 782, 589 Region of Interest (RoI) of widgets in different screenshots mined over 9.3k free Android apps from 27 categories [24]. His extension makes Rico ideal for working with deep-learning algorithms. However, Rico is noisy, presenting wrong and missing GUI component bounds in the semantic annotations [14, 37]. Still, to our knowledge, Rico is the Android screen capture dataset with a greater variety of widgets, specifically 115 classes, and 98 icon types [24]. Moreover, Rico includes the GUI hierarchy for each screen capture, which contains valuable widget information such as the actual "text" for components like "labels" and "text buttons", which serve as ground truth data for evaluating GUI recognition methods [37].

This work seeks to recognize and describe widgets while maximizing their variety. Regarding this, we decided to work with the Rico dataset because it is publicly available and has the largest number of annotated classes and Android screenshots [37]. Nevertheless, we did not work with the total of 213 types of widgets annotated on the original Rico dataset.

We analyzed the Rico classes and grouped them into categories:

- 8 **Atomic classes**[1];
- 13 **Container classes**[2];
- 98 **"Icon" types**[3];
- 27 **"List Item" sub-types**[4];
- 3 **"Pager Indicator" sub-types**[5];
- 7 **"Text" sub-types**[6];
- 54 **"Text Button" sub-types**[7];
- 3 **Confusing classes**[8] which samples have an identical appearance with atomic samples.

In this work, we created our dataset by extracting all samples from the 8 **Atomic classes** and the 97 **Icon types**.

We decided to exclude **Container** classes since they accommodate other clickable elements inside, which may demand more complex interaction flows in test automation tools and lead to incorrect interactions. For example, as shown in Figure 4(a), interacting with a "Multi-Tab" widget is impossible through a single tap on the center coordinate. Instead, it requires a second routine for recognizing the contained widgets, and then, selecting one of them to interact with. In this figure, the red lines indicate that the center of the "Multi-Tab" element does not correspond to a valid clickable coordinate. On the other hand, as shown in Figure 4(b), "Text Buttons" can be triggered with a single tap. Green lines show that the center of this widget leads to activating the "Log in" action.

Moreover, we also excluded classes from the following **sub-type** groups: **"List Item", "Pager Indicator", "Text" and "Text Button"**. Samples of these classes are originally annotated in Rico as belonging to more than one class. However, visually they have the same appearance, i.e., "Text Button-delete" and "Icon-delete", have the same "image pattern" but are annotated as belonging to more than one class. This is considered noisy information that may hinder the convergence of machine learning algorithms. According this, the **Confusing** classes was not considered as well, and analyzing images of **Icon types**, we found that the "Icon-switcher" class is composed of 1046 identical instances, which also look very similar to other types of icons. To avoid this noise information, we also excluded this class.

After discarding the mentioned classes of widgets, we kept 105 categories: 8 **Atomic classes** and 97 **Icon types**.

3.1.1 *Rico Annotations.*

We created two different datasets for implementing the widget recognition approaches: (1) Object Detection + Classification and (2) One-Stage Recognition. These datasets included the 105 classes of widgets previously selected.

The first dataset was generated by cropping the RoIs from the Rico screenshots. This was done by traversing the GUI graph available on the semantic annotations files [24] and saving the labeled crops corresponding to the bounds of each "componentLabel" and "iconClass" key. This dataset was used for training the classification algorithm of the Object Detection + Classification approach. In total, this dataset resulted in 355, 999 images that we split for training, testing, and validation in the proportion of 50:30:20 respectively.

The other dataset was created to train the One-stage recognition approach. Here, we used the Rico semantic annotations files to generate the class and bounding box ground truth annotations, resulting in a dataset of 44, 873 screenshots. We also used the same

[1]Atomic classes: "Checkbox", "Input", "Number Stepper", "On/Off Switch", "Pager Indicator", "Radio Button", "Slider", "Text Button"

[2]Container classes: "Advertisement", "Background Image", "Bottom Navigation", "Button Bar", "Card", "Date Picker", "Drawer", "Map View", "Modal", "Multi-Tab", "Toolbar", "Video", "Web View"

[3]"Icon" types: "add", "arrow_backward", "arrow_downward", "arrow_forward", "arrow_upward", "attach_file", "av_forward", "av_rewind", "avatar", "bluetooth", "book", "bookmark", "build", "call", "cart", "chat", "check", "close", "compare", "copy", "dashboard", "date_range", "delete", "description", "dialpad", "edit", "email", "emoji", "expand_less", "expand_more", "explore", "facebook", "favorite", "file_download", "filter", "filter_list", "flash", "flight", "folder", "follow", "font", "fullscreen", "gift", "globe", "group", "help", "history", "home", "info", "label", "launch", "layers", "list", "location", "location_crosshair", "lock", "menu", "microphone", "minus", "more", "music", "national_flag", "navigation", "network_wifi", "notifications", "pause", "photo", "play", "playlist", "power", "redo", "refresh", "repeat", "reply", "save", "search", "send", "settings", "share", "shop", "skip_next", "skip_previous", "sliders", "star", "swap", "switcher", "thumbs_down", "thumbs_up", "time", "twitter", "undo", "videocam", "visibility", "volume", "volume", "warning", "weather", "zoom_out"

[4]"List-Item" sub-types: "add", "avatar", "book", "bookmark", "call", "chat", "check", "close", "date_range", "delete", "email", "emoji", "facebook", "favorite", "flash", "folder", "font", "fullscreen", "location", "more", "national_flag", "notifications", "playlist", "settings", "sliders", "star", "warning"

[5]"Pager Indicator" sub-types: "menu", "more", "sliders"

[6]"Text" sub-types: "check", "close", "date_range", "favorite", "font", "photo", "wallpaper"

[7]"Text Button" sub-types: "add", "arrow_backward", "arrow_forward", "av_forward", "av_rewind", "avatar", "bookmark", "cart", "chat", "check", "close", "copy", "date_range", "delete", "edit", "emoji", "expand_more", "facebook", "favorite", "file_download", "filter, "flash", "folder", "follow", "font", "gift", "globe", "group", "help", "home", "info", "launch", "layers", "list", "menu", "minus", "more", "notifications", "pause", "photo", "play", "refresh", "search", "send", "settings", "share", "sliders", "star", "swap", "time", "twitter", "videocam", "volume", "warning"

[8]Confusing classes: "Image", "List Item", "Text"

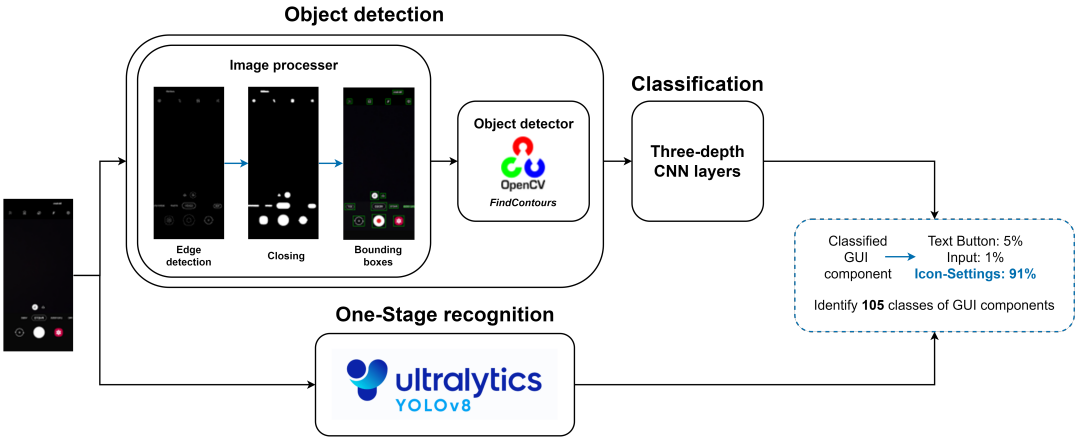


Figure 5: The two approaches implemented for Widget Recognition. The first one uses Digital Image Processing (DIP) techniques for Object Detection (edge filters, binarization, and contour detection) and a Convolutional Neural Network (CNN) model for classification. The second approach uses YOLOv8 pre-trained model to detect and classify the widgets in a single stage.

Table 1: Algorithms and parameterization used in Object Detection stage.

Algorithm	Parameters
Canny Filter	1st Hysteresis threshold (threshold1): 100 2st Hysteresis threshold (threshold2): 200
Morphological Closing	Morphological Operation (op): `cv2.MORPH_CLOSE` Kernel: `np.ones((35, 35), np.uint8`

Contour Detection	Contour Retrieval mode (mode):
	`cv2.RETR_EXTERNAL`
	Contour approximation method (method):
	`cv2.CHAIN_APPROX_SIMPLE`

proportion of 50:30:20 for splitting the data into training, testing, and validation subsets.

3.2 Computer Vision-based widget recognition

The goal of the recognition stage is to localize widgets in Android screenshots and classify them. For this, we implemented two approaches, where the first one uses more traditional techniques, specifically, Digital Image Processing (DIP) for object detection and Convolutional Neural Networks (CNN) for classification. The second approach uses the state-of-the-art "You Only Look Once" (YOLO) object detection algorithm[9].

3.2.1 *Approach 1: Object Detection + Classification.*

According to shown in Figure 5, the "traditional" approach for widget recognition has two stages, object detection and classification. The object detection stage was implemented by, firstly, locating edges using the Canny filter[10] [43], followed by a morphological closing[11] to prevent object splitting [23] and finally, the find contours technique[12] [43] to get the potential RoI on the screenshots. Table 1 shows the parameterization used in each function of the object detection stage.

The Object Detector's output is a set of bounding boxes of all RoIs with a high probability of being widgets. The next step is

Table 2: Details of the 3-depth CNN classifier architecture and the training hyper-parameters.

	Configurations

Layers	`Conv2D(64, (3, 3))`
	`Conv2D(32, (3, 3))`
	`Conv2D(16, (3, 3))`
	`Dense(128))`

Parameters	For all layers, was used ReLU as the activation function, HeUniform as kernel and bias initializers, and L1L2 as kernel and bias regularizers. Between then, a group of layers was used, consisting of:
	`BatchNormalization()`
	`AveragePooling2D((2, 2))`
	`Dropout(0.5)`
	
	At the end of the network, was used:
	`Dropout(0.3)`
	`Dense(105, activation="softmax")`

Hyper-parameters	Optimizer: `Adam(learning_rate=1e-4)` Loss: `"categorical_crossentropy"`

[9]Real-time object detection and image segmentation model, built on cutting-edge advancements in deep learning and CV, offering unparalleled performance according to speed and accuracy [50].
[10]Was used the OpenCV implementation of Canny filter available on: https://docs.opencv.org/4.x/da/d22/tutorial_py_canny.html
[11]Was used the OpenCV implementation of morphological closing available on https://docs.opencv.org/4.x/d9/d61/tutorial_py_morphological_ops.html
[12]Was used the find contour OpenCV implementation available on: https://docs.opencv.org/4.x/d4/d73/tutorial_py_contours_begin.html

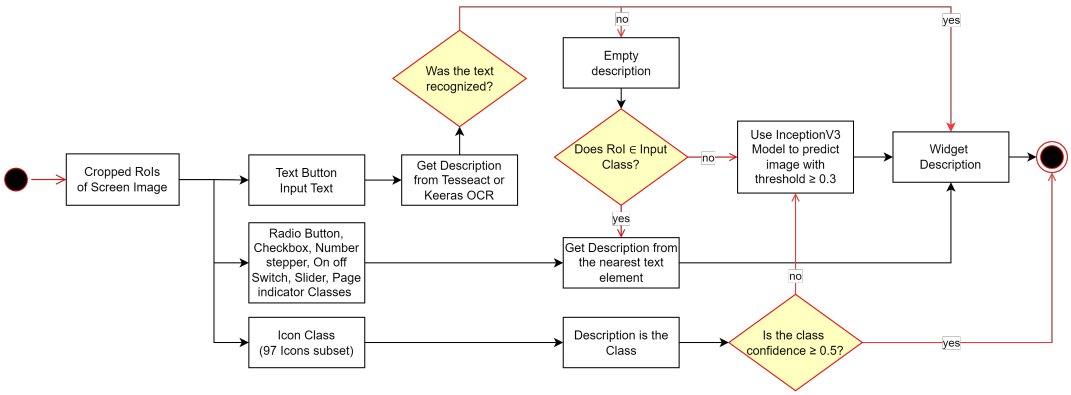


Figure 6: Flowchart of the *Widget Description* stage.

classifying each RoI according to the 105 GUI component classes. For this, we trained a CNN model with three convolutional layers of depth (CNN3). This classifier architecture was proposed by [8]. In the training process, the training dataset was fed to the model in Grayscale format, since early experimentation showed better results when compared with the Red, Green, and Blue (RGB) original format. The input dimensions were 50×50. The model was trained using Tensorflow[13] 2.4 on a server with Ubuntu 18.04 operating system and GPU NVIDIA GeForce RTX 360 TI. Table 2 gives the configuration details about the model and the training hyper-parameters.

3.2.2 *Approach 2: One-Stage Recognition.*

We implemented the one-stage widget recognition approach using YOLOv8 as an alternative solution for Approach 1 described above. YOLOv8 improves the detection accuracy and speed compared with the previous version, making it highly suitable for real-time applications. Although real-time recognition is not the focus of this research, we wanted to explore how this model will perform in our scope, due to the potential application in automated tests where the instant action-motion replay is valuable. To train YOLOv8, we used the Ultralytics[14] framework with the default training parameters.

3.3 Widget Description

In this work, we propose to go beyond widget recognition and describe the RoIs according to the contextual information on the screenshot. For this, we use three main techniques, Optical Character Recognition (OCR), Image Classification, and distance-based heuristic rules. The Figure 6 shows in a flowchart our proposal for describing widget crops according to the class recognized. If the RoI is a "Text Button" or an "Input Text" the content is extracted using Keras-OCR[15] or a Python wrapper of Tesseract[16]. We experimented with both OCRs to verify which performs best in our scope. Different from the "Text Button" class, "Input Text" components are empty sometimes. Hence, if the OCR does not recognize any text, the "Input text" crop gets the description of the nearest textual element on the screenshot. This logic is applied also, to "Checkbox", "Number Stepper", "On/Off Switch", "Pager Indicator", "Radio Button" and "Slider" classes. We calculate the Euclidean distance of the RoI and all textual elements (which are described using the OCR technique) and use the description of the nearest one. Finally, if the approached techniques fail to generate a description, we use the InceptionV3[17] model to extract any semantic information from the RoI.

In the case of "Icon" types classes, we considered the class information explicit enough to describe a RoI. In this work, we focused on recognizing a nice variety of icons (97 classes). However, there are still different icon types unknown to our models. Regarding this, if a RoI is classified as any "Icon" type with a confidence below 0.5, then we use the InceptionV3 model to extract semantic information of the area and validate icon classes that do not exist in the 97 types in our dataset.

4 STUDY

In this work, we have two main objectives. The first is to study how the two state-of-the-art approaches for widget recognition perform in an extended-class dataset (105 classes). The second goal is to propose a method to describe widgets beyond the type or class by including contextual information about what that component represents on screen. We implemented the methods described in the previous sections to attain these goals. Then, to evaluate our proposal, we addressed the following research questions.

4.1 Research Questions

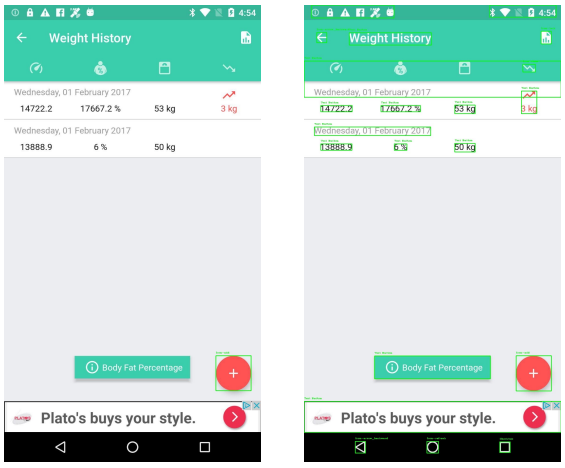
RQ1. Effectiveness of widget recognition: How accurate are the implemented approaches for widget recognition to localize and classify 105 classes of widgets?

RQ2. Analysis of miss-classifications: Which classes are wrongly recognized by the implemented methods?

RQ3. Effectiveness of widget description: How accurate is our proposal for describing widgets?

RQ4. Analysis of uncovered descriptions: What are the major reasons for our widget description method to fail?

[13] Available at https://www.tensorflow.org/
[14] Available at https://github.com/ultralytics/ultralytics
[15] Available at https://keras.io/
[16] Avaiable at https://github.com/tesseract-ocr/tesseract
[17] Available at https://keras.io/api/applications/inceptionv3/



(a) (b)

Figure 7: Rico screenshot with the widgets annotations enclosed in green boxes (a) and the Object Detector + Classifier predictions also enclosed in green boxes (b).

RQ5. Impact of approached CV-based methods for recognizing and describing widgets: When integrated into a mobile test automation tool, the approached CV-based methods for recognizing and describing widgets contribute to generating more comprehensive steps for Test/Script Case generation or Bug Review?

4.2 Measurement

To evaluate the performance of the "traditional" and one-stage widget recognition approaches, we studied the Precision, Recall, and mean Average Precision (mAP). The mAP is a key metric for evaluating object detection models [47]. It combines both Precision and Recall to offer a comprehensive assessment of a model's accuracy. The Precision (P) is the ratio of True Positive detection to the sum of True Positive and False Positive detection. The Recall (R) is the ratio of True Positive detection to the sum of True Positive and False Negative detection [47]. These can be mathematically expressed as:

$$\text{Precision} = \frac{1}{N} \cdot \sum_{N} \left(\frac{\text{True Positives}}{\text{True Positives} + \text{True Negatives}} \right) \quad (1)$$

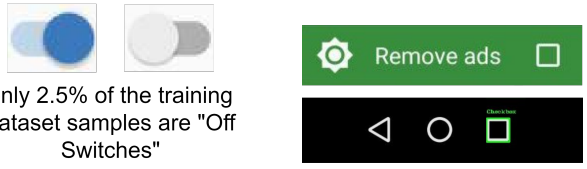
$$\text{Recall} = \frac{1}{N} \cdot \sum_{N} \left(\frac{\text{True Positives}}{\text{True Positives} + \text{False Negatives}} \right) \quad (2)$$

$$\text{mAP} = \frac{1}{N} \cdot \sum_{N} (\text{R}_N - \text{R}_{N-1}) \cdot \text{P}_N \quad (3)$$

where N is the number of classes. mAP metric provides a single value that summarizes the model's overall detection accuracy, making it essential for comparing the performance of different object detection models [47].

Moreover, we assess the *widget description* method using the Character Error Rate (CER). This metric is crucial to evaluate text recognition algorithms[45]. The CER measures the number of character-level errors made by the OCR process and is calculated using the formula:

$$\text{CER} = \frac{S + D + I}{N} \quad (4)$$



(a) (b)

Figure 8: Examples of classes wrongly predicted by the *widget recognition* approaches, (a) shows "on" and "off" switch samples, where "off" instances are under-represented on the training data, and (b) shows a "versatile" appearance widget that fits in two classes, an unselected checkbox, and an icon-app-switch.

where S is the number of substitutions, D is the number of deletions, I is the number of insertions, and N is the total number of characters in the reference text. A lower CER indicates a more accurate OCR system. This metric provides a granular view of OCR performance, enabling developers to identify specific types of errors and improve system robustness by refining algorithms to reduce substitutions, deletions, and insertions. Thus, CER is vital for applications requiring high precision in text recognition, such as digital archiving and automated data entry [45].

5 DISCUSSION

5.1 Widget Recognition

RQ1. Effectiveness of widget recognition: We studied the performance of the two approaches for *widget recognition* on the test set compound by 17, 808 images and their respective annotations. We measured the methods using the mean Average Precision (mAP) metric and attained a low value, where the first approach achieved 0.065, while for YOLOv8, the result was 0.690. To understand these metrics, we did an exhaustive manual inspection of the Rico annotations and the predictions of the widget recognition methods implemented in this work. Then, we find out that Rico has a very high number of missed widget annotations, representing most of the False Positives that impacted the precision of the recognition methods. Figure 7 illustrates this problem: (a) is an annotated screenshot of the Rico dataset, and (b) is the result of executing the Object Detector + Classifier approach for widget recognition on the same image. In (a), more than 80% of the RoI are not annotated, but our recognition approach can detect those areas, unfairly counting as False Positives. Furthermore, the object detector used in the first approach is sensitive to capturing extreme details of the interface, helping to explain the extremely low value performed for the first approach.

RQ2. Analysis of miss-classifications: Regardless of the mentioned Rico annotation problems, we detected some false predictions for both of the widget recognition approaches. During the manual inspection of the methods' predictions, we observed that the "off" samples belonging to the class "On/Off Switch" were always misclassified. Then, we revised the training samples and discovered that only 2.5% of the "On/Off Switch" RoIs were "off" instances, see Figure 8(a). This justifies the miss-predictions of the YOLOv8 recognition model and the CNN3 classifier. To overcome this problem, the "off" switch samples must be augmented to balance with the "on" ones.

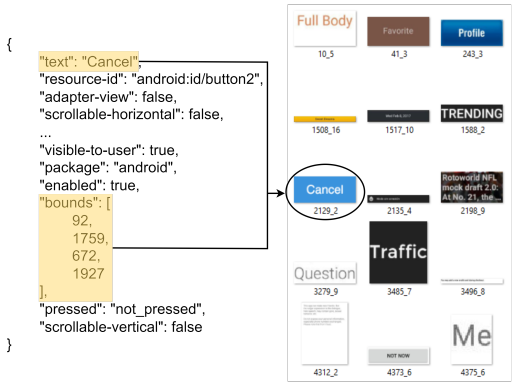


Figure 9: Selection of "Text Button" samples of the dataset created for evaluating the smart descriptor and a fragment of the respective UI hierarchy metadata highlighting the *bounds* and *text attributes*.

Table 3: Character Error Rate (CER) and Response time per request in seconds for Tesseract, Keras-OCR, and the combination of both, in 1,021 "Text Button" crops extracted from manually selected screenshots from Rico testing set.

Method	Mean CER	Response Time (sec)
Tesseract	0.204	2.1
Keras-OCR	0.172	2.28
Tesseract + Keras-OCR	0.171	1.98

Moreover, we found wrong predictions caused by widgets having a "versatile" appearance, meaning that the same image belongs to more than one class. Figure 8(b) shows an example of this, where an unselected "Checkbox" is identical to the "app-switch" icon of the navigation bar.

5.2 Widget Description

To evaluate our Widget Descriptor proposal, we needed the contextual meaning annotations of Rico images. However, the Rico view hierarchies files [24] only have this information for textual elements like "Text Buttons", which is contained in the "text" attribute. This information sparsely appears for other widget types in the "content-desc" attribute. Regarding this, we managed to evaluate quantitatively, only the description generated for "Text Buttons". Due to the mentioned ground truth limitations for the other classes, we did a manual inspection to identify problems in the descriptions generated and have an overall notion of the method's performance.

To evaluate the Widget Descriptor's capacity for recognizing textual components we created a dataset of 1,021 "Text Button" RoIs randomly extracted from the 17,808 samples of the test Rico dataset. For this, we used the UI hierarchy *json* files from Rico [24] to extract the *bounds*, *class* and *text* attributes for each node. Then, we filtered the nodes belonging to the "Text Button" class, used the *bounds* for cropping the RoIs on the full screenshot, and saved the *text* information to generate the RoIs description ground truth.

Figure 9 shows a representative selection of this dataset and a fragment of a UI hierarchy metadata highlighting the *bounds* and *text* attributes.

(a) (b)

Figure 10: Example of "number-only" (a) and "noisy-background" (b) "Text Button".

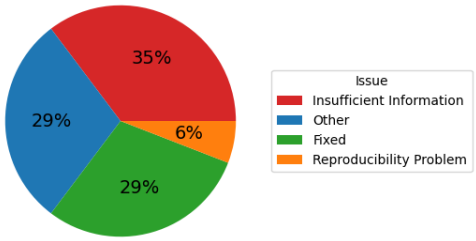


Figure 11: Percentage of Issues reported by the automated exploratory test tool using our widget identification (recognition+description) method for mapping the device under test (DUT) UI at each timestamp.

RQ3 Effectiveness of widget description: The results of evaluating the textual elements are shown in Table 3. Keras-OCR reached a mean CER of 0.172 and a mean execution time per request of 2.28 seconds. Tesseract showed a higher mean CER, 0.204 but better performance with a mean response time per request of 2.1 seconds. Based on these results, we combined the two OCRs sequentially to describe text-based components. Firstly Tesseract is used. Then, if it fails to recognize the text, Keras-OCR should be called. This OCR combination demanded in mean 1.98 seconds per request with a mean CER of 0.171.

RQ4. Analysis of uncovered descriptions: While evaluating the widget descriptor for textual elements, we collected the samples with high CER for manual analysis. Here, we detected that while Tesseract could not recognize "Text Button" samples containing just numbers (Figure 10a), Keras-OCR made accurate predictions for these cases. We also noticed that images with non-solid backgrounds (Figure 10b) had poor results for both OCRs, Tesseract, and Keras.

To have an overall idea of how effective is the *widget description* method for predicting non-textual classes, we ran it on the 17,808 test portion of the Rico images, saved the screenshots with the bounding boxes, and the description generated for each RoI. Then, we manually inspected these results, finding some situations where our method failed to retrieve a valid description for some GUI components. Mainly, these fails occur in "context-dependent" classes when the widget is not side-to-side with the textual widget, but above or below. For example, in some applications is a common design practice to have "Radio Buttons" below a text label. In those cases, our method distance-based method does not perform accurately.

5.3 Applying our widget identification method on an exploratory test tool

RQ5. Impact of approached CV-based methods for recognizing and describing widgets: As a study case, we implemented an API REST service to make available our widget identification approach (recognition + description) for being consumed by an automated Android testing tool focused on exploratory testing. Figure 11 shows the Issues reported by the exploratory tool using our method for six months of 2021–2022. In total 41% of the Issues were stated as being non-reproducible or having insufficient information for reproduction (red and orange portions), 29% were fixed (green) and 29% were reported to third-party owners (blue). From this, we can conclude that the development team effectively processed more than half of the issues reported using our widget identification method to describe steps.

5.4 Limitations and future works

This study investigates two principal approaches for enhancing widget recognition in automated black box Android testing: Object Detection combined with Classification, and a One-Stage Recognition method. Our analysis was supported by two newly introduced datasets, tailored to the demands of deep learning applications in widget recognition and classification. By targeting a comprehensive set of 105 different widget classes and incorporating context-based descriptions, we aimed to improve the scalability of automated testing frameworks.

Despite these advancements, several challenges remain. The diversity and complexity of Android UIs continue to pose significant hurdles. Variations in widget design, dynamic content updates, and the presence of custom widgets that do not adhere to standard conventions all impact the effectiveness of recognition methods. Future work should focus on further refining recognition algorithms to handle these challenges, potentially incorporating additional data sources or leveraging more advanced machine learning techniques.

Even though the One-Stage Recognition Approach has proven to be better, the two approaches implemented for widget recognition presented poor mAP, due to the noisy nature of the data, according to our manual inspection of the two datasets created from Rico. We propose future work to improve the quality of the datasets using techniques and data proposed in a previous work [37], in addition to extending the number of samples for our 105 with smaller but consistent datasets like VINS [9].

To improve our widget description method, we propose to review the distance-based algorithm for "content-dependent" widgets, specifically for those cases where the element is positioned below or above the label. To analyze the feasibility of using the proposed pipeline in real time testing applications, we propose as future work to test the best widget recognition approach of this research combined with widget description component in an environment prepared for this purpose, in order to collect metrics so that improvements and studies can be made.

6 CONCLUSIONS

This paper presents a study of using two advanced approaches to widget recognition for automated Android testing: Object Detection combined with Classification, and a One-Stage Recognition method. We created two datasets of widgets from Rico, one for classification and another one for recognition. We filtered atomic widget types resulting in 105 classes including 97 icon types. We also presented a method for describing widgets beyond just using the class, instead, we extract contextual information about what those components represent on screenshots. Our evaluation of YOLOv8 and the OpenCV Object Detector + CNN Classifier approaches exposed several annotation problems of the Rico dataset, concluding that is not recommended to work with this Rico without performing noise removal routines. Furthermore, the proposed combination of Keras and Tesseract OCRs for extracting information of "Text Button" RoIs presented a low mean CER of 0.172.

ACKNOWLEDGMENTS

This article is a result of the Research and Development project STAR, carried out by the Sidia Institute of Science and Technology, in partnership with Samsung Electronics of Amazônia Ltda. Advertising by the provisions of article 39 of Decree No. 10, 521/2020. This study was financed in part by the Coordenação de Aperfeiçoamento de Pessoal de Nível Superior — Brasil (CAPES) — Finance Code 001. This work was partially supported by Amazonas State Research Support Foundation — FAPEAM — through the POSGRAD project.

REFERENCES

[1] A. A. Abdelhamid, S. Alotaibi, and A. Mousa. 2020. Deep learning-based prototyping of android GUI from hand-drawn mockups. *IET Software* 14 (2020), 816–824. Issue 7. https://doi.org/10.1049/iet-sen.2019.0378

[2] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. 2018. Reinforcement learning for Android GUI testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Lake Buena Vista, FL, USA) *(A-TEST 2018)*. Association for Computing Machinery, New York, NY, USA, 2–8. https://doi.org/10.1145/3278186.3278187

[3] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering* (Essen, Germany) *(ASE '12)*. Association for Computing Machinery, New York, NY, USA, 258–261. https://doi.org/10.1145/2351676.2351717

[4] Domenico Amalfitano, Vincenzo Riccio, Nicola Amatucci, Vincenzo De Simone, and Anna Rita Fasolino. 2019. Combining Automated GUI Exploration of Android apps with Capture and Replay through Machine Learning. *Information and Software Technology* 105 (2019), 95–116. https://doi.org/10.1016/j.infsof.2018.08.007

[5] Luca Ardito, Riccardo Coppola, Simone Leonardi, Maurizio Morisio, and Ugo Buy. 2020. Automated Test Selection for Android Apps Based on APK and Activity Classification. *IEEE Access* 8 (2020), 187648–187670. https://doi.org/10.1109/ACCESS.2020.3029735

[6] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages & Applications* (Indianapolis, Indiana, USA) *(OOPSLA '13)*. Association for Computing Machinery, New York, NY, USA, 641–660. https://doi.org/10.1145/2509136.2509549

[7] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based Android GUI testing using multi-level GUI comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering* (Singapore, Singapore) *(ASE '16)*. Association for Computing Machinery, New York, NY, USA, 238–249. https://doi.org/10.1145/2970276.2970313

[8] Sara Banian, Kai Li, Chaima Jemmali, Casper Harteveld, Yun Fu, and Magy El-Nasr. 2021. *VINS: Visual Search for Mobile User Interface Design.* https://github.com/sbunian/VINS

[9] Sara Banian, Kai Li, Chaima Jemmali, Casper Harteveld, Yun Fu, and Magy El-Nasr. 2021. VINS: Visual Search for Mobile User Interface Design. 1–14. https://doi.org/10.1145/3411764.3445762

[10] Manuele Barraco, Matteo Stefanini, Marcella Cornia, Silvia Cascianelli, Lorenzo Baraldi, and Rita Cucchiara. 2022. CaMEL: Mean Teacher Learning for Image Captioning. arXiv:2202.10492

[11] Jieshan Chen, Chunyang Chen, Zhenchang Xing, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Unblind your apps: predicting natural-language labels for mobile GUI components by deep learning. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. ACM. https://doi.org/10.1145/3377811.3380327

[12] Jing Cheng, Dingmei Tan, Tao Zhang, Aodi Wei, Jingyi Chen, and Ali Khattak Hasan. 2022. YOLOv5-MGC: GUI Element Identification for Mobile Applications Based on Improved YOLOv5. *Mob. Inf. Syst.* 2022 (jan 2022), 9 pages. https://doi.org/10.1155/2022/8900734

[13] Eliane Collins, Arilo Neto, Auri Vincenzi, and José Maldonado. 2021. Deep Reinforcement Learning based Android Application GUI Testing. In *Proceedings of the XXXV Brazilian Symposium on Software Engineering* (Joinville, Brazil) *(SBES '21)*. Association for Computing Machinery, New York, NY, USA, 186–194. https://doi.org/10.1145/3474624.3474634

[14] Richard Hada Degaki, Juan Gabriel Colonna, Yadini Lopez, José Reginaldo Carvalho, and Edson Silva. 2022. Real Time Detection of Mobile Graphical User Interface Elements Using Convolutional Neural Networks. In *Proceedings of the Brazilian Symposium on Multimedia and the Web* (Curitiba, Brazil) *(WebMedia '22)*. Association for Computing Machinery, New York, NY, USA, 159–167. https://doi.org/10.1145/3539637.3558044

[15] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A Mobile App Dataset for Building Data-Driven Design Applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology* (Québec City, QC, Canada) *(UIST '17)*. Association for Computing Machinery, New York, NY, USA, 845–854. https://doi.org/10.1145/3126594.3126651

[16] Giovanni Denaro, Luca Guglielmo, Leonardo Mariani, and Oliviero Riganelli. 2019. GUI testing in production: challenges and opportunities. In *Companion Proceedings of the 3rd International Conference on the Art, Science, and Engineering of Programming* (Genova, Italy) *(Programming '19)*. Association for Computing Machinery, New York, NY, USA, Article 18, 3 pages. https://doi.org/10.1145/3328433.3328452

[17] Android Developers. 2024. *Espresso*. https://developer.android.com/training/testing/espresso

[18] Android Developers. 2024. *UI Automator*. https://developer.android.com/training/testing/other-components/ui-automator

[19] Juha Eskonen, Julen Kahles, and Joel Reijonen. 2020. Automating GUI Testing with Image-Based Deep Reinforcement Learning. In *2020 IEEE International Conference on Autonomic Computing and Self-Organizing Systems (ACSOS)*. 160–167. https://doi.org/10.1109/ACSOS49614.2020.00038

[20] Anna Esparcia-Alcazar, F. (Francisco) Almenar, Mirella Martınez, U. (Urko) Rueda, and T.E.J. Vos. 2016. Q-learning strategies for action selection in the TESTAR automated testing tool. In *Proceedings of the 6TH International Conference on Metaheuristics and Nature Inspired Computing*. 174–180. https://meta2016.sciencesconf.org/ 6th International Conference on Metaheuristic and Nature inspired Computing, META 2016; Conference date: 27-10-2016 Through 31-10-2016.

[21] OpenJS Foundation. 2012. *Appium*. https://appium.io/

[22] Jerry Gao, ShiTing Li, Chuanqi Tao, Yejun He, Amrutha Pavani Anumalasetty, Erica Wilson Joseph, Akshata Hatwar Kumbashi Sripathi, and Himabindu Nayani. 2022. An Approach to GUI Test Scenario Generation Using Machine Learning. In *2022 IEEE International Conference On Artificial Intelligence Testing (AITest)*. 79–86. https://doi.org/10.1109/AITest55621.2022.00020

[23] Xin Gao, Sundaresh Ram, and Jeffrey J. Rodríguez. 2020. Object sieving and morphological closing to reduce false detections in wide-area aerial imagery. *CoRR* abs/2010.15260 (2020). arXiv:2010.15260 https://arxiv.org/abs/2010.15260

[24] Data Driven Design Group. 2024. *Rico: A Mobile App Dataset for Building Data-Driven Design Applications*. http://www.interactionmining.org/rico.html

[25] Shuai Hao, Bin Liu, Suman Nath, William G.J. Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services* (Bretton Woods, New Hampshire, USA) *(MobiSys '14)*. Association for Computing Machinery, New York, NY, USA, 204–217. https://doi.org/10.1145/2594368.2594390

[26] L. V. Haoyin. 2017. Automatic android application GUI testing—-A random walk approach. In *2017 International Conference on Wireless Communications, Signal Processing and Networking (WiSPNET)*. 72–76. https://doi.org/10.1109/WiSPNET.2017.8299722

[27] Saad Hassan, Manan Arya, Ujjwal Bhardwaj, and Silica Kole. 2018. Extraction and classification of user interface components from an image. *International Journal of Pure and Applied Mathematics* 118, 24 (2018), 1–16.

[28] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Lake Buena Vista, FL, USA) *(ESEC/FSE 2018)*. Association for Computing Machinery, New York, NY, USA, 269–282. https://doi.org/10.1145/3236024.3236055

[29] Jianjun Huang, Zhichun Li, Xusheng Xiao, Zhenyu Wu, Kangjie Lu, Xiangyu Zhang, and Guofei Jiang. 2015. SUPOR: Precise and scalable sensitive user input detection for android apps. In *Proceedings of the 24th USENIX Conference on Security Symposium* (Washington, D.C.) *(SEC'15)*. USENIX Association, USA, 977–992.

[30] IBM. 2021. *Rational Functional Tester*. https://www.ibm.com/support/pages/rational-functional-tester-v1013

[31] K. Jaganeshwari and S. Djodilatchoumy. 2021. A Novel approach of GUI Mapping with image based widget detection and classification. In *2021 2nd International Conference on Intelligent Engineering and Management (ICIEM)*. 342–346. https://doi.org/10.1109/ICIEM51511.2021.9445281

[32] Jinseong Jeon and Jeffrey Foster. 2012. Troyd: Integration Testing for Android. *Technical Report CS-TR-5013, Department of Computer Science, University of Maryland, College Park* (2012).

[33] Yavuz Koroglu, Alper Sen, Ozlem Muslu, Yunus Mete, Ceyda Ulker, Tolga Tanriverdi, and Yunus Donmez. 2018. QBE: QLearning-Based Exploration of Android Applications. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 105–115. https://doi.org/10.1109/ICST.2018.00020

[34] Wing Lam, Zhengkai Wu, Dengfeng Li, Wenyu Wang, Haibing Zheng, Hui Luo, Peng Yan, Yuetang Deng, and Tao Xie. 2017. Record and replay for Android: are we there yet in industrial cases?. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering* (Paderborn, Germany) *(ESEC/FSE 2017)*. Association for Computing Machinery, New York, NY, USA, 854–859. https://doi.org/10.1145/3106237.3117769

[35] Yang Li, Gang Li, Luheng He, Jingjie Zheng, Hong Li, and Zhiwei Guan. 2020. Widget Captioning: Generating Natural Language Description for Mobile User Interface Elements. arXiv:2010.04295

[36] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, Evolutionary and Large-Scale: A New Perspective for Automated Mobile App Testing. 399–410. https://doi.org/10.1109/ICSME.2017.27

[37] Yadini Pérez López, Laís Dib Albuquerque, Gilmar Jóia de F. Costa Júnior, Daniel Lopes Xavier, Juan David Ochoa, and Denizard Dimitri Camargo. 2023. Adapting RICO Dataset for Boosting Graphical User Interface Component Classification for Automated Android Testing. In *2023 10th International Conference on Soft Computing Machine Intelligence (ISCMI)*. 118–123. https://doi.org/10.1109/ISCMI59957.2023.10458576

[38] Yadini Pérez López, Juan G. Colonna, Edson De Araujo Silva, Richard Hada Degaki, and Javier Martinez Silva. 2022. Q-funcT: A Reinforcement Learning Approach for Automated Black Box Functionality Testing. In *2022 IEEE 2nd International Conference on Software Engineering and Artificial Intelligence (SEAI)*. 119–123. https://doi.org/10.1109/SEAI55746.2022.9832177

[39] Ying Ma, ChuYi Yu, and Ming Yan. 2022. Icon Label Generation for Mobile Applications by Mean Teacher Learning. https://doi.org/10.21203/rs.3.rs-1888657/v1

[40] Leonardo Mariani, Mauro Pezzè, Oliviero Riganelli, and Mauro Santoro. 2011. AutoBlackTest: a tool for automatic black-box testing. In *2011 33rd International Conference on Software Engineering (ICSE)*. 1013–1015. https://doi.org/10.1145/1985793.1985979

[41] Forough Mehralian, Navid Salehnamadi, and Sam Malek. 2021. Data-driven accessibility repair revisited: on the effectiveness of generating labels for icons in Android apps. In *Proceedings of the 29th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering* (Athens, Greece) *(ESEC/FSE 2021)*. Association for Computing Machinery, New York, NY, USA, 107–118. https://doi.org/10.1145/3468264.3468604

[42] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2018. *The ReDraw Dataset: A Set of Android Screenshots, GUI Metadata, and Labeled Images of GUI Components*. https://zenodo.org/records/2530277

[43] Kevin Moran, Carlos Bernal-Cárdenas, Michael Curcio, Richard Bonett, and Denys Poshyvanyk. 2020. Machine Learning-Based Prototyping of Graphical User Interfaces for Mobile Apps. *IEEE Transactions on Software Engineering* 46, 2 (2020), 196–221. https://doi.org/10.1109/TSE.2018.2844788

[44] Seong-Guk Nam and Yeong-Seok Seo. 2023. GUI Component Detection-Based Automated Software Crash Diagnosis. *Electronics* 12, 11 (2023). https://doi.org/10.3390/electronics12112382

[45] Clemens Neudecker, Konstantin Baierer, Mike Gerber, Christian Clausner, Apostolos Antonacopoulos, and Stefan Pletschacher. 2021. A survey of OCR evaluation tools and metrics. In *Proceedings of the 6th International Workshop on Historical Document Imaging and Processing* (Lausanne, Switzerland) *(HIP '21)*. Association for Computing Machinery, New York, NY, USA, 13–18. https://doi.org/10.1145/3476887.3476888

[46] Tuan Anh Nguyen and Christoph Csallner. 2015. Reverse Engineering Mobile Application User Interfaces with REMAUI (T). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. 248–259. https://doi.org/10.1109/ASE.2015.32

[47] Rafael Padilla, Sergio L. Netto, and Eduardo A. B. da Silva. 2020. A Survey on Performance Metrics for Object-Detection Algorithms. In *2020 International Conference on Systems, Signals and Image Processing (IWSSIP)*. 237–242. https://doi.org/10.1109/IWSSIP48289.2020.9145130

[48] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 153–164. https://doi.org/10.1145/3395363.3397354

[49] Mahmood R, Mirzaei N, and Malek S. 2014. Automated Test Generation for Graphical User Interfaces: Challenges and Opportunities. *ACM Computing Surveys (CSUR)* (2014), 45.

[50] Dillon Reis, Jordan Kupec, Jacqueline Hong, and Ahmad Daoudi. 2024. Real-Time Flying Object Detection with YOLOv8. (2024). https://doi.org/10.48550/arXiv.2305.09972

[51] Ariel Rosenfeld, Odaya Kardashov, and Orel Zang. 2018. Automation of Android Applications Functional Testing Using Machine Learning Activities Classification. In *2018 IEEE/ACM 5th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*. 122–132.

[52] Xiaolei Sun, Tongyu Li, and Jianfeng Xu. 2020. UI Components Recognition System Based On Image Understanding. In *2020 IEEE 20th International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. 65–71. https://doi.org/10.1109/QRS-C51114.2020.00022

[53] Amrita Vazirani. 2022. *What is Android UI Testing?* https://www.browserstack.com/guide/what-is-android-ui-testing

[54] Tuyet Vuong and Shingo Takada. 2019. Semantic analysis for deep Q-network in android GUI testing. In *Proceedings - SEKE 2019 (Proceedings of the International Conference on Software Engineering and Knowledge Engineering, SEKE)*. Knowledge Systems Institute Graduate School, 123–128. https://doi.org/10.18293/SEKE2019-080 31st International Conference on Software Engineering and Knowledge Engineering, SEKE 2019 ; Conference date: 10-07-2019 Through 12-07-2019.

[55] Thi Anh Tuyet Vuong and Shingo Takada. 2018. A reinforcement learning based approach to automated testing of Android applications. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation* (Lake Buena Vista, FL, USA) *(A-TEST 2018)*. Association for Computing Machinery, New York, NY, USA, 31–37. https://doi.org/10.1145/3278186.3278191

[56] Bryan Wang, Gang Li, Xin Zhou, Zhourong Chen, Tovi Grossman, and Yang Li. 2021. Screen2Words: Automatic Mobile UI Summarization with Multimodal Learning. arXiv:2108.03353

[57] Thomas D. White, Gordon Fraser, and Guy J. Brown. 2019. Improving random GUI testing with image-based widget detection. In *Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Beijing, China) *(ISSTA 2019)*. Association for Computing Machinery, New York, NY, USA, 307–317. https://doi.org/10.1145/3293882.3330551

[58] Xusheng Xiao, Xiaoyin Wang, Zhihao Cao, Hanlin Wang, and Peng Gao. 2019. IconIntent: Automatic Identification of Sensitive UI Widgets Based on Icon Classification for Android Apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. 257–268. https://doi.org/10.1109/ICSE.2019.00041

[59] Mulong Xie, Sidong Feng, Zhenchang Xing, Jieshan Chen, and Chunyang Chen. 2020. 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* (Virtual Event, USA) *(ESEC/FSE 2020)*. Association for Computing Machinery, New York, NY, USA, 1655–1659. https://doi.org/10.1145/3368089.3417940

[60] Feng Xue. 2020. Automated mobile apps testing from visual perspective. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis* (Virtual Event, USA) *(ISSTA 2020)*. Association for Computing Machinery, New York, NY, USA, 577–581. https://doi.org/10.1145/3395363.3402644

[61] Shengcheng Yu, Chunrong Fang, Tongyu Li, Mingzhe Du, Xuan Li, Jing Zhang, Yexiao Yun, Xu Wang, and Zhenyu Chen. 2021. Automated Mobile App Test Script Intent Generation via Image and Code Understanding. arXiv:2107.05165 [cs.SE]

[62] Xiaoyi Zhang, Lilian de Greef, Amanda Swearngin, Samuel White, Kyle Murray, Lisa Yu, Qi Shan, Jeffrey Nichols, Jason Wu, Chris Fleizach, Aaron Everitt, and Jeffrey P. Bigham. 2021. Screen Recognition: Creating Accessibility Metadata for Mobile Applications from Pixels. arXiv:2101.04893