Proceedings Paper:
White, T., Fraser, G. and Brown, G. orcid.org/0000-0001-8565-5476 (2019) Improving random GUI testing with image-based widget detection. In: ISSTA 2019 Proceedings of the 28th ACM SIGSOFT International Symposium on Software Testing and Analysis. ISSTA 2019 - 28th ACM SIGSOFT International Symposium on Software Testing and Analysis, 15-19 Jul 2019, Beijing, China. ACM Digital Library , pp. 307-317. ISBN 9781450362245

https://doi.org/10.1145/3293882.3330551

Improving Random GUI Testing with Image-Based Widget Detection

Thomas D. White
tdwhite1@sheffield.ac.uk
Department of Computer Science,
The University of Sheffield
Sheffield, United Kingdom

Gordon Fraser
gordon.fraser@uni-passau.de
Chair of Software Engineering II,
University of Passau
Passau, Germany

Guy J. Brown
g.j.brown@sheffield.ac.uk
Department of Computer Science,
The University of Sheffield
Sheffield, United Kingdom

ABSTRACT

Graphical User Interfaces (GUIs) are one of the most common user interfaces, enabling interactions with applications through mouse movements and key presses. Tools for automated testing of GUIs exist, however they usually rely on operating system specific or framework specific knowledge in order to interact with the application under test. Because of frequent operating system updates and a large variety of GUI frameworks, such tools are made obsolete by time. For an automated GUI test generation tool, supporting many frameworks and operating systems is impractical; new operating system updates can remove required information and each different GUI framework uses unique underlying data structures. We propose a technique for improving random GUI testing by automatically identifying GUI widgets in screen shots using machine learning techniques. This information provides guidance to GUI testing tools in environments not currently supported by deriving GUI widget information from screen shots only. In our experiments, we found that identifying GUI widgets from screen shots and using this information to guide random testing achieved a significantly higher branch coverage in 18 of 20 applications, with an average increase of 42.5% compared to conventional random testing.

KEYWORDS

GUI testing, object detection, black box testing, software engineering, data generation, convolutional neural networks

1 INTRODUCTION

A Graphical User Interface (GUI) enables events to be triggered in an application through visual entities called widgets (e.g., buttons). Using keyboard and mouse, users interact with the widgets on a GUI to fire events in the application. Automated GUI test generation tools (e.g., AutoBlackTest [10], Sapienz [9], or GUITAR [11]) simulate users by interacting with the widgets of a GUI, and they are increasingly applied by companies to test mobile and desktop applications. The effectiveness of these GUI test generation tools depends the information they have available. A naïve GUI test generator simply clicks on random screen positions. However, if a GUI test generator knows the locations and types of widgets on the current application screen, then it can make better informed choices about where to target interactions with the program under test.

GUI test generation tools tend to retrieve the information about available GUI widgets through the APIs of the GUI library of the target application, or the accessibility API of the operating system. However, relying on these APIs has drawbacks: Applications can be written using many different GUI libraries and widget sets, each providing a different API to access widget information. Although this can be circumvented by accessibility APIs, these differ between operating systems, and updates to an Operating System can remove or replace parts of the API. Furthermore, some applications may not even be supported by such APIs, such as those which draw directly to the screen, e.g., web canvasses [1]. These challenges make it difficult to produce and to maintain testing tools that rely on GUI information. Without knowledge of GUI widgets, test generation tools have to resort to blindly interacting with random screen locations.

To relieve GUI testing tools of the dependency on GUI and accessibility APIs, in this paper we explore the use of machine learning techniques in order to identify GUI widgets. A machine learning model is trained to detect the widget types and positions on the screen, and this information is fed to a test generator which can then make more informed choices about how to interact with a program under test. However, generating a widget prediction model is non-trivial: Different GUI libraries and operating systems use different visual appearance of widgets. Even worse, GUIs can often be customized with user-defined themse, or assistive techniques such as a high/low contrast graphical mode. In order to overcome this challenge, we randomly generate Java Swing GUIs, which can be annotated automatically, as training data. The final machine learning model uses only visual data and can identify widgets in real application GUIs without needing additional information from an operating system or API.

In detail, the contributions of this paper are as follows:

- We describe a technique to automatically generate GUIs in large quantities, in order to serve as training data for GUI widget prediction.
- We describe a technique based on deep learning that adapts machine learning object detection algorithms to the problem of GUI widget detection.
- We propose an improve random GUI testing approach that relies on no external GUI APIs, and instead selects GUI interactions based on a widget prediction model.
- We empircally investigate the effects of using GUI widget prediction on random GUI testing.

In our experiments, for 18 out of 20 Java open source applications tested, a random tester guided using predicted widget locations achieved a significantly higher branch coverage than a random tester without guidance, with an average coverage increase of 42.5%. Although our experiments demonstrate that the use of an API that provides the true widget details can lead to even higher coverage, such APIs are not always available. In contrast, our widget prediction library requires nothing but a screenshot of the application, and even works across different operating systems.

2 BACKGROUND

Interacting with applications through a GUI involves triggering events in the application with mouse clicks or key presses. Lo et al. [8] define three types of widgets in a GUI:

- Static widgets in a GUI are generally labels or tooltips.
- Action widgets fire internal events in an application when interacted with (e.g. buttons).
- Data widgets are used to store data (e.g., text fields).

In this paper, we focus on identifying action and data widgets.

The simplest approach to generating GUI tests is through clicking on random places in the GUI window [6], hoping to hit widgets by chance. This form of testing ("monkey testing") is effective at finding crashes in applications and is cheap to run; no information is needed (although knowing the position and dimensions of the application on the screen is helpful). Monkey is now also commonly used to test mobile applications through tools like the Android Monkeyrunner [5].

GUI test generation tools can be made more efficient by providing them with information about the available widgets and events. This information can be retrieved using the GUI libraries underlying the widgets used in an application, or through the operating system's accessibility API. For example, Bauersfeld and Vos created GUITest [2] (now known as TESTAR), which uses the MacOSX Accessibility API to identify possible GUI widgets to interact with, and then randomly chooses from the available widgets during test generation. TESTAR has been applied to many industrial applications, including a web-based application from the rail sector [4]. The AutoBlackTest [10] relies on a commercial testing tool (IBM Rational Functional Tester) to retrieve widget information, and then uses Q-Learning to select the most promising widgets for interaction.

In contrast to these randomized approaches, GUI ripping [11] aims to identify *all* GUI widgets in an application to permit systematic test genration. However, a recent study of by Nguyen et al [12] found that, although GUI ripping enables effective testing and flexible support for automation, there are drawbacks mainly related to the GUI ripping. For example, GUI trees have to be manually validated, and component identification issues can lead to inaccurate GUI trees being generated.

The problem of widget identification is not only relevant for test input generation, but also for asserting the test outcome. For example, the Sikuli [18] tool uses OpenCV, an image processing library, to match images of GUI widgets saved in a test against the current application's GUI. Matching an image of the button decreases the chance of tests failing if the application's GUI changes intentionally (e.g. by moving the button from the top of a GUI to the bottom). Sikuli also features assertions, in which tests can check that some part of the GUI exists after performing an interaction on the application under test. However, Sikuli tries to exactly match images of previously seen widgets, and therefore cannot be used to identify new, not previouly seen widgets.

With the exception of randomized testing, all the current approaches rely on an automated method of extracting widget information from a GUI. There exists applications and application scenarios where widget information cannot be automatically derived, and image labelling may be able to help with this.

3 PREDICTING GUI WIDGETS FOR TEST GENERATION

In order to improve random GUI testing, we aim to identify widgets in screen shots using machine learning techniques. A challenge lies in retrieving a sufficiently large labelled training dataset to enable modern object recognition approaches to be applied. We produce this data by (1) generating random Java Swing GUIs, and (2) labelling screenshots of these applications with widget data retrieved through GUI ripping based on the Java Swing API. The trained network can then predict the location and dimensions of widgets from screen shots during test generation, and thus influence where and how the GUI tester interacts with the application.

3.1 Identifying GUI Widgets

Environmental factors such as operating system, user-defined theme, or application designer choice effect the appearance of widgets. Each application can use a unique widget palette. When widget information cannot be extracted through use of external tools, e.g., an accessibility API, then this diversity of widgets presents a problem for GUI testing tools. Applications that render GUIs directly to an image buffer (e.g., web canvas applications) generally cannot have their GUI structure extracted automatically. Pixels are drawn directly to the screen and there is no underlying XML or HTML structure to extract widget locations. We propose a technique of identifying GUI widgets solely through visual information. This is an instance of image labelling, i.e., the process of automatically extracting and "tagging" parts of an image.

Some methods such as Region-based Convolutional Neural Network (R-CNN) by Girshick et al [7] work by selecting areas of the image to input through the neural network. Convolutional Neural Networks (CNN) identify patterns in images and can be expensive to compute, especially if a sliding window inputs subsets of the image through a CNN multiple times. During GUI testing, we need to be able to recognize widgets quickly. Therefore, we use You Only Look Once (YOLO), proposed by Redmon et al. [13], which labels an image by seeding the whole image through a CNN once. YOLO is capable of predicting the positions and dimensions of objects in an image.

The input to YOLO is an image with width and height being a multiple of 32 pixels and equal in value. To predict labels, YOLO continuously downsamples the input image into $N \cdot N$ grid cells, where N is the width or height divided by 32 in the last layer. For example, if the input dimension is $(416, 416)$, YOLO will predict widgets in a $(13, 13)$ grid. YOLOv2 [14] is an extension to YOLO. YOLOv2 predicts B boxes per grid cell. A cell is responsible for a prediction if the centre of a box falls inside the dimensions of the respective cell.

Each box contains five predicted values: the location (x, y), the dimension $(width, height)$ and a confidence score for the prediction (c). Predicting multiple boxes per grid cell aids in training as

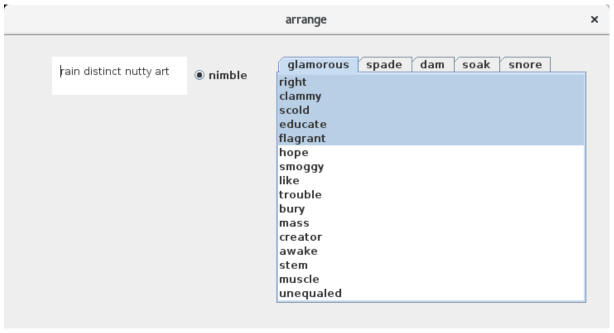


Figure 1: A generated Java Swing GUI. Each element has a random chance to appear.

Algorithm 3.1: RandomWidgetTree(*nodeCount*)

```
nodes = [Container]
while |nodes| < nodeCount
  do { nodes ← nodes ⋃ RandomWidgetType()
while (|nodes| > 1)
  do { node ← sample(nodes, 1)
       parent ← sample(nodes, 1)
       if isContainer(parent) and node ≠ parent
         then { parent.children ← parent.children ⋃ node
                nodes ← nodes \ node
return (nodes[0])
```

it allows different aspect ratios to be used for each box in each cell. The aspect ratios are passed to the algorithm and multiply the predicted width and height of each box. To calculate the aspect ratios to use, the dimensions of all boxes in the training data are clustered into N clusters, where N is the number of boxes predicted per cell. The centroid of each cluster gives the respective aspect ratios to supply to YOLO.

A single class is predicted from C predefined classes for each grid cell. In total, this makes the network's output $N \cdot N \cdot (B \cdot 5 + C)$. We can now filter the predicted boxes using the confidence values. Boxes with confidence values close to zero may not be worth investigating.

Using the YOLO convolutional neural network, we can automatically identify GUI components in a screen shot. We chose the YOLO algorithm for our network due to the speed it can process entire images, and the accuracy it achieves on predictions. Our implementation of YOLO only uses black and white images, so the first layer of the network only has a single input per pixel opposed to the three (r, g, b) values proposed in the original YOLO paper [13].

3.2 Generating Synthetic GUIs

One issue with using a neural network is that it requires large amounts of labelled data for training. To obtain labelled screen shots, we generate synthetic applications. A synthetic application is one with no event handlers, containing only a single screen with random placements of widgets. We use 11 standard types of widgets in generated GUIs, which are shown in Table 1.

Algorithm 3.2: RandomJFrame(*width*, *height*, *nodeCount*)

```
procedure ApplyWidget(container,widget)
  swingComponent ← ComponentFromWidget(widget)
  i ← 0
  while i < |widget.children|
    do { child ← widget.children[0]
         ApplyWidget(swingComponent, child)
         i ← i + 1
  container.add(swingComponent)

jframe ← JFrame<init>(width, height)
rootNode ← RandomWidgetTree(nodeCount)
ApplyWidget(jframe, rootNode)
return (jframe)
```

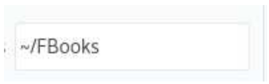
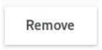
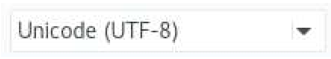
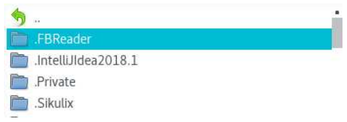
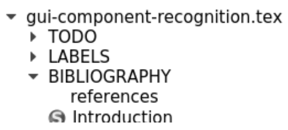
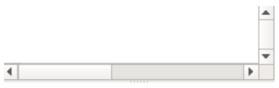
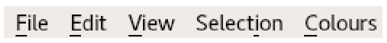
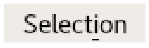
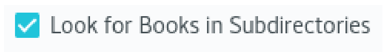
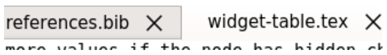
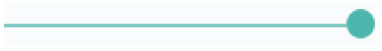
To generate synthetic applications, we use the Java Swing GUI framework. Initial attempts at generating GUI by entirely random selection and placement of widgets yielded one-dimensional GUIs and poor performance of the resulting prediction model. To create more realistic GUIs, our approach therefore generates an abstract tree beforehand, and uses this tree as a basis for the generated GUI.

First, we randomly choose a Swing layout manager and then generate a random tree where each node represents a GUI widget. Only widgets which can contain other widgets can be assigned child nodes in the tree, for example, a tab pane can have children representing other GUI widgets assigned to it, but a button cannot. Algorithm 3.1 shows how a random abstract tree of GUI widget types is generated. Here, the nodes list initially contains a "Container" widget type which is to eliminate an infinite loop later if the $RandomWidgetType()$ function returns no containers. The call to $RandomWidgetType$ randomly returns one of the 11 types of widgets. Each widget has the same probability of appearing but we found that some GUI widgets are constructed of others when using Java Swing, e.g. a Combo Box also contains a button, and a scroll bar contains two buttons, one at each end. However, we lowered the probability for *menu_items* to appear as a *menu_item* requires a corresponding *menu*. We found that weighting *menu_items* with an equal probability to appear made *menus* appear on nearly all generated GUIs.

To generate a Swing GUI, Algorithm 3.2 walks through the generated tree. Each node is assigned a random position and dimension inside its parent. The position of a node is randomly selected based on the layout manager. For example, with a GridLayout, we randomly assign the element in the current node to a random (x, y) coordinate in the grid. However, with a FlowLayout, the position does not matter as all widgets appear side by side in a single line. In algorithm 3.2 , the $container.add$ method call in the $ApplyWidget$ procedure is from the JComponent class of Java Swing, and the random position is seeded here depending on the current LayoutManager.

Once a Swing GUI has been generated, Java Swing allows us to automatically extract information for all widgets. This includes position on screen, dimension and widget type. This is similar to the approach current GUI testing tools use when interacting with an application during test execution.

Table 1: Widgets that the model can identify in a Graphical User Interface

Widget	Description	Example
Text Field	Allows input from the keyboard to be stored in the GUI to be used later.	
Button	Allows an event to be triggered by clicking with the mouse.	
Combo Box	Allows selection of predefined values. Clicking either inside the box or the attached button opens predefined options for users to select.	
List	Allows selection of predefined values, similar to a Combo Box, but the values are present at all times. Scrolling may be needed to reveal more values.	
Tree	Similar to a list but values are stored in a tree structure. Clicking a node may reveal more values if the node has hidden child elements.	 gui-component-recognition.tex TODO LABELS BIBLIOGRAPHY references Introduction
Scroll Bar	A horizontal or vertical bar used for scrolling with the mouse to reveal more of the screen.	
Menu	A set of usually textual buttons across the top of a GUI	
Menu Item	An individual button in a menu. Clicking usually expands the menu revealing more interactable widgets.	
Toggle Button	Buttons that have two states toggled by clicking on them.	
Tabs	Buttons which change the contents in all or part of the GUI when clicked.	
Slider	A button that can be click-and-dragged in a certain axis, changing some value which is usually a numeric scale e.g. volume of a music application.	

3.3 A Random Bounding Box Tester

Once widgets are identified, they are used to influence a random GUI tester. We created a tester which randomly clicks inside a given bounding box. At the most basic level, a box containing the whole application GUI is provided, and the tester will randomly interact with this box. One of three actions is executed on the selected box: a) left click anywhere inside the given box; b) right click anywhere inside the given box; c) left click anywhere inside the given box and type either a random string (e.g., "Hello World!" in our implementation) or a random number (e.g., between -10000 and 10000 in our implementation). We use these two textual inputs to represent the most common use for text fields: storing a string of characters or storing a number. Algorithm 3.3 shows the algorithm for interacting with a box given to the GUI tester. In this algorithm, $rand(x, y)$returns a random number between x and y inclusive. $LeftClick(x, y)$ and $RightClick(x, y)$ represent moving the mouse to position x, y on the screen and either left or right clicking respectively. $KeyboardType(string)$ represents pressing the keys present in string in chronological order.

Algorithm 3.3: RANDOMINTERACTION(*box*)

```
interaction ← rand(0, 2)
x ← box.x + rand(0, box.width)
y ← box.y + rand(0, box.height)
if interaction == 0
  then LEFTCLICK(x, y)
  else if interaction == 1
  then RIGHTCLICK(x, y)
  else if interaction == 2
  then
        LEFTCLICK(x, y)
        inputType ← rand(0, 1)
        inputString ← ""
        if inputType == 0
          then inputString ← "Hello World!"
          else
            inputNumber ← rand(-10000, 10000)
            inputString ← inputNumber.toString()
        KEYBOARDTYPE(inputString)
```

We can refine the box provided to this random tester. Using the trained YOLO network, we can select a random box with a confidence greater than some value C. When seeded to the tester, the tester will randomly click inside one of the predicted widgets from the network.

Finally, we can provide the tester with a box directly from Java Swing. This implementation currently only supports Java Swing applications but will ensure that the GUI tester is always clicking inside the bounding box of a known widget currently on the screen.

4 EVALUATION

To evaluate the effectiveness of our approach when automatically testing GUIs, we investigate the following research questions:

- RQ1 How accurate is a model trained on synthetic GUIs when identifying widgets in GUIs from real applications?
- RQ2 How accurate is a model trained on synthetic GUIs when identifying widgets in GUIs from other operating system and widget palettes?
- RQ3 What benefit does random testing receive when guided by predicted locations of GUI widgets from screen shots?
- RQ4 How close can random guided by predicted widget locations come to an automated tester guided by the exact positions of widgets in a GUI?

4.1 Model Training

In order to create the prediction model, we created synthetic GUIs on Ubuntu 18.04, and to capture different GUI styles, we used different operating system themes. We generated 10,000 GUI applications per theme and used six light themes: the default Java Swing theme, adapta, adwaita, arc, greybird; two dark themes: adwaita-dark, arc-dark, and two high contrast themes which are default with Ubuntu 18.04. These are all popular themes for Ubuntu and were chosen so that the pixel histograms of generated GUI images were similar to that of real GUI images.

In total this resulted in 100,000 synthetic GUIs, which we split as follows: 80% of data was used as training data, 10% as validation data, and 10% as testing data. To train a model using this data, the screen shots are fed through the YOLO network and the predicted boxes from the network are compared against the actual boxes retrieved from Java Swing. If there is a difference, the weights of the model are updated so next time the prediction will be more accurate.

It is important to have a validation dataset to determine whether the model is over-fitting on the training data. This can be done by checking the training progress of the model against the training and validation dataset. During training, the model is only exposed to the training dataset, so if the model is improving when evaluated against the training dataset, but not improving on the validation dataset, the model is over-fitting.

Afterwards, we trained a model which uses the YOLOv2 network. During training, we artificially increased the size of input data using two techniques: brightness and contrast adjustment. Each image has a 10% chance to be manipulated for each of these adjustments and the adjustment was a random shift of 10%. For example, an image could be made up to 10% lighter/darker and have the pixel intensity values moved up to 10% closer/further from the median of the image's intensity values. Artificial data inflation allows detection of widgets using a wider range of themes by exposing the network to more varied data during training.

4.2 Experimental Setup

4.2.1 RQ1. To evaluate RQ1, we compare the performance when predicting GUI widgets in 250 screen shots of real applications against performance when predicting widgets in synthetic applications. 150 of the screen shots were taken from the top 20 Swing applications on SourceForge and annotated via the Swing API. The remaining 100 screen shots were taken from the top 15 applications on the Ubuntu software center and manually annotated. The model used to predict widget locations was trained on only synthetic GUIs, and in RQ1 we see if the model is able to make predictions for real applications.

YOLO predicts many boxes, which could cause a low precision. To lower the amount of boxes predicted, we pruned any predicted boxes below a certain confidence threshold. To tune this confidence threshold, we evaluated different confidence values against the synthetic validation dataset. As recall is more important to us than precision, we used the confidence value with the highest F2-measure to compare synthetic against real application screen shots. We found this value C to be 0.1 for through parameter tuning on the synthetic validation dataset.

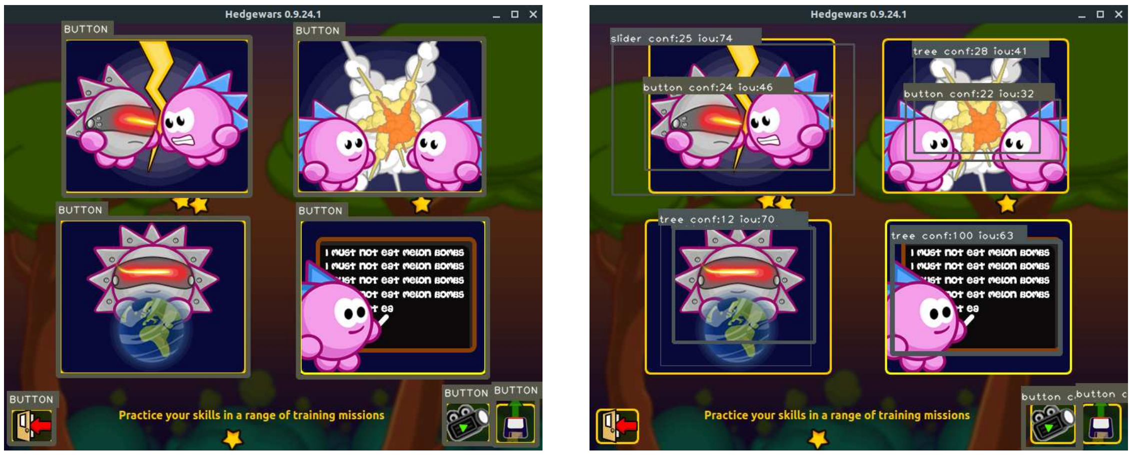
In order to assess whether a predicted box correctly matches with an actual box in a GUI, we present two metrics. Both metrics use only predicted boxes with a confidence value greater than C.

The first metric (called "center point") involves checking if the center of a predicted bounding box falls inside the actual bounding box. Precision is the number of predicted boxes with a center point inside an actual box, divided by the number of all predicted boxes. Recall for this technique is the number of actual boxes with the center point of a predicted box inside it, divided by the total number of actual boxes.

The second metric involves using the Intersection-over-union (IoU) between the predicted and actual boxes. The IoU of two boxes is the area that the boxes intersect, divided by the union of both areas. An IoU value of one indicates that the boxes are identical, having an equal intersection and union. An IoU of 0 indicates the boxes have no area of overlap. See Figure 3 for an example of IoU values for overlapping boxes. The shaded area indicates overlap between both boxes. Precision is the number of predicted boxes which have a corresponding actual box with an IoU greater than 0.3 divided by the number of predicted boxes. Recall is the number of actual boxes which have a corresponding predicted box with an IoU greater than 0.3, divided by the total number of actual boxes.

We use two metrics because of the features that a box is predicting. If a box has a well suited center point, then the first metric will perform well. If the center point and the dimension prediction is accurate, then the second metric will also perform well. An inaccurate dimension prediction is indicated by the second metric performing worse than the first, and an inaccurate box center prediction will cause both metrics to perform poorly.

4.2.2 RQ2. To evaluate RQ2, we use the same principle as in RQ1. However, the comparison datasets are the synthetic dataset



(a) Manually Annotated (b) Predicted Annotations

Figure 2: Manually annotated (a) and predicted (b) boxes on the Ubuntu application "Hedge Wars".

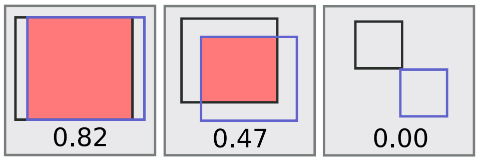


Figure 3: Intersection over Union values for various overlapping boxes.

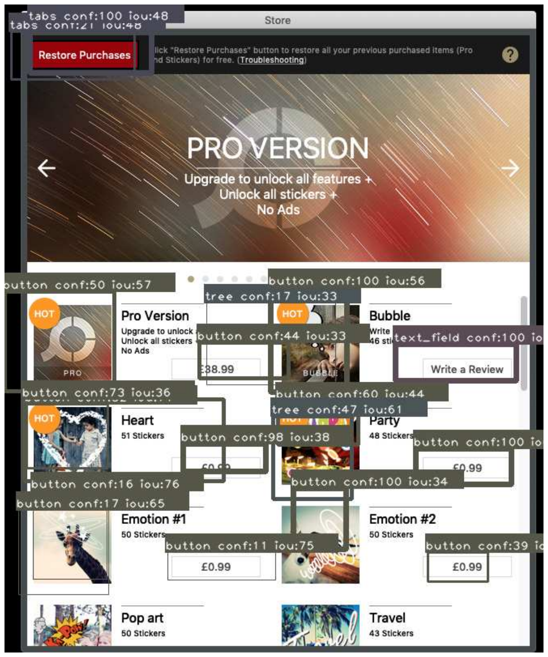


Figure 4: Predicted bounding boxes on the OSX application "Photoscape X"

and a set of screen shots taken from the Apple store and manually annotated. We gathered 50 screen shots, five per application of the top 10 applications on the store.

4.2.3 RQ3. To evaluate RQ3, we compare the branch coverage of tests generated by a random clicker to tests where the clicker is guided by predicted bounding boxes. The subject under test are 20 Java Swing applications, taken from SourceForge, GitHub or personal project pages. We limited the random tester to 1000 actions. On a crash or application exit, the application under test was restarted. Each technique was applied 30 times on each of the applications. Although all the applications use Java Swing, this was to aid conducting experiments when measuring branch coverage and allow retrieval of the positions of widgets currently on the screen from the Java Swing API. Our approach should work on many kinds of applications using any operating system.

4.2.4 RQ4. To answer RQ4, we compare the branch coverage of tests generated by a random clicker guided by predicted bounding boxes, to a random clicker guided by the known locations of widgets retrieved from the Java Swing API. We use the same applications as RQ3. We allowed each tester to execute 1000 actions. On a crash or application exit,the application under test is restarted. Each technique ran on each application for 30 iterations.

4.3 Threats to Validity

There is a chance that our model over-trains on the training and validation synthetic GUI dataset and therefore achieves a high precision and recall on these datasets. To counteract this, we use the third test dataset when calculating precision and recall values for the synthetic dataset which has been completely isolated from the training procedure.

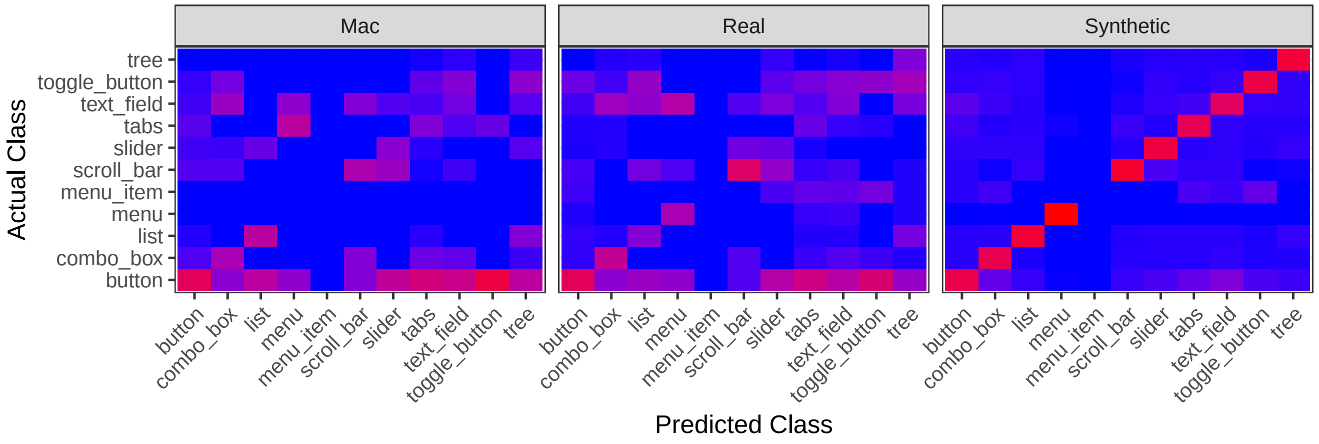


Figure 5: Confusion Matrix for class predictions

Table 2: The applications tested when comparing the three testing techniques.

Application	Description	LOC	Branches
Address Book	Contact recorder	363	83
bellmanzadeh	Fuzzy decision maker	1768	450
BibTex Manager	Reference Manager	804	309
BlackJack	Casino card game	771	178
Dietetics	BMI calculator	471	188
DirViewerDU	View directories and size	219	90
JabRef	Reference Manager	60620	23755
Java Fabled Lands	RPG game	16138	9263
Minesweeper	Puzzle game	388	155
Mobile Atlas Creator	Create offline atlases	20001	5818
Movie Catalog	Movie journal	702	183
ordrumbox	Create mp3 songs	31828	6064
portecle	Keystore manager	7878	2543
QRCode Generator	Create QR codes for links	679	100
Remember Password	Save account details	296	44
Scientific Calculator	Advanced maths calculator	264	62
Shopping List Manager	List creator	378	62
Simple Calculator	Basic maths calculator	305	110
SQuiz	Load and answer quizzes	415	146
UPM	Save account details	2302	530

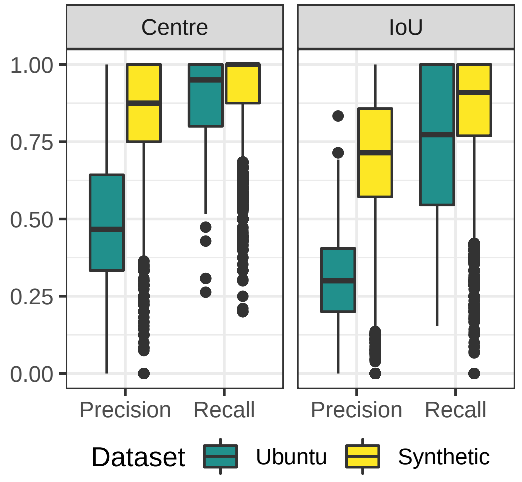


Figure 6: Precision and recall of synthetic data against real GUIs on Ubuntu/Java Swing using two different metrics: center point or IoU > 0.3.

To ensure that our real GUI screen shot corpus represents general applications, the Swing screenshots were from the top applications on SourceForge, the top rated applications on the Ubuntu software center, and the top free applications from the Apple Store.

In object detection, usually an IoU value of 0.5 or more is used for predicted box to be considered a true positive. However, we use an IoU threshold of 0.3 as the predicted box does not have to exactly match the actual GUI widget box, but it needs enough overlap to enable interaction. Russakovsky et al. found that training humans to differentiate between bounding boxes with an IoU value of 0.3 or 0.5 is challenging [16], so we chose the lower threshold of 0.3.

As the GUI tester uses randomized processes, we ran all configurations on all applications for 30 iterations. We used a two-tail test to compare each technique and a Vargha-Delaney A_{12} to find the best technique.

4.4 Results

4.4.1 RQ1: How accurate is a model trained on synthetic data when detecting widgets in real GUIs? Figure 6 shows the precision and recall using the two different metrics: center point and IoU. We can see that predicting widgets on screen shots of Ubuntu and Java Swing GUIs achieves a lower precision and recall than on synthetic GUIs. However, most widgets are identified as shown by a high recall value. A low precision but high recall could indicate that we are predicting too many widgets in each GUI screen shot. Figure 2a shows an example of a manually annotated image, and Figure 2b shows the same screen shot but with predicted widget boxes.

Currently we have only evaluated if the predicted box aligns with an actual box. Figure 5 shows the confusion matrix for class

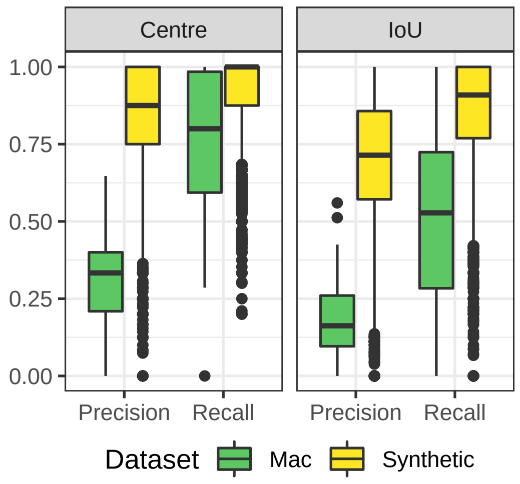


Figure 7: Precision and recall of synthetic data against real GUIs on Mac OSX using two different metrics: center point or IoU > 0.3

predictions. A red square indicates a high proportion of predictions, and blue low. We can see that for synthetic applications, most class predictions are correct. However, the model struggles to identify *menu_items* and this is most likely due to the lower probability of them appearing in synthesized GUIs. The network would rather classify them as a button which appears much more commonly through all synthesized GUIs.

From the confusion matrix, another problem for classification seems to be buttons. Buttons are varied in shape, size and foreground. For example, a button can be a single image, a hyper-link, or text surrounded by a border. Subtle modifications to a widget can change how a user perceives the widget's class, but are much harder to detect automatically.

While this shows that there is room for improvement of the prediction model, these improvements are not strictly necessary for the random tester as described in Section 3.3, since it interacts with all widgets in the same manner irrespective of the predicted type. Hence, predicting the correct class for a widget is not as important as identifying the actual location of a GUI, which our approach achieves. However, future improvements of the test generation approach may rely more on the class prediction.

RQ1: In our experiments, widgets in real applications were detected with an average recall of 95% and 77% for the center and IoU metrics.

4.4.2 RQ2: How accurate is a model trained on synthetic data when detecting widgets on a different operating system? To detect whether widgets can be detected in other operating systems with a different widget palette, we apply a similar approach to RQ1 and use the same two metrics, evaluated on screenshots taken on a different operating system and different applications.

Figure 7 shows the precision and recall using the two different metrics: center point and IoU. We again see a lower precision and recall on screen shots of OSX (Mac) GUIs compared to synthetic GUIs, but we still identify above 50% of widgets in these applications.

A lower precision indicates many false positive predictions when using the OSX theme in applications. A large difference between the first and second metric for recall indicates that our technique has difficulty in predicting correct dimensions for bounding boxes on OSX. See Figure 4 for correct predicted boxes with a corresponding matched manually annotated box using the IoU metric in "Photoscape X". However, for the purposes of testing the exact bounding boxes are less relevant as long as the interaction happens somewhere within the bounding box of the actual widget, which is represented by the center point metric. Here, the recall is substantially higher, achieving an average recall of 80% verses 52% for the IoU metric.

RQ2: GUI widgets can be identified in different operating systems using a model trained on widgets with a different theme, achieving an average recall of 80% and 52% for the center and IoU metrics.

4.4.3 RQ3: What benefit does random testing receive when guided by predicting widget positions? Figure 8 shows the branch coverage achieved by the random tester when guided by different techniques. Here we can see that interacting with predicted GUI widgets achieves a significantly higher coverage for 18 of the 20 applications tested. Table 3 shows the mean branch coverage for each technique, where a bold value indicates significance.

Overall, guiding the random tester with predicted widget locations increased coverage by an average of 42.5%. The main coverage increases were in applications with sparse GUIs, like Address Book (24%⟶48%) and Dietetics (20%⟶54%). The predicted widgets also aided the random tester to achieve coverage where complex sequences of events are needed, such as the Bellmanzadeh application (22%⟶28%). Bellmanzadeh is a fuzzy logic application, and requires many fields to be created of different types. Random is unlikely to create many variables of unique types, but when guided by predicted widget locations, is likely to interact with the same widgets again to create more variables.

One notable example here is JabRef, where unguided random achieved 6.6% branch coverage, significantly better than random guided by widget predictions which achieved 5.2%. JabRef is a bibtex reference manager, and by default it starts with no file open. The only buttons accessible are "New File" and "Open". The predicted boxes contain an accurate match for the "Open" button and a weak match for the "New File" button. If the "Open" button is pressed, a Java file browser opens, locking the main JabRef window.

As we randomly select a window to interact with from the available, visible windows, any input into the main JabRef window is ignored until the file browser closes. There are two ways to exit the file browser: clicking the "Cancel" button or location a valid JabRef file and pressing "Open". There are, however, many widgets on this screen to interact with lowering the chance of hitting cancel, and it is near impossible to find a valid JabRef file to open for both the prediction technique and the API technique. Even if the "Cancel" button is pressed, there is a high chance of interacting with the "Open" button again in the main JabRef window.

Table 3: Branch coverage of random guided by no guidance, widget prediction and the Java Swing API. Bold is significance.

Application	**Pred**iction Cov.	**Rand**om Cov.	p_v (Pred, Rand)	$\hat{A}_{12}$(Pred, Rand)	**API** Cov.	p_v (Pred, API)	$\hat{A}_{12}$(Pred, API)
Address-Book	0.484	**0.235**	**<0.001**	**0.032**	**0.370**	**<0.001**	**0.237**
bellmanzadeh	0.276	**0.215**	**<0.001**	**0.048**	**0.425**	**<0.001**	**1.000**
BibTex-Manager	0.214	**0.160**	**<0.001**	**0.145**	**0.347**	**<0.001**	**0.998**
BlackJack	0.355	**0.167**	**<0.001**	**0.143**	**0.848**	**<0.001**	**1.000**
Dietetics	0.544	**0.197**	**<0.001**	**<0.001**	0.564	0.067	0.640
DirViewerDU	0.728	**0.522**	**<0.001**	**<0.001**	**0.576**	**<0.001**	**0.089**
JabRef	0.052	**0.066**	**<0.001**	**0.768**	0.060	0.608	0.540
Java-FabledLands	0.105	**0.056**	**<0.001**	**0.098**	**0.102**	**<0.001**	**0.122**
Minesweeper	0.837	**0.811**	**<0.001**	**0.170**	**0.850**	**<0.001**	**0.859**
Mobile-Atlas-Creator	0.120	**0.059**	**<0.001**	**0.199**	**0.224**	**<0.001**	**1.000**
Movie-Catalog	0.581	**0.328**	**<0.001**	**0.007**	**0.643**	**<0.001**	**0.826**
ordrumbox	0.192	**0.181**	**<0.001**	**0.056**	**0.203**	**<0.001**	**0.905**
portecle	0.063	**0.049**	**<0.001**	**0.121**	**0.106**	**<0.001**	**0.948**
QRCode-Generator	0.673	**0.582**	**<0.001**	**0.024**	**0.658**	**0.010**	**0.304**
Remember-Password	0.333	**0.255**	**<0.001**	**0.182**	**0.535**	**<0.001**	**0.968**
Scientific-Calculator	0.588	**0.469**	**<0.001**	**0.129**	**0.863**	**<0.001**	**1.000**
ShoppingListManager	0.758	**0.563**	**<0.001**	**0.032**	0.758	1.000	0.500
Simple-Calculator	0.769	**0.460**	**<0.001**	**<0.001**	**0.864**	**<0.001**	**1.000**
SQuiz	0.111	0.111	1.000	0.500	**0.130**	**<0.001**	**1.000**
UPM	0.125	**0.060**	**<0.001**	**0.040**	**0.460**	**<0.001**	**0.986**
Mean	0.395	0.277	0.050	0.135	0.479	0.084	0.746

On the other hand, the random technique has a low chance of hitting the "Open" button. When JabRef starts, the "New" button is focused. We repeatedly observe the random technique click anywhere in the tool bar and type "Hello World!". As soon as it pressed the space key, a new JabRef project would open. This then unlocks all the other buttons to interact with in the JabRef tool bar

RQ3: In our experiments, widget prediction significantly increased branch coverage by an average of 42.5% over random testing.

4.4.4 RQ4: How close can random guided by predicted widget locations come to an automated tester guided by the exact positions of widgets in a GUI?. Using GUI ripping to identify actual GUI widget locations serves as a "golden" model of how much testing could be improved with a perfect prediction model. Therefore, Figure 8 also shows branch coverage for a tester guided by widget positions extracted from the Java Swing API. It is clear that whilst predicted widget locations aids the random tester in achieving a higher branch coverage, unsurprisingly, using the positions of widgets from an API is still superior. This suggests that there is still room for improving the prediction model further.

However, notably there are cases where the widget prediction technique *improves* over using the API positions. One such case is DirViewerDU. This is an application consisting of only a single tree spanning the whole width and height of the GUI. If a node in the tree is right clicked, a pop-up menu appears containing a custom widget not supported or present in the API widget positions. However, the prediction approach correctly identifies this as an interactable widget and interacts with it.

Another application is the Address Book application. Both guidance techniques lead the application into a state with two widgets: a text field and a button. To leave this GUI state, text needs to be typed in the text field and then the button needs to be clicked. If no button is clicked, an error message is shown and the GUI state remains the same. However, the information of the text field is not retrieved by the Swing API as it is a custom widget. The API guided approach then spends the rest of the testing budget clicking the button, producing the same error message. Predicted widget guidance identifies the text field, and can leave this state to explore more of the application.

RQ4: Exploiting the known locations of widgets through an API achieves a significantly higher branch coverage than predicted locations, however widget prediction can identify and interact with custom widgets not detected by the API.

5 RELATED WORK

Bajammal et al. undertook previous work in generating assertions for web canvasses [1]. Normally, web pages have a Document Object Model (DOM), however web canvasses do not have this underlying exploitable structure so have limited testability for current testing tools. For web canvasses, applications draw directly to a buffer representing the screen contents seen by users.

The approach by Bajammal et al. identified common shapes in web canvasses and could generate assertions from identified shapes. This was achieved through multiple image processing steps. Although shapes could be identified, identification and classification of GUI widgets is different. The approach used by Bajammal et al. may be able to identify shapes in GUI widgets, such as squares for text fields, but would not be able to differentiate between GUI widget classes.

However, we may be able to use this work to generate assertions. We predict rectangles that contain widgets on a GUI. Hence, we can directly apply the assertion technique to generate assertions on the application under test.

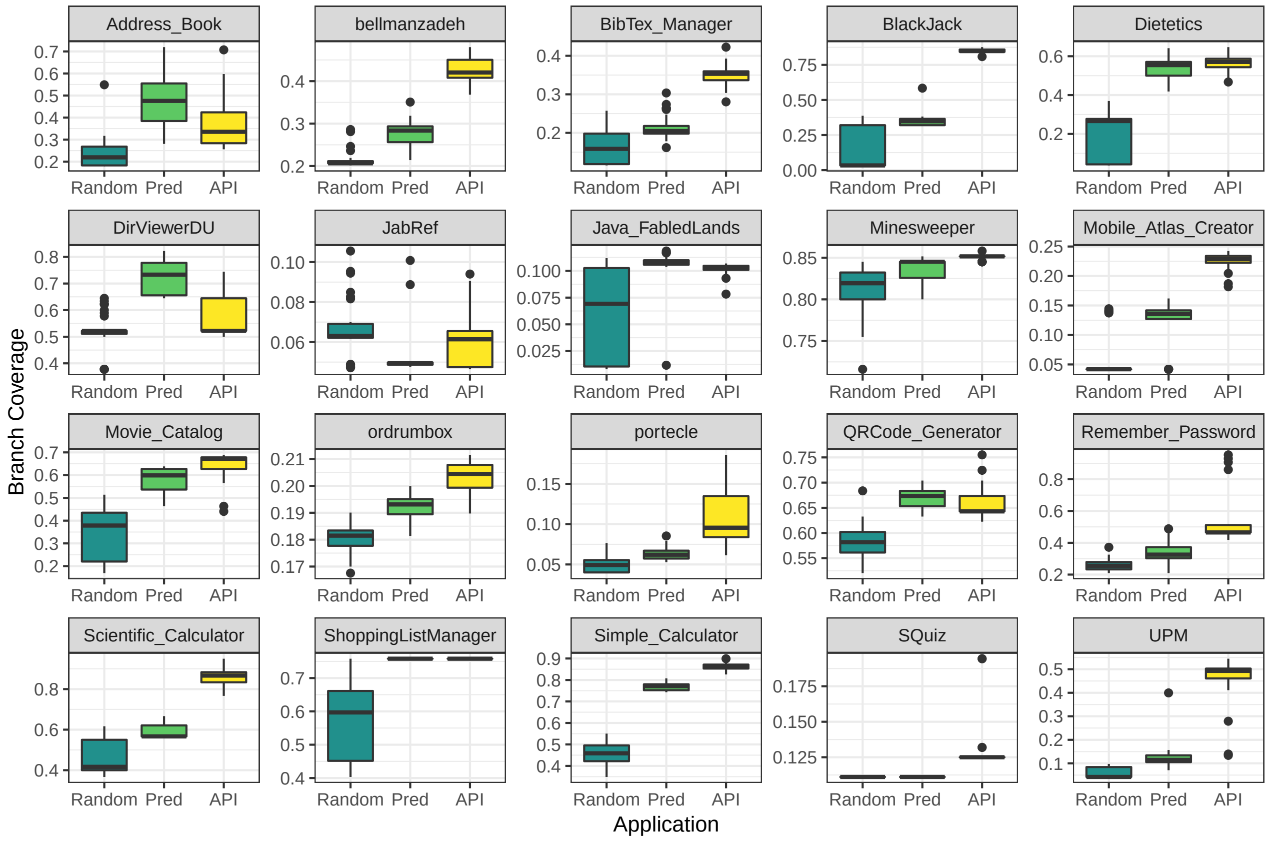


Figure 8: Branch Coverage achieved by a random clicker when clicking random coordinates, guided by predicted widgets positions and guided by the Swing API.

Previous work on applying context to GUI widgets involved searching for possible descriptive labels for each data widget. Becce et al. search above and to the left of data widgets for static widgets that can provide more information for testers about the type of data to input [3]. Coverage increase of 6% and 5% were found in the two applications tested when providing context about data widgets to the tool AutoBlackTest.

6 CONCLUSIONS

In conclusion, we found that a model trained on screen shots of synthetic GUIs can identify widgets in real GUIs. Applying this model during random GUI testing led to a significant coverage increase in 18 out of 20 applications in our evaluation. A particular advantage of this approach is that the prediction model is independent of a specific GUI library or operating system. Consequently, our prediction model can immediately support any GUI testing efforts.

Comparison to a "golden" model with perfect information of GUI widgets shows that there is potential for future improvement:

- Firstly, we need to find a better method of classifying GUI widgets. A tab that changes the contents of all or part of a GUI's screen has the same function as a button, so they could be grouped together.
- We currently use YOLOv2 and this predicts classes exclusively: if a button is predicted, there is no chance that a tab could also be predicted. Newer methods of object detection (e.g. YOLOv3 [15]) focus on multiple classification, where a widget could be classified as a button and as a tab. This could improve the classification rate of widgets that inherit attributes and style.
- Whilst labor intensive, further improvements to the widget prediction model could be made by training a model on a labeled dataset of *real* GUIs and augmenting this dataset with generated GUIs. The efficiency of the model is dependent on the quality of training data.
- Furthermore, in this paper we focused on a single system with various themes. However, it may be beneficial to train the model using themes from many operating systems and environments to improve performance when identify widgets across different platforms.

Besides improvements on the prediction model itself, there is potential to make better use of the widget information during test generation. For example, if there are a limited number of boxes to interact with, it may be possible to increase the efficiency of the tester by weighting widgets differently depending on whether they have previously been interacted with (e.g., [17]). This could be further enhanced using a technique like Q-learning (cf. AutoBlackTest [10]).

REFERENCES

[1] M. Bajammal and A. Mesbah. 2018. Web Canvas Testing Through Visual Inference. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST)*. 193–203. https://doi.org/10.1109/ICST.2018.00028

[2] S. Bauersfeld and T. E. J. Vos. 2012. GUITest: A Java Library for Fully Automated GUI Robustness Testing. In *2012 Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*. 330–333. https://doi.org/10.1145/2351676.2351739

[3] Giovanni Becce, Leonardo Mariani, Oliviero Riganelli, and Mauro Santoro. 2012. Extracting Widget Descriptions from GUIs. In *Fundamental Approaches to Software Engineering*, Juan de Lara and Andrea Zisman (Eds.). Springer Berlin Heidelberg, Berlin, Heidelberg, 347–361.

[4] Hatim Chahim, Mehmet Duran, and Tanja EJ Vos. 2018. Challenging TESTAR in an industrial setting: the rail sector. (2018).

[5] Developers, Android. 2015. Monkeyrunner. (2015).

[6] Justin E Forrester and Barton P Miller. 2000. An Empirical Study of the Robustness of Windows NT Applications Using Random Testing. In *Proceedings of the 4th USENIX Windows System Symposium*. Seattle, 59–68.

[7] Ross B. Girshick, Jeff Donahue, Trevor Darrell, and Jitendra Malik. 2013. Rich feature hierarchies for accurate object detection and semantic segmentation. *CoRR* abs/1311.2524 (2013). arXiv:1311.2524 http://arxiv.org/abs/1311.2524

[8] R. Lo, R. Webby, and R. Jeffery. 1996. Sizing and estimating the coding and unit testing effort for GUI systems. In *Proceedings of the 3rd International Software Metrics Symposium*. 166–173. https://doi.org/10.1109/METRIC.1996.492453

[9] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. ACM, 94–105.

[10] Leonardo Mariani, Oliviero Riganelli, and Mauro Santoro. 2011. The AutoBlackTest Tool: Automating System Testing of GUI-based Applications. *ECLIPSE IT 2011* (2011), 78.

[11] A. Memon, I. Banerjee, and A. Nagarajan. 2003. GUI ripping: reverse engineering of graphical user interfaces for testing. In *10th Working Conference on Reverse Engineering, 2003. WCRE 2003. Proceedings*. 260–269. https://doi.org/10.1109/WCRE.2003.1287256

[12] Bao Nguyen, Bryan Robbins, Ishan Banerjee, and Atif Memon. 2014. GUITAR: An innovative tool for automated testing of GUI-driven software. *Automated Software Engineering* 21 (03 2014). https://doi.org/10.1007/s10515-013-0128-9

[13] Joseph Redmon, Santosh Kumar Divvala, Ross B. Girshick, and Ali Farhadi. 2015. You Only Look Once: Unified, Real-Time Object Detection. *CoRR* abs/1506.02640 (2015). arXiv:1506.02640 http://arxiv.org/abs/1506.02640

[14] Joseph Redmon and Ali Farhadi. 2016. YOLO9000: Better, Faster, Stronger. *CoRR* abs/1612.08242 (2016). arXiv:1612.08242 http://arxiv.org/abs/1612.08242

[15] Joseph Redmon and Ali Farhadi. 2018. Yolov3: An incremental improvement. *arXiv preprint arXiv:1804.02767* (2018).

[16] O. Russakovsky, L. Li, and L. Fei-Fei. 2015. Best of both worlds: Human-machine collaboration for object annotation. In *2015 IEEE Conference on Computer Vision and Pattern Recognition (CVPR)*. 2121–2131. https://doi.org/10.1109/CVPR.2015.7298824

[17] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. ACM, 245–256.

[18] Tom Yeh, Tsung-Hsiang Chang, and Robert C. Miller. 2009. Sikuli: Using GUI Screenshots for Search and Automation. In *Proceedings of the 22Nd Annual ACM Symposium on User Interface Software and Technology (UIST '09)*. ACM, New York, NY, USA, 183–192. https://doi.org/10.1145/1622176.1622213