ActivityGen: Extracting Enabled Activities from Screenshots

Harry H. Beyel*, **Sovin Manuel and Wil M. P. van der Aalst**

RWTH Aachen University, Aachen, Germany

Abstract. Many tasks in organizations are performed in a desktop environment. It is possible to record users' interactions in a desktop environment by taking screenshots when an action happens. The result is an interaction log. By considering the associated images of a record, it is possible to detect which activity was performed and which activities were enabled. This information can be extracted, resulting in a translucent event log. Such a translucent event log is valuable and can be used as input for dedicated process-mining techniques. The results can be used to analyze human-computer interactions or create bots for robotic process automation. However, current techniques for extracting information on enabled activities rely on template matching, which is rigid and sensitive to variations. To solve this issue, we present our modular framework, ActivityGen. ActivityGen detects and labels graphical user interface elements by also considering additional information. ActivityGen uses more advanced techniques to overcome the limitations of previous approaches and can extract information without a user's input. Furthermore, it can be adjusted to a user's needs. It detects graphical user interface elements more accurately than state-of-the-art techniques and labels them faster, more robust, and more domain-oriented than state-of-the-art techniques.

1 Introduction

An enormous amount of tasks in organizations are done in desktop environments. The analysis of such tasks has great potential. For example, interactions between humans and computers can be analyzed. In this analysis, potential issues concerning interfaces can be identified. Also, it is possible to create bots for *Robotic Process Automation* (RPA) [50]. Such bots are used to automate repetitive tasks in a desktop environment. To create bots faster and also more robust bots, data-driven approaches are needed. Such techniques are developed in the field of *Robotic Process Mining* (RPM) [31], a subfield of *task mining* [20], which is related to *process mining* [52]. Process-mining techniques can be grouped into three categories: process discovery [3], conformance checking [11], and process enhancement [16]. Input data for process-mining techniques are provided in the form of an *event log*. Such an event log consists of *events*, each having a *case identifier*, an *activity*, a *timestamp*, and perhaps additional data attributes. Thus, to apply process mining in a desktop environment, digitally capturing the interactions between humans and computers is essential. There are some tools that capture these interactions, but they have limitations. WinParrot and JitBit Macro Recorder are unsuitable due to recording too low-level interactions or not considering enough context. Other tools, like the Action Logger [30], can only be applied to a limited number of applications. An application-independent approach can be achieved by using *interaction logs*. To create an interaction log, a user's screen is recorded, capturing an application's Graphical User Interface (GUI) such that each entry of a record is associated with an image showing the state of a screen. An example is provided in Table 1. By converting an interaction log into an event log, a *translucent event log* can be created. A translucent event log is based on an event log and additionally captures information on activities that can be executed — next to the executed activity. An example is shown in Table 2. Dedicated process-mining techniques can be applied to translucent event logs [7, 53].

Table 1. Example interaction log.

Interaction	Case	Snapshot	x	y	Timestamp
i_1	100	IMG_01.png	100	150	13:37
i_2	100	IMG_02.png	200	900	13:45
i_3	200	IMG_23.png	675	320	16:20

Table 2. Example translucent event log.

Event	Case	Activity	Timestamp	Enabled Activities
e_1	100	a	13:37	{a, e}
e_2	100	b	13:45	{b}
e_3	200	c	16:20	{c}

The current technique to transform an interaction log into a translucent event log is presented in [6]. The authors use labeled templates and template matching to extract the information. If a template is detected in a screenshot, the label is added to the set of enabled activities of an entry. This approach has multiple shortcomings. First, template matching is rigid and sensitive to changes. Such changes are likely to happen, for example, due to different screen ratios between users and resizing windows. Second, a user must provide all templates with corresponding labels that should be detected. Creating these is time-consuming and prone to errors.

Extracting information on executable activities in a screenshot can be divided into two parts: detecting GUI elements and labeling them. Current works of detecting GUI elements, such as [56, 59], do not consider user input (e.g., providing a template of a GUI element to detect) or need a lot of training data. In our case, such data are inaccessible since we only have a limited number of runs of a task. Current works of labeling GUI elements, such as [4, 27], have inaccessible data, cannot be adjusted or need much time for labeling,

* Corresponding Author. Email: beyel@pads.rwth-aachen.de

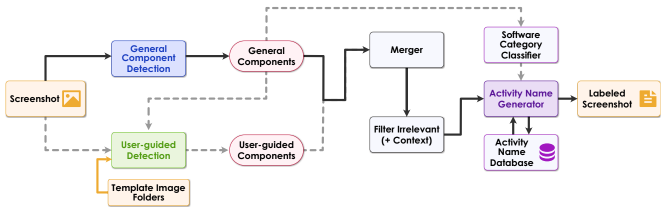


Figure 1. Overview on ActivityGen.

which leads to a time-consuming transformation from an interaction log into a translucent event log. To overcome the limitations of existing works, we propose a framework that detects and labels GUI elements, thus enabling rich extraction of enabled activities. An overview is provided in Figure 1. Our framework can incorporate user input but does not rely on it. Furthermore, by design, the underlying methods in our framework can be changed so that various methods can be used, making it robust for future techniques. The detection and labeling mechanisms do not depend on each other. As denoted in our evaluation, the quality of the detection mechanisms is better than state-of-the-art techniques (by 5% concerning the F1 score). The labeling is less hardware intensive and faster than state-of-the-art techniques, and, as observed in our experiments, it is more expressive and robust. Due to its design, it can also incorporate information beyond images.

2 Related Work

2.1 *Detecting GUI Components*

In [56], UI Element Detection (UIED) is introduced. The approach is applied to GUIs. UIED combines traditional computer vision techniques, like canny edge detection and flood-filling, with deep learning to detect GUI components. These components are assigned to a category using a deep-learning model. UIED uses a top-down approach for detecting shapes in a screenshot. Therefore, noise does not influence the detection. In contrast, we train another classifier on cleaner and more recent data. Furthermore, we allow for a user-guided detection of components, i.e., template images of GUI components are provided by a user and used to detect similar instances. In [43], mobile interfaces are generated given a drawing. The approach uses a pre-date deep-learning method, and basic geometric shapes are detected. These detected shapes are aggregated into more complex objects, i.e., the approach is a bottom-up technique. Besides shapes, text is also detected. Nevertheless, the approach only differentiates between text and non-text components. Additionally, some approaches solely rely on deep-learning [55, 57, 58, 59]. These methods rely on training deep-learning object-detection models like YOLO [46] or Faster Region-based Convolutional Neural Networks (R-CNN) [48]. In [57], YOLO is fine-tuned and used to detect drawn GUI elements on sketches. In [55, 59], GUI components are extracted using a trained Faster R-CNN. Applying heuristics to these components leads to inferring the hierarchical structure of screenshots or accessible metadata. However, detecting this structure may not be possible in our case, e.g., if the background is shaded. Overall, a drawback of these techniques is that they need an enormous amount of data to be reliable. Yet, in our scenario, the amount of instances available for training is limited. Thus, the mentioned approaches are less applicable.

2.2 *Generating Activity Names*

There is work focusing on generating captions for images. Such captions can be interpreted as activities that we generate in our work. In [33], WidgetCaption is presented. The work focuses on generating a natural language description for the functionality of a GUI component. They introduce a new dataset based on the RICO dataset [17]. Originally, the RICO dataset contained screenshots from multiple Android apps. The authors build on the stored screenshots and assign a label to each element in the images. Each label describes the functionality of an element. The newly created dataset is used to train an encoder-decoder model that considers the view hierarchy in Android apps and an image of a component to generate captions. No dedicated pre-training is used. In [27], Pix2Struct is introduced, iterating on WidgetCaption [33]. It takes a screenshot as input in which the GUI component is outlined using a bounding box and creates a caption. To ensure robustness, fine-tuning and pre-training are applied. For fine-tuning, the WidgetCaption dataset is used. Another work in this area is Spotlight [32]. Similar to Pix2Struct, it takes a screenshot and a region of interest as input and uses an encoder-decoder model. It is pre-trained on predicting text-based attributes and fine-tuned on the WidgetCaption dataset [33]. In contrast to Pix2Struct, other regions of interest are also extracted. Pix2Struct and Spotlight perform similarly. In [4], Lexi is presented. Lexi aims to learn representations of screenshots and their components. The authors propose a new dataset called UICaption, which was only created for training Lexi and scraped from how-to guide websites and instruction manuals. The dataset contains GUI screenshots paired with descriptions of functionality. The dataset is not publicly available. One of the pre-training tasks is masked language modeling, in which the model learns to predict parts of the caption in context to other given words and the image of the component. The different pre-training tasks and the new dataset allow the model to learn a representation of the functional properties of different GUI elements. In contrast to the formerly mentioned approaches, we directly use the information available from the GUI components to generate activity names. This implies that the detection system stays the same. As a result, a user-specific language model can be deployed to allow for domain-specific activity names, thus allowing for various application scenarios. Moreover, the bounding boxes of the GUI components are automatically determined using our approach, and thus, it is not necessary to provide them manually. Furthermore, other meta-data can be incorporated to create the translucent event log. Our approach also allows us to infer activities w.r.t. to an application's functionality.

2.3 *Extracting Information from Interaction Logs*

There are works focusing on extracting various information from interaction logs, especially in the RPM field [20, 31]. However, we only mention a few since none of the works extract activities from screenshots. In [26], extracting data from semi-structured documents using examples is presented. However, it only works on documents, particularly websites (HTML), spreadsheets, and other text. This information may not always be accessible; hence, our approach using screenshots is more general. Also, icons may be used in applications, making text inaccessible. In [39, 40], the authors detect components on a screenshot and enrich UI logs, similar to interaction logs, by extending them with information about the number of instances of a component class. Our approach differentiates by also detecting user-given components and inferring the activities that describe them, which the mentioned approach does not do. In [2], a tool is presented that converts user interface logs into RPA scripts but does not identify or label components. Only activity names such as "mouseClick" and "changeField" are created. Activities can be abstracted to a higher level, but there is no instruction or reference for performing this step in the work. In [35], a concept is represented that captures user interaction in a desktop environment to create a process model. While they mention image recognition, their prototype does not use GUI component identification and matching.

3 Preliminaries

3.1 Logs

First, we define translucent event logs. $\mathcal{U}_{case}$ is the universe of case identifiers, $\mathcal{U}_{act}$ is the universe of activity names, and $\mathcal{U}_{time}$ is the universe of timestamps.

Definition 1 (Translucent Event Log). *$\mathcal{U}_{ev}$ is the universe of events. $e \in \mathcal{U}_{ev}$ is an event, $\pi_{case}(e) \in \mathcal{U}_{case}$ is the case of e, $\pi_{time}(e) \in \mathcal{U}_{time}$ is the time of e, $\pi_{en}(e) \subseteq \mathcal{U}_{act}$ is the set of enabled activities of e, and $\pi_{act}(e) \in \pi_{en}(e)$ is the activity of e. A translucent event log L is a set of events $L \subseteq \mathcal{U}_{ev}$.*

Concerning e_2 in Table 2, $\pi_{case}(e_2) = 100$, $\pi_{act}(e_2) =$ b, $\pi_{time}(e_2) =$ 13:45, and $\pi_{en}(e_2) = \{$b$\}$. When recording user interactions, we have different information available. In the recorded interaction log, each entry refers to a case, a timestamp, a snapshot (an image capturing the screen), and x- and y-coordinates identifying the cursor's position when the interaction happened. Additional information, e.g., an application name, is also possible but is not required.

Definition 2 (Interaction Log). *$\mathcal{U}_{in}$ is the universe of interactions, $\mathcal{U}_{snap}$ is the universe of snapshots, and $\mathcal{U}_{app}$ is the universe of application names. $i \in \mathcal{U}_{in}$ is an interaction, $\pi_{case}(i) \in \mathcal{U}_{case}$ is the case of i, $\pi_{time}(i) \in \mathcal{U}_{time}$ is the time of i, $\pi_s(i) \in \mathcal{U}_s$ is the snapshot of i, $\pi_x(i) \in \mathbb{N}$ and $\pi_y(i) \in \mathbb{N}$ are the cursor's coordinate, and $\pi_{app}(i) \in \mathcal{U}_{app}$ is the application name of i. An interaction log I is a set of interactions $I \subseteq \mathcal{U}_{in}$.*

For i_1 in the log presented in Table 1, $\pi_{case}(i_1) = 100$, $\pi_s(i_1) =$ IMG_01.png, $\pi_x(i_1) = 100$, $\pi_y(i_1) = 150$, and $\pi_{time}(i_1) =$ 13:37.

3.2 Computer Vision

In our work, we use a wide range of computer-vision techniques, covering modern machine-learning and traditional methods. Concerning machine-learning methods, Convolutional Neural Networks (CNNs) [1, 22] are used. CNNs are neural networks with a grid-like topology and spatial dependencies and use convolutions. These properties make CNNs ideal for image-processing tasks. Residual Neural Networks (ResNets) [25] are improved CNNs such that the gradient does not become smaller during backpropagation. Some algorithms also use an encoder-decoder architecture [49]. This structure maps an input sequence to an output sequence with different lengths. Transformers [54] extend the idea of the encoder-decoder architecture. In contrast to CNNs and ResNets, self-attention layers capture long-range dependencies. All the aforementioned machine-learning methods rely on training data. In contrast, traditional methods only need a provided image. Canny edge detection [10] is applied to detect edges in images. The flood-fill algorithm [51] detects regions in an image with the same color. It starts by selecting a seed pixel and pushing it onto a stack. While the stack is not empty, the algorithm fills the current pixel with the desired color and pushes neighboring pixels with similar colors onto the stack. This continues until the stack is empty, resulting in a connected region with the same color. Morphological closing [24] smooths and closes gaps in binary images, where pixels are either black or white. This method makes shapes more solid and continuous. Template matching [9] detects if a given component is contained in an image, yet the match has to be exact. In contrast, feature matching [36, 37] is more robust. Sharp edges are needed to apply this technique.

3.3 Neural Language Models

The general goal of language models is to predict the next word or token given a sequence of words [5]. Current models are based on the transformer architecture [54]. Large-language models are a special kind of language model consisting of millions of parameters. Visual-language models [45] combine textual and visual information. Given an image-text pair, the model learns to encode this information and to decode text answers.

4 ActivityGen

The goal of our work is to transform an interaction log into a translucent event log. To do so, we need to infer activities from snapshots. In a desktop environment, various applications can be used, each with its unique GUI. This uniqueness leads to issues. In each application, different activities are hidden behind various shapes, such as icons. At the same time, not all icons refer to an activity. Some may be just used as an image with no interaction. Also, there are contextual dependencies of interaction elements. For example, a search field can be placed in different surroundings, resulting in a different meaning. Therefore, we propose a framework that can be adjusted to a user's needs. An overview of the framework is provided in Figure 1. As depicted, ActivityGen consists of multiple modules. In each module, components play a vital role. A component is smaller than a snapshot and reflects an area of interest.

Definition 3 (Component). *$\mathcal{U}_{comp}$ is the universe of components. If a component $c \in \mathcal{U}_{comp}$ appears in a snapshot $s \in \mathcal{U}_{snap}$, this is denoted as $c \sqsubseteq s$. $\pi_{x_1}(c), \pi_{x_2}(c) \in \mathbb{N}$ represent the horizontal boundaries of a component within a snapshot, $\pi_{y_1}(c), \pi_{y_2}(c) \in \mathbb{N}$ refer to the vertical boundaries. In addition, $\pi_{x_1}(c) < \pi_{x_2}(c)$ and $\pi_{y_1}(c) < \pi_{y_2}(c)$.*

Necessary for the detection are provided snapshots. If a user provides a template, detecting components in the snapshot is also user-guided. However, this is not required. The general component detection detects all components of interest independent of possible input. The collected information is merged, components get classified, and irrelevant components are filtered out. To generate context-relevant activities, we provide an optional software category classifier. Finally, an activity name is generated, and the component gets labeled. In the following, we spotlight the modules of ActivityGen. Thereby, we focus on the characteristics and the implementation of the framework.

4.1 General Component Detection

Given an image, this module detects text- and non-text components. For a snapshot $s \in \mathcal{U}_{snap}$, the general component detection can be described as follows:

$$gcd(s) = \{c \in \mathcal{U}_{comp} \mid c \sqsubseteq s\} \quad (1)$$

Our approach is based on UIED [56]. We first focus on the text-detection. Subsequently, we explain the detection of non-text components. Also, we explain the merging process of the detected components. Finally, we explain how non-text components are classified and provide a classification step for icons.

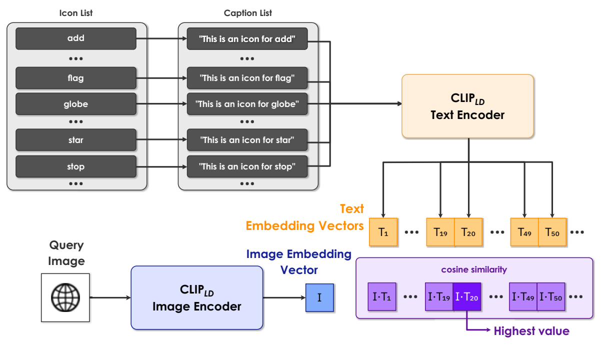
The image is processed, and if text is detected, it is extracted and saved for further analysis. To detect text, we use PaddleOCR [19].

To detect non-text components, we use a six-step algorithm. First, we convert the provided image into a binary image using a gradient map, which assigns new colors based on brightness. Noise is removed by using morphological closing [24]. Second, we detect connected regions by applying a flood-filling algorithm [51] on the binarized image. Third, the previously detected regions are treated as components, and we filter them based on their shape and size. Fourth, we merge intersecting components. Fifth, we investigate whether components consist of components. To do so, we check whether the inner side of components is blank, i.e., devoid of pixel values. If that is the case, a component is recognized as "Block"; else, it is recognized as "Compo". Last, if components are exceptionally large, we apply again a flood-filling algorithm [51] to identify potential overseen components.

After the detection of text and non-text components, we merge them based on their overlap. When non-text and text components intersect, the detected text has precedence in merging. Furthermore, detected text components are aggregated into lines and paragraphs based on proximity. In this step, we also consider the relation between components. If a component y is fully included in another component x, y is saved as a child of component x. This information is saved in intermediate outputs of ActivityGen.

The component classifier categorizes the detected non-text components of type "Compo". There are various non-textual GUI components, such as radio buttons. To categorize components, we use a deep-learning-based image classifier. In UIED [56], the RICO dataset is used [17]. In our work, we modified the ReDraw dataset [41], a labeled dataset of GUI components taken from Android screenshots and optimized for classifying components. The dataset contains various component classes. We focus on the following: "Button", "CheckBox", "EditText", "Image", "ImageButton" (which we refer to as icon), "RadioButton", and "Switch". We noticed that the examples of "ImageView" and "ImageButton" are similar, primarily consisting of icon images. Therefore, we removed the "ImageView" class and introduced an "Image" class instead. The "Image" class contains images from the validation set of the Coco validation set 2017 [34] and the YouTube Thumbnails dataset [42], enabling the detection of general website images. We call this dataset *ReDraw*$_{cls}$. We use a ResNet-50 [25] that is pre-trained on the ImageNet dataset [18] and fine-tuned using the dataset that we created. During the classification, the model outputs a probability for the classification. The class of a component is only changed if the model is certain, i.e., the prediction probability is above a pre-defined threshold that a user can define.

Many functions in a GUI are hidden behind icons. Some icons may be program-exclusive. Therefore, training a classification model on a predefined set of icons would lead to failures if unseen icons appear. To overcome this, we adopt an open vocabulary classification approach using Contrastive Language-Image Pre-Training (CLIP) [45]. An overview of the approach is shown in Figure 2. CLIP is trained on a dataset of 400 million image-text pairs to determine which image



Icon List
add
flag
globe
star
stop
Caption List
"This is an icon for add"
"This is an icon for flag"
"This is an icon for globe"
"This is an icon for star"
"This is an icon for stop"
CLIP$_{LD}$ Text Encoder
Text Embedding Vectors
T$_1$ T$_{19}$ T$_{20}$ T$_{49}$ T$_{50}$
Query Image
CLIP$_{LD}$ Image Encoder
Image Embedding Vector
I
cosine similarity
I·T$_1$ I·T$_{19}$ I·T$_{20}$ I·T$_{49}$ I·T$_{50}$
Highest value

Figure 2. Overview of icon classification approach.

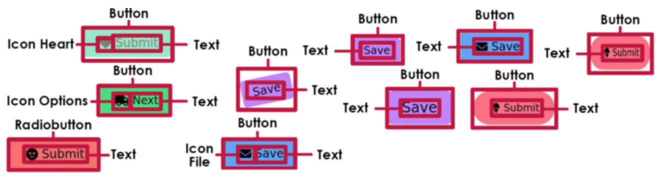


Figure 3. Results of general component detection.

corresponds to which text. As a result, the model learns to compare images and texts. Thus, it learns to differentiate between concepts of an icon. Ultimately, it learns to generalize combinations of multiple concepts. We use OpenClip [14], trained on the DataComp-1B dataset [21] by LAION. We name this model Clip$_{LD}$. In prototype testing, this version demonstrated the best results for icon classification. Results of the general component detection are displayed in Figure 3.

4.2 User-Guided Detection

User-guided detection can be used but does not have to. Given a snapshot $s \in \mathcal{U}_{snap}$ and a set of components $C \subseteq \mathcal{U}_{comp}$, the user-guided detection can be described as follows:

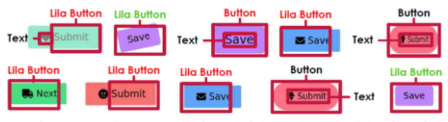
$$ugd(s, C) = \{c \in C \mid c \sqsubseteq s\} \quad (2)$$

The module consists of three independent parts, for which a user can decide which to use. These parts are template matching, feature matching, and Visual Similarity Matching (ViSM). All three parts detect if a template image is contained in a snapshot. Nevertheless, they use different approaches. In the following, we present the three parts and discuss them.

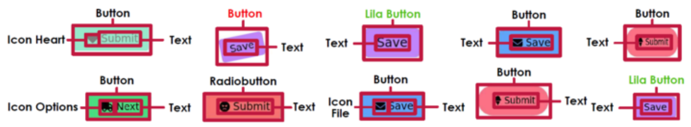
As described, template matching evaluates whether a provided template is part of an image. GUI components may be identical across different instances. Therefore, we included template matching in our framework. At the same time, template matching is rigid and sensitive to variations, as pointed out before.

As introduced, feature matching is a more robust technique than template matching. Nevertheless, using this technique in our approach leads to challenges. There may be multiple instances of a component contained in a snapshot. Nevertheless, only one would be detected. This issue can be addressed using a sliding window such that the template or detector is applied at different positions and scales across the image to detect multiple instances. However, this approach slows down the matching process. The same approach can also be applied in template matching to detect multiple instances, yet template matching is less computationally expensive.

(a) Template image: "Lila Button".



(b) Template matching with a threshold of 0.6.



(c) ViSM with a threshold of 0.99.

Figure 4. Results of user-guided detection. A correct matching is indicated in green, and an incorrect one in red.

The last part is ViSM, which has increased flexibility. Contrary to the former approaches, it requires the components detected by the general component detection. Visual templates are compared with the detected components. For example, given an image of a checked checkbox, similar checkboxes are found in components. To accomplish this, we utilize ResNet-50 models [25] and SimCLR [13]. SimCLR is a framework designed for self-supervised training of models to learn visual representations of images. These visual representations are captured as embedding vectors, which can be compared using similarity measures. In this context, visual representation refers to the model's ability to learn and encode visual cues from images, enabling subsequent comparisons. In [13], it is demonstrated that this approach, despite requiring a smaller amount of labeled data (only 1% of the labeled data in relation to the size of the unlabeled data), achieves comparable or superior performance compared to models trained using supervised methods. This approach is advantageous for GUI components, where numerous variations exist within the same component category, allowing us to compare them visually. For training, we extend the ReDraw$_{\text{cls}}$ dataset by adding synthetically created buttons. We created 6,000 new images of buttons with different colors, icons, texts, and shapes using Tailwind CSS. Besides its popularity, another reason is that Tailwind CSS offers a basic design system that provides low-level utility classes rather than pre-designed components, making generating different variants easier. The images are distributed into the different split sets according to the previous distribution ratio of ReDraw$_{\text{cls}}$. Thus, we allow for more differentiation between instances. We call the extended dataset *ReDraw$_{vism}$*. Results of the user-guided detection are displayed in Figure 4.

4.3 Software Category Classifier

Understanding the context of an application is crucial to interpreting actions and words in an application. User interfaces convey information about an application's functionalities and interactions through keywords. For example, in a to-do application, we commonly find words like "tasks" and "deadlines". By comparing this information to our existing knowledge of categories and their associated information, we determine which category an application belongs to. In our work, we extract text from components using general component detection. The extracted text is embedded into a vector space, allowing it to measure semantic similarity to other texts. As text embedder, we use all-MiniLM-L6-v2 [47]. To determine the category, we define a set of categories and associated keywords. The embedding of each keyword is compared to the embedding of the extracted text, allowing the assignment of a category. Our approach allows users to assign categories that are use-case-specific.

4.4 Filtering Components

We filter out components that belong to classes "Image", "Block", or "Compo". Furthermore, we exclude text components if they contain long text or have a parent component. We used more than three words as the threshold, but it can be adjusted by a user. In addition, we exclude text, image, or icon components if their parent component is either a button or of class "EditText". Besides, depending on their state, we assign checkbox components as inner text "check" or "uncheck". If a component has a preexisting activity name, a new name will not be generated. An optional step is adding surrounding text. It determines if a component has a direct ancestor of class "Block" or "Compo". If it does, the text contained in this ancestor component is aggregated and associated with the component. Nearby text is also added to the "EditText" and "Checkbox" components if they do not have text inside. This follows the idea that the information about what the component does is indicated by nearby text.

4.5 Activity Name Generator

Given merged and filtered components and, if available, their context, this module infers activities associated with the components. The module consists of three independent approaches. A template-based natural language approach, a basic language model, and an extended language model. For simplicity, we refer for any $c \in \mathcal{U}_{comp}$ its activity by $\pi_{act}(c) \in \mathcal{U}_{act}$.

The template-based natural language generation is the most stable option for generating activity names since it uses textual templates. For example, "Click on the Button ⟨inner text⟩", where ⟨inner text⟩ is the label that a button contains. It is inspired by [29], where captions are generated for the entire screen by using natural language templates. Currently, this module supports components of the classes "Checkbox", "Button", and "EditText".

The basic language model (LM$_{\text{basic}}$) has the best balance between reliability and flexibility in terms of generational quality. To overcome the lack of available training data, we employ the concept of knowledge distillation [23]. It describes the process of training a so-called student model with the supervision of a teacher model. As teacher model, we use gpt-3.5-turbo from OpenAI [44]. An entry in the dataset consists of the following information: the application's name, the component class, the inner text, and the activity name that the model should predict. The created dataset contains over 10,000 training examples and about 1,200 entries for the test and validation set. Over 854 different applications are covered. As student model, we utilize Pythia-410m-deduped [8], a transformer-based language model. We chose this model because of interpretability research and different sizes, allowing future researchers to compare the differences in model sizes. We trained the model on causal language modeling, i.e., next token prediction. We use the best model evaluated by using the perplexity metric on the validation set.

The extended language model (LM$_{\text{extended}}$) shows how further information can be used to improve the generation context. The extended dataset for this model is similar to the former dataset, but we add two new attributes to each entry: the category of an application and the surrounding text. As model, Flan-t5-base is used [15]. The advantage of this model is its prefix methodology for fine-tuning, which involves fine-tuning the model on a particular task by incorporating a prefix to the input text during training, which is more suitable for generating different prefixes for different use cases. This allows the creation of different extensions for future training. The model is trained on question-answering using the component information as context. We use the best model evaluated using the ROUGE score on the validation set.

5 Creating Translucent Event Logs

We use ActivityGen to transform an interaction log into a translucent event log. For each interaction, we detect the component that was interacted with and use its activity as an event's activity. Furthermore, we use the other detected components' activities in the snapshot of an interaction to assign enabled activities to an event. The case and timestamp remain the same. For a snapshot $s \in \mathcal{U}_{snap}$, $\pi_c(s) \subseteq \mathcal{U}_{comp}$ denote the set of detected components in a snapshot.

Definition 4 (Transformation)**.** *Let $I \subseteq \mathcal{U}_{in}$ be an interaction log and $L \subseteq \mathcal{U}_{ev}$ be a translucent event log. L is the transformed interaction log I if there is a bijective function $I \to L$ such that for any $i \in I$ there is an $e \in L$ with $\pi_{time}(e) = \pi_{time}(i)$ and $\pi_{case}(e) = \pi_{case}(i)$. Moreover, $\pi_{act}(e) = \pi_{act}(c)$ where $c \in \pi_c(\pi_s(i))$ with $\pi_{x_1}(c) \leq \pi_x(i) \leq \pi_{x_2}(c)$ and $\pi_{y_1}(c) \leq \pi_y(i) \leq \pi_{y_2}(c)$ such that no other component is intersecting. Additionally, $\pi_{en}(e) = \{\pi_{act}(c) \mid c \in \pi_c(\pi_s(i))\}$.*

6 Evaluation

To allow for local execution and to enable broad use of our techniques, we implemented our work and evaluated our approach using the following settings: We utilized an Intel i5-13600K @ 3.49GHz as CPU, an Nvidia RTX 4080, 16GB VRAM, Driver version: 532.03, Cuda 12.1 as GPU, 32GB RAM, WSL2 using Ubuntu 22.04 as OS. Concerning software, we use opencv-python 4.6.0.66, paddleocr 2.7.0.2, and paddlepaddle 2.5.1. In the following listed settings, we only point out settings that are not the same as for the original work, respectively, non-standard settings. The LM_{basic} is trained with 10 epochs with early stopping, a learning rate of 1e-5, and a batch size of 32. The weight decay is set to 1e-5, and AdamW optimizer and a linear scheduler are used. The best model is loaded. Concerning the component classifier, 15 epochs, a learning rate of 5e-4, an input size of 224, and a batch size of 32 are used. Also, a step size of 7 and a gamma of 0.1 are utilized. In addition, a momentum of 0.9 for the SGD optimizer is used. Again, the best model is used at the end. Concerning the model for ViSM, 50 epochs with early stopping, a learning rate of 5e-4, an input size of 128, and a batch size of 256 are utilized. Also, the number of hidden dimensions is 128, the temperature is 0.07, and the weight decay is set to 1e-4. The AdamW optimizer and the cosine annealing scheduler are used. Datasets in the domain of interfaces that fit our requirements are rare or not publicly accessible, e.g., UICaption [4]. Thus, we are limited to a small selection of datasets for training and evaluating our approaches. To compare the labeling methods, we use the Pix2Struct large model fine-tuned on the WidgetCaption dataset. We refer to that model as $\text{Pix2Struct}_{\text{W}}$. Spotlight [32] and Lexi [4] are not publicly available. Our code and data are available [38].

6.1 Quantitative Evaluation

6.1.1 Detecting GUI Elements

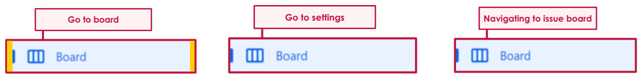
A quantitative analysis is conducted to assess the general detection's effectiveness and the bounding boxes' accuracy. Since our approach is based on UIED [56], the state-of-the-art technique, we anticipate that ActivityGen will yield a similar performance. This evaluation employs the test set from the RICO dataset [17]. The images are resized to a height of 800 pixels, consistent with the dimensions utilized in the original papers [12, 56]. Despite inherent imperfections and noise within the RICO dataset, it offers a valuable overall direction for assessment. To ensure a fair comparison, identical parameters are employed for both ActivityGen and UIED. The comparison focuses on the detection quality rather than individual classes, mirroring the approach taken in the UIED evaluation. The evaluation is based on Intersection over Union (IoU) > 0.9. IoU quantifies the degree of overlap between a predicted bounding box and the actual ground truth bounding box. This metric yields values from 0 to 1, where higher values indicate better alignment. A predicted bounding box is considered a true positive if its IoU with the corresponding ground truth bounding box surpasses the threshold of 0.9. Comparing the metrics shown in Table 3, we denote that the originally reported performance of UIED was not successfully reproducible. The initial papers documented F1 scores ranging between 0.5 and 0.6 for both scenarios, attributed to a higher level of precision. We experimented with various parameter configurations, but the reported results were not achieved. However, it is important to acknowledge that the reported outcomes in the original UIED papers were documented in 2020, predating newer additions to the project, which we use as the base. For instance, the transition from EAST to PaddleOCR for text detection represents one of the notable modifications. Despite these transformations, it is worth noting that even with the changes implemented, the performance of both ActivityGen and UIED remains comparatively similar. Nonetheless, ActivityGen's F1 score is roughly 5% better without text. The results are on the lower side, though this can be attributed to the noisiness of RICO and the high IoU threshold. The lower values for all components with text compared to those without text can be explained by PaddleOCR and the definition of bounding boxes in the Android hierarchy.

Table 3. Results on the test set of RICO [17] using IoU > 0.9.

		Precision	Recall	F1
UIED	all	0.239	0.564	0.337
	no text	0.332	0.515	0.404
ActivityGen	all	0.233	0.567	0.332
	no text	0.390	0.465	0.426

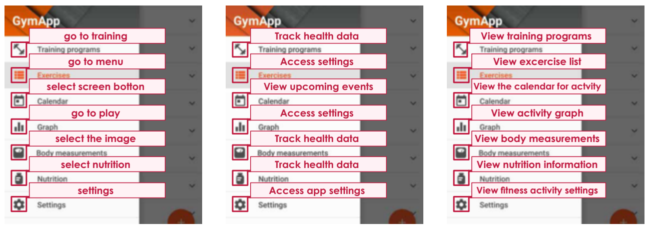
6.1.2 Inference Time

To assess the inference time required by ActivityGen, we again employ the test set of the RICO dataset [17]. The evaluation focuses on measuring the duration of the entire generation pipeline for the initial 100 images within the test set. An image may contain more than one component. The images undergo a resizing process, bringing them to 800 pixels in height. We utilize the LM_{basic} and save previously generated activity names that convey identical information. All functionalities are activated throughout this assessment except for the user-guided detection feature. The total time taken for the process amounts to 162.686 seconds, i.e., roughly 1.6 seconds are needed on average for each image. For 75% of images, less than two seconds are needed. It can be noted that the more components are contained in an image, the more time is required. This inference speed stands in contrast to alternative methods. For $\text{Pix2Struct}_{\text{W}}$, generating captions for all components can extend beyond 20 seconds for



(a) Correct caption by Pix2Struct$_W$. (b) Wrong caption by Pix2Struct$_W$. (c) Generation by LM$_{basic}$.

Figure 5. Captions generated by Pix2Struct$_W$ [27] and ActivityGen. Change in the bounding box is highlighted in yellow.



(a) Pix2Struct$_W$. (b) LM$_{basic}$. (c) LM$_{extended}$.

Figure 6. Captions generated by Pix2Struct$_W$ and ActivityGen.

a picture (5 times more than ActivityGen's maximum time for the whole pipeline). ActivityGen's faster performance can be attributed to its lower hardware demands, which are reflected in the VRAM usage. While the whole system uses 9979 Megabytes when using Pix2Struct$_W$, the whole system needs, when running ActivityGen, at most, 7794 Megabytes; 7000 Megabytes is the median usage.

6.2 Qualitative Evaluation

6.2.1 Detecting

To test out the robustness in detecting design variations, we use our dataset ReDraw$_{vism}$. This dataset allows for assessing the system's ability to identify variations. The results concerning the general component detection are depicted in Figure 3. As we observe, general detection is robust to variations of different components. Most buttons are correctly classified independently of color, rotation, shape, and size. However, there are tendencies to wrong classifications due to similarity to other components. For instance, a button with an angry face is interpreted as a radio button due to the roundness of the icon with nearby text. Also, the text and buttons' icons tend to merge due to PaddleOCR. Next, we investigate the detection capabilities of user-guided detection. We focus on template matching and ViSM. As a template, we use the button depicted in Figure 4(a). The results concerning template matching are depicted in Figure 4(b), and the results concerning ViSM are shown in Figure 4(c). Template matching with a threshold of 0.9 yields no results. With a threshold of 0.6, some buttons are detected, but the detection is noisy, and the bounding boxes are incorrect, which shows the shortcomings in template matching. In contrast, the detection using ViSM is better. Only relevant buttons are detected, regardless of their size. However, distortions like rotation cannot be detected due to the model's training data.

6.2.2 Labeling

We first evaluate the stability. The result is shown in Figure 5. While Pix2Struct$_W$ is prone to changes in the bounding box, our approach does not suffer from such changes. As we detect the bounding boxes of detected components, we automatically extract the component's information and pass it to the language model, making the process robust against the positions of elements.

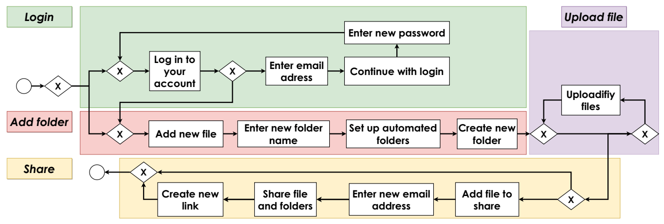


Figure 7. Process model generated by applying the Inductive Miner [28] on the generated translucent event log.

Next, we evaluate the differences between the generated labels by different methods. We use one image of the RICO dataset [17], and the results are depicted in Figure 6. The generation time for Pix2Struct$_W$ was 8.9 seconds. In contrast, both methods of ActivityGen were significantly faster, each requiring only 2.2 seconds for the same task. As the real application name is unknown, we use the placeholder "Mobile App". LM$_{extended}$ generates concise and actionable labels. At the same time, the model is biased since settings are associated with fitness. LM$_{basic}$ occasionally generates reasonable activity names, although more broad. It recognizes the subject of the application. The results of Pix2Struct$_W$ are less topic-sensitive than the others.

6.2.3 Case Study

To demonstrate ActivityGen's capabilities for process mining, we conduct an end-to-end case study using Dropbox. The cases are as follows. First case: The user logs in by entering email and password. After that, the user gets to the main dashboard. There, the user creates a folder. A folder dialog appears to set up the folder (name and other options). After that, a file is uploaded to the folder. Lastly, a link is generated to share the folders with others. A dialog appears where the user can add recipients, and then the user copies the link for sharing. Second case: At first, a folder is created. After that, multiple files are uploaded. Using the Celonis Task Mining tool, we create an interaction log and then apply ActivityGen. We use ActivityGen with LM$_{basic}$. For activity name generation, we use the application name "Dropbox" and the current view of the application, e.g., "Dropbox - Login". We only consider activities as enabled activities if they are also executed at some point. The process model that can be discovered on the created event log is depicted in Figure 7. Generally, the activity names are captured well by ActivityGen and are reasonable. They describe the interaction well and lead to a correct model for the workflow. The activity name generation shows summarizing abilities for longer (checkbox) labels: The option "Set up this folder to automatically handle tasks like organizing your content and converting files. You will set up your automation after we create the folder." is automatically interpreted as "Set up automated folders". The activity names are also consistent across different variations. As such, duplicates from case two are recognized as a possible loop, indicating that uploading can be done more than once and to login is not necessary every time. "Log in to your account" is seen twice, as the first and last components in the login procedure have the same inner text. Nevertheless, there are some minor errors. "Uploadify files" stands for "Upload files" but is mistaken due to errors in detection, where a nearby arrow icon is interpreted as text "1". This error is forwarded to the activity generation, which leads to an inconsistent output, as the training data do not contain such noise, and "Enter new password" is erroneous as no *new* password is entered in interactions. By combining ActivityGen with process mining, we are able to get insights into the interactions between humans and computers. At the same time, the output can be used to identify potential for automation by using dedicated RPM or process mining techniques.

7 Conclusion

We presented the modular framework ActivityGen. As presented, the framework can perform its task independently from user input and can be adjusted to a user's needs. Since a user can still provide input, we ensure that use-case-relevant elements can be detected. The framework performs at least state-of-the-art results in detecting GUI elements, labels them faster than state-of-the-art techniques, and generates more specific labels. Also, additional information, e.g., the application name, can be used. Due to the small inference time, it is beneficial for transforming an interaction log into a translucent event log. Besides the transformation, potential applications include documentation purposes and creating manuals. Also, visually impaired people may use it since relevant interaction elements are detected, and well-representative captions are generated quickly, which can be read to the user. Parts of our method rely on training data. Many standard interfaces use the same assets and design principles, like the law of proximity. Hence, a potential lack of data might not be an issue. For custom interfaces, such as legacy software with little available data, user-guided detection offers valuable assistance.

However, the classification could be better, as shown by the misclassification based on misinterpreting an emoticon. Also, activity names may not always fit. There are also points for future work. First, our proposed framework does not focus on users' privacy. Even though the underlying interaction log is anonymized, snapshots can be used to conclude a user's identity. In addition, a preprocessing step, including redacting parts of screenshots, seems beneficial to prevent the sharing of sensitive information, e.g., email addresses. Second, users may perform their tasks by solely relying on the keyboard. Since we capture the cursor's position, some techniques, for example, the assignment of activities, may not work in such a case. Third, it is possible that users have to scroll before their interaction. However, we only focus on the screen when the interaction happens, not what is outside the displayed content. Therefore, potential activities are not captured. Fourth, since the generation of translucent event logs is improved, more dedicated process-mining techniques can be developed. This can lead to a better understanding of human-computer interactions.

Acknowledgements

We thank the Alexander von Humboldt (AvH) Stiftung for supporting our research.

References

[1] C. C. Aggarwal. *Neural Networks and Deep Learning - A Textbook*. Springer, 2018. ISBN 978-3-319-94462-3. doi: 10.1007/978-3-319-94463-0.

[2] S. Agostinelli, M. Lupia, A. Marrella, and M. Mecella. Automated generation of executable RPA scripts from user interface logs. In *BPM - Blockchain and RPA Forum*, pages 116–131. Springer, 2020. doi: 10.1007/978-3-030-58779-6_8.

[3] A. Augusto, R. Conforti, M. Dumas, M. L. Rosa, F. M. Maggi, A. Marrella, M. Mecella, and A. Soo. Automated discovery of process models from event logs: Review and benchmark. *IEEE Trans. Knowl. Data Eng.*, 31(4):686–705, 2019. doi: 10.1109/TKDE.2018.2841877.

[4] P. Banerjee, S. Mahajan, K. Arora, C. Baral, and O. Riva. Lexi: Self-supervised learning of the UI language. In *EMNLP*, pages 6992–7007. Association for Computational Linguistics, 2022.

[5] Y. Bengio, R. Ducharme, P. Vincent, and C. Janvin. A neural probabilistic language model. *J. Mach. Learn. Res.*, 3:1137–1155, 2003.

[6] H. H. Beyel and W. M. P. van der Aalst. Creating translucent event logs to improve process discovery. In *ICPM Workshops*, pages 435–447. Springer, 2022. doi: 10.1007/978-3-031-27815-0_32.

[7] H. H. Beyel and W. M. P. van der Aalst. Translucent precision: Exploiting enabling information to evaluate the quality of process models. In *RCIS*, pages 29–37. Springer, 2024. doi: 10.1007/978-3-031-59468-7_4.

[8] S. Biderman, H. Schoelkopf, Q. G. Anthony, H. Bradley, K. O'Brien, E. Hallahan, M. A. Khan, S. Purohit, U. S. Prashanth, E. Raff, A. Skowron, L. Sutawika, and O. van der Wal. Pythia: A suite for analyzing large language models across training and scaling. In *ICML*, pages 2397–2430. PMLR, 2023.

[9] R. Brunelli. *Template Matching Techniques in Computer Vision: Theory and Practice*. John Wiley & Sons, 2009. ISBN 978-0-470-51706-2. doi: 10.1002/9780470744055.

[10] J. F. Canny. A computational approach to edge detection. *IEEE Trans. Pattern Anal. Mach. Intell.*, 8(6):679–698, 1986. doi: 10.1109/TPAMI.1986.4767851.

[11] J. Carmona, B. F. van Dongen, A. Solti, and M. Weidlich. *Conformance Checking - Relating Processes and Models*. Springer, 2018. ISBN 978-3-319-99413-0. doi: 10.1007/978-3-319-99414-7.

[12] J. Chen, M. Xie, Z. Xing, C. Chen, X. Xu, L. Zhu, and G. Li. Object detection for graphical user interface: old fashioned or deep learning or a combination? In *ESEC/FSE*, pages 1202–1214. ACM, 2020. doi: 10.1145/3368089.3409691.

[13] T. Chen, S. Kornblith, M. Norouzi, and G. E. Hinton. A simple framework for contrastive learning of visual representations. In *ICML*, pages 1597–1607. PMLR, 2020.

[14] M. Cherti, R. Beaumont, R. Wightman, M. Wortsman, G. Ilharco, C. Gordon, C. Schuhmann, L. Schmidt, and J. Jitsev. Reproducible scaling laws for contrastive language-image learning. In *CVPR*, pages 2818–2829. IEEE, 2023. doi: 10.1109/CVPR52729.2023.00276.

[15] H. W. Chung, L. Hou, S. Longpre, B. Zoph, Y. Tay, W. Fedus, Y. Li, X. Wang, M. Dehghani, S. Brahma, A. Webson, S. S. Gu, Z. Dai, M. Suzgun, X. Chen, A. Chowdhery, A. Castro-Ros, M. Pellat, K. Robinson, D. Valter, S. Narang, G. Mishra, A. Yu, V. Zhao, Y. Huang, A. Dai, H. Yu, S. Petrov, E. H. Chi, J. Dean, J. Devlin, A. Roberts, D. Zhou, Q. V. Le, and J. Wei. Scaling instruction-finetuned language models. *Journal of Machine Learning Research*, 25(70):1–53, 2024.

[16] M. de Leoni. Foundations of process enhancement. In *Process Mining Handbook*, pages 243–273. Springer, 2022. doi: 10.1007/978-3-031-08848-3_8.

[17] B. Deka, Z. Huang, C. Franzen, J. Hibschman, D. Afergan, Y. Li, J. Nichols, and R. Kumar. Rico: A mobile app dataset for building data-driven design applications. In *UIST*, pages 845–854. ACM, 2017. doi: 10.1145/3126594.3126651.

[18] J. Deng, W. Dong, R. Socher, L. Li, K. Li, and L. Fei-Fei. Imagenet: A large-scale hierarchical image database. In *CVPR*, pages 248–255. IEEE Computer Society, 2009. doi: 10.1109/CVPR.2009.5206848.

[19] Y. Du, C. Li, R. Guo, X. Yin, W. Liu, J. Zhou, Y. Bai, Z. Yu, Y. Yang, Q. Dang, and H. Wang. PP-OCR: A practical ultra lightweight OCR system. Preprint arXiv:abs/2009.09941, 2020.

[20] M. Dumas, M. L. Rosa, V. Leno, A. Polyvyanyy, and F. M. Maggi. Robotic process mining. In *Process Mining Handbook*, pages 468–491. Springer, 2022. doi: 10.1007/978-3-031-08848-3_16.

[21] S. Y. Gadre, G. Ilharco, A. Fang, J. Hayase, G. Smyrnis, T. Nguyen, R. Marten, M. Wortsman, D. Ghosh, J. Zhang, E. Orgad, R. Entezari, G. Daras, S. M. Pratt, V. Ramanujan, Y. Bitton, K. Marathe, S. Mussmann, R. Vencu, M. Cherti, R. Krishna, P. W. Koh, O. Saukh, A. J. Ratner, S. Song, H. Hajishirzi, A. Farhadi, R. Beaumont, S. Oh, A. Dimakis, J. Jitsev, Y. Carmon, V. Shankar, and L. Schmidt. Datacomp: In search of the next generation of multimodal datasets. In *NeurIPS*, 2023.

[22] I. J. Goodfellow, Y. Bengio, and A. C. Courville. *Deep Learning*. MIT Press, 2016. ISBN 978-0-262-03561-3.

[23] Y. Gu, L. Dong, F. Wei, and M. Huang. Minillm: Knowledge distillation of large language models. In *ICLR*. OpenReview.net, 2024.

[24] R. M. Haralick, S. R. Sternberg, and X. Zhuang. Image analysis using mathematical morphology. *IEEE Trans. Pattern Anal. Mach. Intell.*, 9 (4):532–550, 1987. doi: 10.1109/TPAMI.1987.4767941.

[25] K. He, X. Zhang, S. Ren, and J. Sun. Deep residual learning for image recognition. In *CVPR*, pages 770–778. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.90.

[26] V. Le and S. Gulwani. Flashextract: a framework for data extraction

by examples. In *PLDI*, pages 542–553. ACM, 2014. doi: 10.1145/2594291.2594333.

[27] K. Lee, M. Joshi, I. R. Turc, H. Hu, F. Liu, J. M. Eisenschlos, U. Khandelwal, P. Shaw, M. Chang, and K. Toutanova. Pix2struct: Screenshot parsing as pretraining for visual language understanding. In A. Krause, E. Brunskill, K. Cho, B. Engelhardt, S. Sabato, and J. Scarlett, editors, *ICML*, pages 18893–18912. PMLR, 2023.

[28] S. J. J. Leemans, D. Fahland, and W. M. P. van der Aalst. Discovering block-structured process models from event logs - A constructive approach. In *PETRI NETS*, pages 311–329. Springer, 2013. doi: 10.1007/978-3-642-38697-8_17.

[29] L. A. Leiva, A. Hota, and A. Oulasvirta. Describing UI screenshots in natural language. *ACM Trans. Intell. Syst. Technol.*, 14(1):19:1–19:28, 2023. doi: 10.1145/3564702.

[30] V. Leno, A. Polyvyanyy, M. L. Rosa, M. Dumas, and F. M. Maggi. Action logger: Enabling process mining for robotic process automation. In *Proceedings of the Dissertation Award, Doctoral Consortium, and Demonstration Track at BPM*, volume 2420 of *CEUR Workshop Proceedings*, pages 124–128. CEUR-WS.org, 2019.

[31] V. Leno, A. Polyvyanyy, M. Dumas, M. L. Rosa, and F. M. Maggi. Robotic process mining: Vision and challenges. *Bus. Inf. Syst. Eng.*, 63 (3):301–314, 2021. doi: 10.1007/s12599-020-00641-4.

[32] G. Li and Y. Li. Spotlight: Mobile UI understanding using vision-language models with a focus. In *ICLR*. OpenReview.net, 2023.

[33] Y. Li, G. Li, L. He, J. Zheng, H. Li, and Z. Guan. Widget captioning: Generating natural language description for mobile user interface elements. In *EMNLP*, pages 5495–5510. Association for Computational Linguistics, 2020. doi: 10.18653/v1/2020.emnlp-main.443.

[34] T. Lin, M. Maire, S. J. Belongie, J. Hays, P. Perona, D. Ramanan, P. Dollár, and C. L. Zitnick. Microsoft COCO: common objects in context. In *ECCV*, pages 740–755. Springer, 2014. doi: 10.1007/978-3-319-10602-1_48.

[35] C. Linn, P. Zimmermann, and D. Werth. Desktop activity mining - A new level of detail in mining business processes. In *INFORMATIK - Workshops*, pages 245–258. GI, 2018.

[36] D. G. Lowe. Object recognition from local scale-invariant features. In *ICCV*, pages 1150–1157. IEEE Computer Society, 1999. doi: 10.1109/ICCV.1999.790410.

[37] D. G. Lowe. Distinctive image features from scale-invariant keypoints. *Int. J. Comput. Vis.*, 60(2):91–110, 2004. doi: 10.1023/B:VISI.0000029664.99615.94.

[38] S. Manuel, H. H. Beyel, and W. M. P. van der Aalst. Code and data for "ActivityGen: Extracting Enabled Activities from Screenshots", 2024. Available at: http://doi.org/10.5281/zenodo.13375064.

[39] A. Martínez-Rojas, A. J. Ramirez, J. G. Enríquez, and H. A. Reijers. Analyzing variable human actions for robotic process automation. In *BPM*, pages 75–90. Springer, 2022. doi: 10.1007/978-3-031-16103-2_8.

[40] A. Martínez-Rojas, A. J. Ramirez, J. G. Enríquez, and H. A. Reijers. A screenshot-based task mining framework for disclosing the drivers behind variable human actions. *Inf. Syst.*, 121:102340, 2024. doi: 10.1016/J.IS.2023.102340.

[41] K. Moran, C. Bernal-Cárdenas, M. Curcio, R. Bonett, and D. Poshyvanyk. Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Trans. Software Eng.*, 46(2):196–221, 2020. doi: 10.1109/TSE.2018.2844788.

[42] P. Mukhopadhyay. Youtube thumbnail dataset. https://www.kaggle.com/datasets/praneshmukhopadhyay/youtube-thumbnail-dataset, 2022. Accessed: 2023-08-14.

[43] T. A. Nguyen and C. Csallner. Reverse engineering mobile application user interfaces with REMAUI (T). In *ASE*, pages 248–259. IEEE Computer Society, 2015. doi: 10.1109/ASE.2015.32.

[44] Open AI. https://platform.openai.com/docs/models/gpt-3-5, 2023. Accessed: 2023-08-14.

[45] A. Radford, J. W. Kim, C. Hallacy, A. Ramesh, G. Goh, S. Agarwal, G. Sastry, A. Askell, P. Mishkin, J. Clark, G. Krueger, and I. Sutskever. Learning transferable visual models from natural language supervision. In *ICML*, pages 8748–8763. PMLR, 2021.

[46] J. Redmon, S. K. Divvala, R. B. Girshick, and A. Farhadi. You only look once: Unified, real-time object detection. In *CVPR*, pages 779–788. IEEE Computer Society, 2016. doi: 10.1109/CVPR.2016.91.

[47] N. Reimers, O. Espejel, and P. Cuenca. https://huggingface.co/sentence-transformers/all-MiniLM-L6-v2, 2021. Accessed: 2023-08-14.

[48] S. Ren, K. He, R. B. Girshick, and J. Sun. Faster R-CNN: towards real-time object detection with region proposal networks. In *NeurIPS*, pages 91–99, 2015.

[49] I. Sutskever, O. Vinyals, and Q. V. Le. Sequence to sequence learning with neural networks. In *NeurIPS*, pages 3104–3112, 2014.

[50] R. Syed, S. Suriadi, M. Adams, W. Bandara, S. J. J. Leemans, C. Ouyang, A. H. M. ter Hofstede, I. van de Weerd, M. T. Wynn, and H. A. Reijers. Robotic process automation: Contemporary themes and challenges. *Comput. Ind.*, 115:103162, 2020. doi: 10.1016/j.compind.2019.103162.

[51] S. Torbert. *Applied Computer Science, Second Edition*. Springer, 2016. ISBN 978-3-319-30864-7. doi: 10.1007/978-3-319-30866-1.

[52] W. M. P. van der Aalst. *Process Mining - Data Science in Action, Second Edition*. Springer, 2016. ISBN 978-3-662-49850-7. doi: 10.1007/978-3-662-49851-4.

[53] W. M. P. van der Aalst. Lucent process models and translucent event logs. *Fundam. Informaticae*, 169(1-2):151–177, 2019. doi: 10.3233/FI-2019-1842.

[54] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin. Attention is all you need. In *NeurIPS*, pages 5998–6008, 2017.

[55] T. D. White, G. Fraser, and G. J. Brown. Improving random GUI testing with image-based widget detection. In *ISSTA*, pages 307–317. ACM, 2019. doi: 10.1145/3293882.3330551.

[56] M. Xie, S. Feng, Z. Xing, J. Chen, and C. Chen. UIED: a hybrid tool for GUI element detection. In *ESEC/FSE*, pages 1655–1659. ACM, 2020. doi: 10.1145/3368089.3417940.

[57] Y.-S. Yun, J. Jung, S. Eun, S.-S. So, and J. Heo. Detection of gui elements on sketch images using object detector based on deep neural networks. In *ICGHIT*, pages 86–90, Singapore, 2019. Springer Singapore.

[58] C. Zhang, T. Shi, J. Ai, and W. Tian. Construction of GUI elements recognition model for AI testing based on deep learning. In *DSA*, pages 508–515. IEEE, 2021. doi: 10.1109/DSA52907.2021.00075.

[59] X. Zhang, L. de Greef, A. Swearngin, S. White, K. I. Murray, L. Yu, Q. Shan, J. Nichols, J. Wu, C. Fleizach, A. Everitt, and J. P. Bigham. Screen recognition: Creating accessibility metadata for mobile applications from pixels. In *CHI*, pages 275:1–275:15. ACM, 2021. doi: 10.1145/3411764.3445186.