A tool-supported method to generate User Interface logs

Antonio Martínez-Rojas
University of Seville
amrojas@us.es

Andrés Jiménez-Ramírez
University of Seville
ajramirez@us.es

José González Enríquez
University of Seville
jgenriquez@us.es

Hajo A. Reijers
Utrecht University
h.a.reijers@uu.nl

Abstract

The rise of robotic process automation (RPA) fuels areas like robotic process mining and task mining. Although traditional process mining research can exploit a range of resources (i.e., event logs) to test and benchmark new techniques, that is not the case for robotic process mining. Moreover, benchmark data for RPA needs to incorporate detailed references and properties to elements of the graphical user interface that a software robot is intended to interact with. Therefore, it is not feasible to create such data by hand. To address this omission, the current paper proposes a tool-supported method to generate synthetic event logs for evaluating RPA techniques. To mimic real-life scenarios closely, these logs can be tailored to incorporate variations along a wide range of dimensions. As an application example of the method, the paper describes a case generator tool, which is publicly available, that can be used to benchmark robotic process mining proposals. We also elaborate on further applications of the method in ways that are beneficial to the BPM and RPA communities.

Keywords: Robotic Process Automation, Event Log Generation, Task Mining, User Interface

1. Introduction

Robotic Process Automation (RPA) is a fast-growing technology that leverages user interface (UI) automation. Both industry and academia have shown considerable interest in this technology (Enríquez et al., 2020; Syed et al., 2020). RPA delivers process automation at the UI level, which brings significant benefits. In principle, this means that no involvement of any IT department is required. Moreover, RPA is the only feasible strategy for process automation in contexts where it is not possible to access the underlying IT infrastructure, e.g., when legacy systems or virtual desktop environments are involved.

The focus of this work is on the combination of RPA with the Process Mining paradigm, which is a recurrent theme in recent literature, e.g., Geyer-Klingeberg et al., 2018; Jimenez-Ramirez et al., 2019; Leno et al., 2021; Leno et al., 2019; W. M. van der Aalst, 2021. Research initiatives in this context involve (1) obtaining event logs from the interaction between the user and the front-end of the information systems (the so-called UI logs) and (2) processing them for different aims, e.g., analyzing and discovering the processes to be automated or generating automation scripts themselves. Although existing tools are effective to obtain UI logs from real-world systems (e.g., by means of screen recorders), they may bring along several issues that relate to testing and benchmarking, in particular for academic purposes. First of all, when the logged information comes from real use cases, such a log can rarely be shared publicly because of privacy issues. Secondly, creating synthetic UI logs for controlled experiments is time-consuming and error-prone; it entails repeating similar sequences of interactions continuously in order to produce a log with a reasonable number of events.

To some extent, existing techniques from the Process Mining field can be applied to generate synthetic logs, e.g., by simulating process models (Burattin, 2016; vanden Broucke et al., 2012). In addition, some logs are publicly available and categorized, so that the community can use them to benchmark their new proposals (W. van der Aalst, 2017; van Dongen, 2019). Even in the specific context of RPA, approaches exist that can generate UI logs for experimentation (Agostinelli et al., 2021; Leno et al., 2019). The biggest

drawback of all existing resources and techniques is that they focus on the *events* but neglect the *graphical* information of the UIs that relate to these events, i.e., screen captures. As identified in the recent literature (Jimenez-Ramirez et al., 2019; Jiménez-Ramírez et al., n.d.; Martínez-Rojas et al., 2022), including information on what happens on the screen in the UI log is of the utmost importance for a thorough analysis and support of processes for RPA. We shall refer to this missing aspect in logs as *image data*.

The current paper addresses the generation of synthetic, comprehensive UI logs that offer image data on top of event-related data. The contribution of the proposal is twofold (cf. Fig. 1). First, in order to generate UI logs that meet the desired criteria (e.g., with respect to the number of events, frequency of cases, look and feel of the system), a catalog of generic variability functions is designed to act over different parts of the UI log. Second, a tool-supported method is provided to generate UI logs. More precisely, the generation involves the extension of a given excerpt of a UI log, the so-called *seed log*, by considering a set of configured variability functions on the one hand and the desired variability criteria on the other.

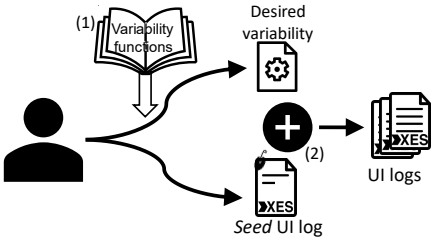

Figure 1. Overall contribution.

We present a case generator tool that is implemented on the basis of the proposed method. This tool is publicly available at http://canela.lsi.us.es/agosuirpa/ and has been used successfully in a research context.

The rest of the paper is organized as follows. Section 2 explains the notion of variability in UI logs and reflects on the state of the art. Section 3 presents the method. Section 4 elaborates on the application of the method to a real research project. Section 5 describes the related work. And, finally, section 6 concludes the paper.

2. Variability in User Interface Logs

UI logs are commonly used for logging the interaction between the user and the front-end of a system while she is performing a process. Different approaches have used UI logs (Agostinelli et al., 2020; Jimenez-Ramirez et al., 2019; Leno et al., 2021) to collect events at a user interface level. Although differences between these exist, their commonalities can be gathered in the following definition:

Definition 1 *A **UI Log** extends the definition of an event log (***xes***) by incorporating additional attributes for each event regarding the corresponding UI interaction. These attributes can be grouped into the following categories: (1) Process attributes: traditional information on events that appear in event logs, e.g., timestamp, event id, or case id. (2) Application attributes: information obtained from the system with which the interaction takes place, e.g., app name, screen capture, screen size, or other attributes obtained from the application. (3) User input attributes: information on the action performed by the user during an event, e.g., type of action (i.e., mouse click, or keystroke), X and Y click coordinates, keystrokes, or other attributes obtained from the user.*

When the user performs a step (i.e., clicking a mouse button or typing text with the keyboard), a new event is stored in the UI log. All steps are grouped into traces that relate to the different cases the user performs, i.e., all the events related to the same instance of the process belong to the same trace, also known as *case*. A UI log may contain traces of different process variants.

In general, UI log attributes can take on different values under different circumstances, but the range of values is typically constrained. Variability of attribute values may be observed across the whole log, within a particular trace, or even for one specific event. For instance, the range of values of the X coordinates of a click event —which can be represented with the *Int* data type— should be inside the vertical resolution of the display in question. This can be understood as a general value constraint for the entire log, i.e., the system resolution is likely to be constant across the log. However, the range of attribute values may be further constrained by the boundaries of the button that is being clicked. This can be understood as specific for this particular event, i.e., the button boundaries determine which X coordinates a click on this button can take on.

While many of the UI log attributes relate to procedural or temporal aspects of an event, some approaches (Jimenez-Ramirez et al., 2019) include the screen capture path as one of them. This feature is quite relevant in contexts where the front-end application cannot be accessed, which is common in the Business Process Outsourcing sector. Here, users typically work through virtual desktops (e.g., Team Viewer or Citrix). Although the reference to the screen capture path is textual, the linked screen capture is an image.

The variability and design of UIs have been researched since the 80s (Kasik, 1982). To date, several

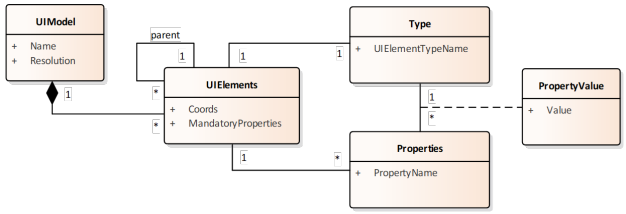

Figure 2. Metamodel for a UI. Adapted from the Concrete UI Model of Akiki et al., 2016

proposals have been developed to devise UIs (Akiki et al., 2014). In general, such approaches deal with the topic of variability on different levels, from the abstract dynamic behavior of the system (i.e., navigability) to the specific implementation of each system view (e.g., the application HTML code). Although modern approaches exist in this area (Martinie et al., 2014; Sanctorum & Signer, 2019), they are all based on a widely-accepted framework called CAMELEON (Calvary et al., 2003). It distinguishes models for the UI on 4 different levels: task, abstract, concrete, and final. It is in the *concrete UI model* where the look and feel of the application is composed of the specific UI elements instances, i.e., buttons, input texts, images, etc.

Although the behavior of the elements enables navigability through the different UIs of an app, the considered UI log explicitly includes how the different UIs (i.e., the screen captures) are visited. By adapting the metamodel suggested by Akiki et al., 2016, which focuses on the static features of a UI, a concrete UI model can be characterized as shown in Fig. 2. This metamodel proposes that each UI consists of a set of UIElements. These can be composed and, according to its UIElementType (e.g., button type or checkbox type), each UIElement exposes a set of attribute-value pairs (e.g., text value for a button or status for a checkbox).

Work on UI papers tends not to focus on a standardized palette of UI elements: the considered elements slightly differ from one author to the other. For instance, Gupta and Mohapatra, 2020 consider a palette of 21 different UI elements, while Moran et al., 2018 consider just 14. Regardless of the types of UI elements that are considered, the variability of a concrete UI model can be divided into two types (Akiki et al., 2016):

- Feature-set: These variations focus on what UI elements are shown in the UI. These variations include adding or deleting UI elements and changing their values or state, e.g., disabling a button or modifying the text of an input text.

- Layout: These variations mainly refer to usability aspects. Variations of this type include changing the position or size of the UI elements or even changing the UI types, e.g., changing a drop-down list by radio buttons or displaying a vertical menu in a horizontal layout.

In conclusion, the variability of image attributes can be defined according to the possible variations in the features-set and layout of the concrete UI, i.e., the screen capture. This insight will drive the design of our method.

3. Generating User Interface Logs

In line with the stated objectives to generate synthetic UI logs, our method requires a user to select which variability functions need to be applied from a given catalog (cf. Sect 3.1). These functions will be used to extend an initial UI log provided by the user into a larger log of a varied nature (cf. Sect. 3.2). A software infrastructure is required to support the suggested method (cf. Sect. 3.3).

3.1. The Variation Function Catalog

The variation function catalog is designed to contain the main utilities to perform atomic variations in a UI log related to both event (i.e., the UI log attributes) and image (i.e., the UI model) information, as presented in Sect. 2. Although the content of the catalog can be extended with new functions, all of them conform to the following definition:

Definition 2 *A **Variation Function** VF=(name, allowedAttributes, f) is identified by its* $name$*, can be applied to any of the log attributes included in* $allowedAttributes$*, and defines a function* f *that receives the following parameters:*

1. *The triplet* UI log, event id*, and* attribute name *which points to the value that the function varies.*

2. *A* class *value which links the styles between VFs.*

3. *A* memory *which is a multimap that enables exposing information between variation functions. The key of this multimap is the triplet (event Id, attribute name, VF name), and the values are any information that the VF wants to store to be consulted by other VFs. Since more than one VF can be applied to the same attribute, the memory is a multimap instead of a map.*

4. *A custom list of* params *required by the function.*

In addition, f *returns the new value of the attribute.*

A variation function can be used to vary the value of an attribute of an event in the UI log. It is possible to create VFs that are independent of some

of the parameters. For instance, the *ClickOnCoordsVF* example (cf. Alg. 1), which can be applied to the *X coordinate* or *Y coordinate* attributes, only returns the X or Y coordinate in a bounding box received as $params$, i.e., the function returns a random number between an upper and lower bound. This random number is also stored in the memory to allow for reuse by other VFs.

Algorithm 1: *ClickOnCoordsVF* Example

Input : UILog log, Int $eventId$, Str $attName$, Int $class$, Multimap $memory$, Int $lBound$, Int $uBound$
Output: Int $newValue$

1 $newValue \leftarrow randomInt(lboud, uBound)$
2 $memory.put((eventid, attname, "ClickOnCoordsVF"), newvalue)$

Next to the catalog, the proposed method includes the *UI repository*, which contains UI elements to ease the definition of the VFs related to changes in the *feature-set* of the images. This repository allows filtering the UI elements by the use of three parameters: (1) the UI element *type* (e.g., checkbox, button, etc.), (2) the *state* that depends on the type (e.g., pushed and released for buttons, checked and unchecked for checkboxes), and (3) the *family* id, which refers to the look and feel of the UI element. More than one image can be stored in the repository for a single UI element [1] yet when the repository is accessed exactly one random image is returned.

To illustrate the use of the UI repository by the VFs, consider the following example (cf. Alg. 2). The *CheckBoxVF* is applied to the *screencapture* attribute. Then, it receives the bounding box where a checkbox UI element appears in the screen capture. So, this function substitutes the existing checkbox with another checkbox version from the UI repository. In this case, the VF $class$ attribute is passed as the UI Element $family$. The idea is to keep the same look and feel between the different VFs of the same case in the UI log. This function makes the new checkbox that is gathered from the $UIRepo$ available in memory so that other VFs can use it when required.

To access elements from memory, the current approach supports the following expressions: $\$memory.get(eventId, attrName, VFName)[order]$ Where the order is the index in the collection returned for this key. This type of expression can be used *inside* a function as well as a *parameter* when a VF is selected, which will be dynamically resolved when the VF function is applied.

[1]For this approach, the images included in the repository corresponds to the dataset proposed by Moran et al., 2018

Algorithm 2: *CheckBoxVF* Example

Input : UILog log, Int $eventId$, Str $attName$, Int $class$, Multimap $memory$, Int $trBound$, Int $tlBound$, Int $brBound$, Int $blBound$
Output: Str $newValue$

1 Str $path \leftarrow log.get(eventId, "screencatpure")$
2 Str $chkBxImgPath \leftarrow UIRepo.get([\text{type="}checkbox\text{",family=}class])$
3 $memory.put((eventid, attname, "CheckBoxVF"), chkBxImgPath)$
4 $ImgDraw(path, chkBxImgPath, trBound, tlBound, brBound, blBound)$
5 $newValue \leftarrow path$

In general, event information within the UI log can be varied with a single VF. However, to sufficiently vary image attributes it may be desirable to use more VFs. As can be seen in the *CheckBoxVF* example, the function modifies the input image (the *path* variable) by the abstract method *ImgDraw*. This places a given image in the place indicated by the bounding box, so it modifies the linked image. Another sequence of VFs could be applied to the same image attribute to also alter other parts of that image.

The current proposal includes 8 VFs for event attributes and 16 VFs for image attributes. They can be consulted at https://github.com/RPA-US/agosuirpa/wiki/Variability-Functions

3.2. Generating UI logs from a Seed Log

Instead of generating the UI logs from scratch, the generator *extends* and *variates* a given *seed log*.

Definition 3 *A **seed log** is a UI log that contains only one case for each process variant.*

In order to configure the generation of the UI log, the method requires the following parameters:

- $LogSize$: the desired size (i.e., number of events) of the log. The seed log will be *extended* with events until this number is reached. The final log may be slightly larger since it only contains full cases. Thus, the size will depend on the number of events of each process variant in the seed log.
- $Weights$: the frequency weight of each process variant in the generated log. This list of values gives a relative weight to each process variant, so

that variations of the process variants with higher weights are more likely to appear in the generated log. A fully-balanced log is achieved using the same weights for all the alternatives.

- $VFParams$: a list of tuples *(location, name, params)* that specify where to *variate* the seed log: $location$ is the concrete event in the seed log where the VF is to be applied (i.e., it identifies the event and attribute), $name$ identifies the VF to be applied, and $params$ is the list of custom parameters for the VF.[2]

Algorithm 3: *UILog Generation* Algorithm

Input : UILog $seedLog$, Int $LogSize$, List $Weights$, List $VFParams$
Output: UILog $newLog$

1 List $traces$ ←separateCases($seedLog$)
2 **while** $|newLog| \leq LogSize$ **do**
3 Trace $trace$ ←random($traces, Weights$)
4 $newLog \xleftarrow{+} variate(trace, VFParams)$
5 **end**

The overall algorithm that governs the generation is shown in Alg. 3. First, the list of process alternatives of the seed log (i.e., each case) is separated into traces. Then, as long as the generated log has not exceeded the required $LogSize$, one of these traces is randomly selected according to the given $Weights$. The resulting log will include the trace after applying the variability functions indicated in $VFparam$ (cf. Alg. 4).

Algorithm 4 runs over all the events of the trace to create a variation of each one. A default VF behavior to calculate the process attributes of the log (i.e., event, case, activity ids, and timestamp, cf. Def. 1) is applied when it does not make sense to override (cf. line 2 in Alg. 4). This function (1) calculates the next *ids* following incremental counters for events and cases, (2) copies the activity id, and (3) calculates the next timestamp of the event preserving the intervals of the original seed log. The rest of the event attributes are calculated in two different ways. The first option is that those attributes that do not present a VF associated with them (i.e., the $location$ parameter does not point to them) are just copied from the seed log (cf. line 4 in Alg. 4). Note that $screencapture$ attributes are always modified; therefore, the *copy* function replicates the image indicated by the attribute's value (i.e., the path) and returns the new path. The other option applies when the $location$ of a VFParam points to the current

[2]Note that the $allowedAttributes$ field of the VF (cf. Def. 2) must include the attribute in the $location$.

Algorithm 4: *Trace Variation* Algorithm

Input : Trace $trace$, List $params$
Output: UILog $newTrace$

1 **foreach** *Event* $event \in trace$ **do**
2 Event $newEvent \leftarrow copyDefaults(event)$
3 **foreach** *Attribute* $att \in event$ **do**
4 Attribute $newAtt \leftarrow copy(att.name, att.value)$
5 **if** $params.contain(trace.id, event.id, att.id)$ **then**
6 **foreach** *VFParam* $p \in params.get(trace.id, event.id, att.id)$ **do**
7 VariabilityFunction $vf \leftarrow VFRepo.get(p.name)$
8 $newAtt.value \leftarrow vf.f(newAtt.value, trace.id, p.params)$
9 **end**
10 **end**
11 $newEvent \xleftarrow{+} (newAtt.name, newAtt.value)$
12 **end**
13 $newTrace \xleftarrow{+} newEvent$
14 **end**

attribute. Then, its function f is applied to generate the new attribute (cf. lines 5 to 8 in Alg. 4). When calling such function, the trace.id is considered as the *class*, so that all the events of the same trace are changed with a similar look and feel. Note that more than one VF can point to the same attribute and, therefore, all of them are sequentially applied to generate the new attribute. As the last step, the resulting attribute is included in the event and, when all the attributes are processed, the created event is included in the new trace.

If the user does not include any VF in the method, an extended UI log will be obtained with the same application attributes, user input attributes, and screen captures as the seed log. Clearly, this would represent a rather unrealistic UI log. While the application attributes of the UI log tend to be the same between different cases of the same process variant (i.e., they can be directly copied from the seed log), the user input attributes and the screen captures are likely to differ.

Example 1 To illustrate this proposal, Fig. 3 depicts an example process with synthetic screen captures for each activity. The process consists of

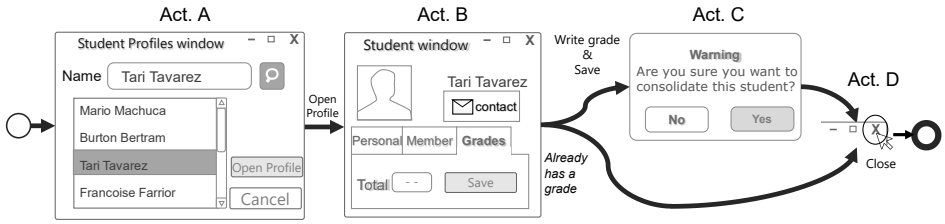

Figure 3. Process screenshots of a teacher entering students' marks.

#ID	case	activity	timestamp	event_type	keystrokes	click_type	click_coords	screencapture
1	1	A	2022/01/10-0:4:32	keystroke	"Tari Tavarez"			img_1
2	1	A	2022/01/10-0:4:35	click		Left	120,205	img_1
3	1	B	2022/01/10-0:4:50	click		Left	98,172	img_2
4	1	B	2022/01/10-0:4:35	keystroke	"10"			img_2
5	1	B	2022/01/10-0:4:50	click		Left	10,240	img_2
6	1	C	2022/01/10-0:5:09	click		Left	115,206	Img_3
7	1	D	2022/01/10-0:5:19	click		Left	130,126	img_4
8	2	A	2022/01/10-0:5:15	keystroke	"Burton Bert"			img_5
9	2	A	2022/01/10-0:5:19	click		Left	122,211	img_5
10	2	B	2022/01/10-0:5:22	click		Left	98,172	img_6
11	2	D	2022/01/10-0:5:25	click		Left	130,126	img_7

Figure 4. Seed log for the example of a teacher entering students marks.

4 activities that a teacher performs to consolidate students' marks through an web system: Search student profiles (Act.A), Open profile (Act.B), Confirm edition (Act.C), and Close profile (Act.D).

To create a synthetic UI log from this example, a *seed log* (cf. Fig. 4) must be provided. In this example, the log consists of 11 events and 2 distinct cases, i.e., one for each process variant. In addition, we specify: (1) the desired *log size* (e.g., 150 events), (2) the frequency *weights* (e.g., [80,20], meaning that process variant 1 will be 4 times more likely to appear in the resulting UI log than process variant 2), and (3) a number of *VFParams*.

For this example, we include VFs to vary the student names across the different cases. This name appears in three places: keystroke attribute in event #1, two times in the screen capture of event #1 (i.e., in Act.A), and in the screen capture of event #3 (i.e., Act.B). To exemplify the use of the method, let us consider how this can be done for the keystroke attribute and for the first screen capture.

To vary the student name that is typed by the user, we apply the VF with the *name* $random\_text\_from\_list$ to the *location* (#1, "keystrokes") (cf. Def. 2). The custom *parms* of this function is a list of strings, e.g., ["Student 1", "tudent 2", "Student 3"]. At random, one of these strings is selected, which will replace the value in the "keystrokes" column. Next to that, this VF stores the selected string in the *memory*, which may be of use for another function.

Second, this student's name must appear in the list shown in the associated screenshot, i.e., img_1. To make this happen, we apply a new VF with the *name* $write\_text\_in\_image$ to the *location* (#1, "screencapture"). Here, the custom *params* are (1) the *text* and (2) the *bounding box* where to print the text inside the screenshot. As the reader may notice, the text that must be printed is exactly the name selected by the previous VF. Therefore, to specify the *text* parameter, access to memory is opened with the following expression: $\$memory.get(1, "keystrokes", "random\_text\_from\_list")[0]$. This expression points to the value stored in the memory when the function $random\_text\_from\_list$ was applied to (#1, "keystrokes") the first time (cf. [0]), i.e., the student name.

In a highly similar way, the dependency between this keystrokes value and the rest of the occurrences in the trace can be established using the same expression as above.

3.3. Software Infrastructure

Figure 5 depicts the infrastructure developed to support the presented method.[3] The 5 data units, developed in PostgreSQL, are the following: (1) The

[3]The source code can be accessed at https://github.com/RPA-US/agosuirpa.

VFCatalog contains the catalog of the implemented variability functions written in Python (cf. Def. 2). These VFs can be accessed by their *name*, *allowedAttributes*, or *class*. (2) The *UIRepo* includes UI elements crops to be used by the VFs that require working with screen capture attributes. These UI element crops can be accessed by their UI element *type*, *state* if it has, and the *family* which refers to the look and feel. (3) The *LogStore* manages the logs used by the method. This store keeps track of both the initial seed logs and the generated logs. (4) The *ScreenStore* registers the images corresponding to the screen captures linked in the log. In addition, this store keeps the information about the UI elements that have been identified in the screen capture so that it becomes easier to identify them when creating the VFs related to variations in the UI log. (5) The *ConfigStore* keeps the configuration history that has been used to generate the UI logs from the seed logs. That is, it keeps track of the *LogSize*, *Weights*, and *VFParams* that are specified. The *LogsStore*, *ScreensStore*, and *ConfigStore* are securely stored for each user of the platform.

In addition, the infrastructure contains the following processing components: (1) The *REST API*, as offered by the server, developed with the Django framework. It exposes the main platform capabilities to be used as a service. It can be accessed at http://canela.lsi.us.es/api/v1/docs/. (2) The *UI Analyzer*, which is a component developed in Python with the OpenCV library. It is in charge of processing the screen captures of the *ScreenStore* to detect the existing UI elements and complements the information to be included when defining the VF parameters in the configurations. (3) The *User Configuration Wizard*, which is the front-end developed with React. It assists the user to interact with the platform, i.e., to create the configurations.

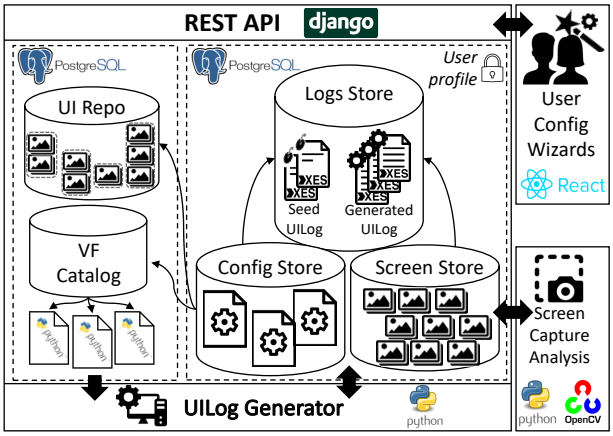

Figure 5. Developed infrastructure

Since creating a complete configuration might entail some complexity, this wizard allows a user to create it iteratively and interactively. The wizard support both the event and the image attributes. As to the latter, as can be seen in Fig. 6, the wizard even provides support to select or define the crops in the image, which helps to ensure the correct definition of the VFs. (4) The *Log Generator*, as developed in Python, which is the main component. It implements the method described in Sect. 3. More precisely, it offers a public endpoint for Alg. 3.

4. Application

To demonstrate the applicability of our proposal, we will discuss a specific tool that has been used in a recent research project. The set-up of the tool is described in Sect. 4.1, while Sect. 4.2 reflects on its benefits.

4.1. Set-up of AGOSUIRPA

The method proposed in Sect. 3 has been used for the implementation of a tool, called AGOSUIRPA, which can be accessed at http://canela.lsi.us.es/. Its aim is to support users in two tasks: (1) generating synthetic cases for case studies or experimentation in the context of RPA combined with Process Mining, and (2) publishing the cases as open data so that other users can both use and extend them. Against this backdrop, the tool comprises three phases (cf. Fig. 7).

First, in the **scenario** generation phase, the seed UI log provided by the user is replicated to have a set of scenarios, i.e., modified seed UI logs. This phase addresses an issue when doing experimentation: results may only be statistically significant if obtained from a sufficiently large number of similar scenarios. Therefore, the number of desired scenarios and a first variability configuration are required. The VFs included in this phase must focus on making changes in the look and feel of the images rather than making more detailed changes in the event information. It is important to note that in this setting the seed UI log is just varied but not extended. So, the resulting UI log for each scenario has the same size as the seed UI log.

Second, in the **case** generation phase, the method is applied to the seed UI log of each scenario. In contrast to the first phase, the seed logs are extended and varied according to a variability configuration, a list of log sizes, and a list of frequency weights. This second variability configuration should include VFs that go beyond the look and feel in the sense that the newly generated events present meaningful differences from the events in the seed UI log. Since experimentation is typically performed over cases of different properties, this tool allows specifying two main property sets that

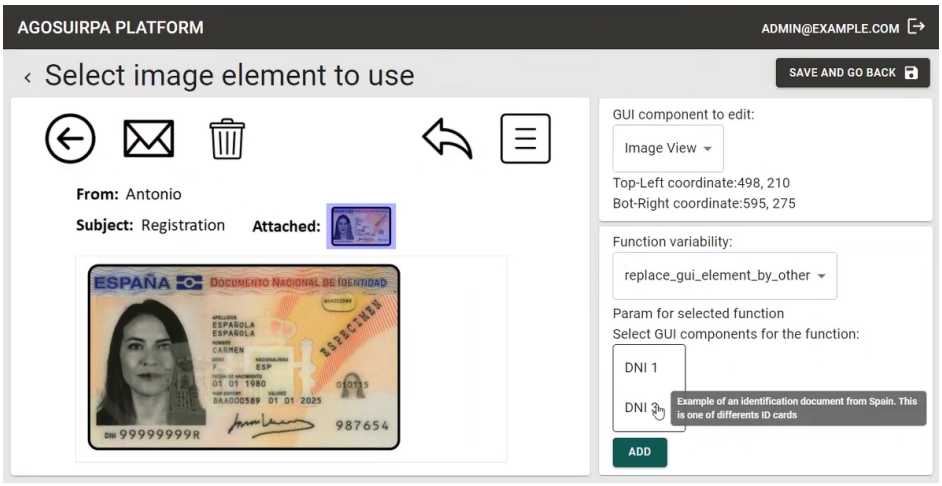

Figure 6. The User Configuration Wizard window which assist the user when configuring a VF related to a screen capture (left). This windows allows crop selections (in blue) and the configuration of a VF parameters (right).

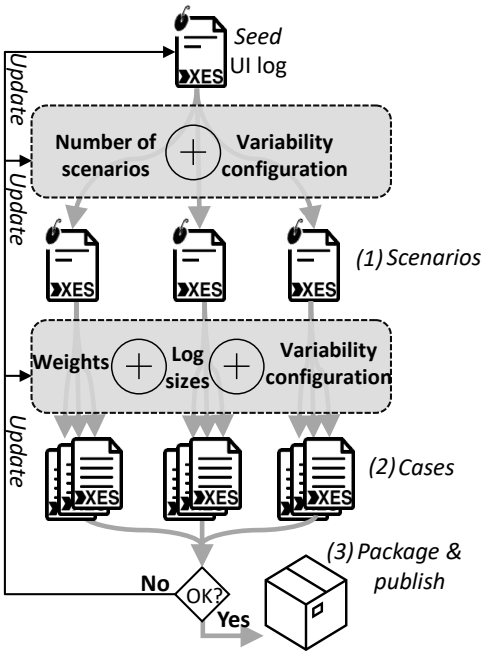

Figure 7. Case generator process.

recurrently appear in Process Mining experimentation: the log size and the balancing of cases. For each combination of log size and frequency weights, the tool invokes the method with the same seed UI log of the scenario and VFs. That is, the tool will generate a number of $|screnarios| \times |logSizes| \times |frequencyWeights|$ cases, i.e., UI logs.

Third, if the user considers that the resulting cases are valid for experimentation, they can be packaged and made **publicly available** on the platform. This publication includes the resulting UI logs, a user-given description, and the configuration used in both the scenario and the case generation phases. In addition, if any issue is detected in the resulting cases, the user can update the input seed UI log or the configuration of any of the previous phases to obtain a new version of the cases until the desired data is achieved.

4.2. Benefits of AGOSUIRPA

This development of AGOSUIRPA can be seen as a concrete application in the academic context of the method that we proposed in the present work. More precisely, in their research paper, Martínez-Rojas et al., 2022 describe how the conditions that govern decisions can be discovered in those business processes where the relevant information is on the screen rather than in the UI log. For this purpose, AGOSUIRPA was used to generate cases for three different processes. This data was specifically designed to evaluate the quality of the method for decision discovery. The three processes contain 6 activities on average, a single decision point each, and more than a single variant per process.

The AGOSUIRPA tool was highly beneficial in the described setting since it helped to efficiently and automatically generate the cases for these processes. By using AGOSUIRPA, the team only needed (1) to specify a seed UI log for each of these processes and (2) to configure the scenario generator phase and the case generator phase. In particular, 40 screen captures and events were designed for the 3 processes in total. In addition, 92 variability functions were configured in the tool. For each of the 3 processes, 30 scenarios, with 4 different log sizes, and 2 frequency weights were selected. Eventually, the tool generated 720 UI logs with a number of 4320 corresponding screen captures. The generated UI logs associated with each of these processes can be accessed at: https:

//canela.lsi.us.es/agosuirpa/experiment/1, https://canela.lsi.us.es/agosuirpa/experiment/2, and https://canela.lsi.us.es/agosuirpa/experiment/3 respectively.

Although no exact time measurement took place to compare the situation with and without the tool, the time invested with the tool was a matter of days, while manual edits of the UI logs and images are likely to have required weeks of work. The use of the data helped to show that the discovery method was robust under the varying properties of the processes and UIs involved.

5. Related Work

Other approaches exist for the generation of event logs and UIs. In the context of event logs, the Process Mining community has been actively contributing with both tools (Burattin, 2016; Jouck & Depaire, 2019; vanden Broucke et al., 2012) and benchmarks (W. van der Aalst, 2017; van Dongen, 2019). vanden Broucke et al., 2012 defined a plugin for ProM[4] to generate event logs from a given Petri Net, which allows to inject rare events which, typically, are complicated to capture in real systems. In turn, Burattin, 2016 describes a standalone tool that can generate random process models and simulate them to obtain event logs. Similarly, Jouck and Depaire, 2019 define a methodology for generating artificial logs by simulating models. In addition, other tools with similar objectives can be found online, e.g., ("BIMP simulator tool", 2022), which uses a BPMN model as input to generate an event log. Next to these tools, van Dongen, 2019 put together many event logs as a challenge for researchers. With a similar objective, W. van der Aalst, 2017 includes several event logs and their corresponding process model so that the community can use these as benchmarks. Unlike the present work, these approaches focus on event information of the event log, lacking any references to the image attributes that might be associated with them. Furthermore, although some of the discussed approaches enable configuring the generation of logs, the configuration options are often more coarse-grained than in the present proposal.

In the context of the generation of UIs, several research proposals (Akiki et al., 2016; Martinie et al., 2014; Sanctorum & Signer, 2019) base the modeling of the UIs of an information system on different levels. This ranges from the most abstract level (i.e., the navigability of the system) to the most concrete level (i.e., the system's source code). Although these levels vary from author to author, they are based on the *de-facto* task model standard (Calvary et al., 2003). Besides this, Abb and Rehse, 2022 offers a reference data model for UI logs, which could be used as an alternative to the metamodel used in this work, although it is not intended for the subsequent generation of variability based on it. Some proposals go beyond the definition of a singular UI. They provide mechanisms to derive UI variants from a given feature model (Martinez et al., 2017) so that a different configuration of features will produce a different UI. Unlike the present work, these proposals focus on automating the transition from abstract levels (e.g., task models or feature models) to a more concrete level. They generally lack support for generating UI variations at the same abstraction level.

[4]ProM is a Process Mining framework.

6. Conclusion

The present work arises from the need for a methodological framework (cf. Sec. 3) and a support tool (cf. Sect. 4) that allows the scientific community to generate synthetic event logs to evaluate RPA techniques simply and systematically. The method's applicability has been demonstrated with a concrete tool for an academic research project, which shows the usefulness and efficiency of the proposal. In particular, it turned out to be possible to synthetically generate a multitude of varied event logs for evaluation purposes.

The proposed method has limitations. Two types of UI variability were identified in Sect. 2. However, the current version of the approach neglects most of the *layout* types. For example, the method does not provide efficient mechanisms to vary the type of UI elements or make significant changes to the layout, which in turn, may be rare to occur inside a realistic UI log. Nevertheless, the *feature-set* type is widely covered, enabling us to consider a wide spectrum of realistic cases. To address these shortcomings, implementing such new mechanisms is planned in our roadmap. Moreover, future work encompasses: (1) defining a case variables list as input, so that data can be generated including this level of detail with Variability Functions, and (2) developing new mechanisms to automatically detect the look and feel of the application in terms of fonts, sizes, and icons according to the seed log.

Acknowledgments: This research has been supported by the Spanish Ministry of Science, Innovation and Universities under the NICO project (PID2019-105455GB-C31) and by the FPU scholarship program, granted by the Spanish Ministry of Education and Vocational Training (FPU20/05984)

References

Abb, L., & Rehse, J.-R. (2022). A reference data model for process-related user interaction logs.

International Conference on Business Process Management, 57–74.

Agostinelli, S., Lupia, M., Marrella, A., & Mecella, M. (2020). Automated generation of executable RPA scripts from user interface logs. *BPM*, 116–131.

Agostinelli, S., Lupia, M., Marrella, A., & Mecella, M. (2021). Smartrpa: A tool to reactively synthesize software robots from user interface logs. *CAiSE*, 137–145.

Akiki, P. A., Bandara, A. K., & Yu, Y. (2014). Adaptive model-driven user interface development systems. *ACM Computing Surveys*, *47*(1), 1–33.

Akiki, P. A., Bandara, A. K., & Yu, Y. (2016). Engineering adaptive model-driven user interfaces. *IEEE Transactions on Software Engineering*, *42*(12), 1118–1147.

BIMP simulator tool [Available at https://bimp.cs.ut.ee/simulator]. (2022).

Burattin, A. (2016). PLG2: multiperspective process randomization with online and offline simulations. *BPM Demo Track*, 1–6.

Calvary, G., Coutaz, J., Thevenin, D., Limbourg, Q., Bouillon, L., & Vanderdonckt, J. (2003). A unifying reference framework for multi-target user interfaces. *Interacting with computers*, *15*(3), 289–308.

Enríquez, J. G., Jiménez-Ramírez, A., Domínguez-Mayo, F., & García-García, J. (2020). Robotic process automation: A scientific and industrial systematic mapping study. *IEEE Access*, *8*, 39113–39129.

Geyer-Klingeberg, J., Nakladal, J., Baldauf, F., & Veit, F. (2018). Process mining and robotic process automation: A perfect match. *BPM*, 124–131.

Gupta, P., & Mohapatra, S. (2020). Html atomic ui elements extraction from hand-drawn website images using mask-rcnn and novel multi-pass inference technique. *CLEF (Working Notes)*.

Jimenez-Ramirez, A., Reijers, H. A., Barba, I., & Del Valle, C. (2019). A method to improve the early stages of the robotic process automation lifecycle. *CAiSE*, 446–461.

Jiménez-Ramírez, A., Chacón-Montero, J., Wojdynsky, T., & González Enríquez, J. (n.d.). Automated testing in robotic process automation projects.

Jouck, T., & Depaire, B. (2019). Generating artificial data for empirical analysis of control-flow discovery algorithms: A process tree and log generator. *Business & Information Systems Engineering*, *61*.

Kasik, D. J. (1982). A user interface management system. *SIGGRAPH*, 99–106.

Leno, V., Polyvyanyy, A., Dumas, M., La Rosa, M., & Maggi, F. M. (2021). Robotic process mining: Vision and challenges. *Business & Information Systems Engineering*, *63*(3), 301–314.

Leno, V., Polyvyanyy, A., La Rosa, M., Dumas, M., & Maggi, F. M. (2019). Action logger: Enabling process mining for robotic process automation.

Martinez, J., Sottet, J.-S., Frey, A. G., Ziadi, T., Bissyandé, T., Vanderdonckt, J., Klein, J., & Le Traon, Y. (2017). Variability management and assessment for user interface design. *Human centered software product lines* (pp. 81–106). Springer.

Martínez-Rojas, A., Jiménez-Ramírez, A., Enríquez, J. G., & Reijers, H. (2022). Analyzing variable human actions for robotic process automation. *International Conference on Business Process Management*, 75–90.

Martinie, C., Navarre, D., & Palanque, P. (2014). A multi-formalism approach for model-based dynamic distribution of user interfaces of critical interactive systems. *International journal of human-computer studies*, *72*(1), 77–99.

Moran, K., Bernal-Cárdenas, C., Curcio, M., Bonett, R., & Poshyvanyk, D. (2018). Machine learning-based prototyping of graphical user interfaces for mobile apps. *IEEE Transactions on Software Engineering*, *46*(2), 196–221.

Sanctorum, A., & Signer, B. (2019). A unifying reference framework and model for adaptive distributed hybrid user interfaces. *RCIS*, 1–6.

Syed, R., Suriadi, S., Adams, M., Bandara, W., Leemans, S. J., Ouyang, C., ter Hofstede, A. H., van de Weerd, I., Wynn, M. T., & Reijers, H. A. (2020). Robotic process automation: Contemporary themes and challenges. *Computers in Industry*, *115*, 103162.

van der Aalst, W. (2017). Testing Representational Biases.

van der Aalst, W. M. (2021). Process mining and rpa: How to pick your automation battles? *Robotic Process Automation: Management, Technology, Applications. De Gruyter*, 223–239.

vanden Broucke, S., De Weerdt, J., Vanthienen, J., & Baesens, B. (2012). An improved process event log artificial negative event generator.

van Dongen, B. (2019). Dataset BPI Challenge 2019 [4TU.Centre for Research Data].