Identifying Candidate Routines for Robotic Process Automation from Unsegmented UI Logs

Volodymyr Leno*†, Adriano Augusto*, Marlon Dumas†, Marcello La Rosa*,
Fabrizio Maria Maggi‡†, Artem Polyvyanyy*
*University of Melbourne, Australia
{a.augusto, marcello.larosa, artem.polyvyanyy}@unimelb.edu.au
†University of Tartu, Estonia
{leno, marlon.dumas}@ut.ee
‡Free University of Bozen-Bolzano, Italy
maggi@inf.unibz.it

***Abstract*—Robotic Process Automation (RPA) is a technology to develop software bots that automate repetitive sequences of interactions between users and software applications (a.k.a. routines). To take full advantage of this technology, organizations need to identify and to scope their routines. This is a challenging endeavor in large organizations, as routines are usually not concentrated in a handful of processes, but rather scattered across the process landscape. Accordingly, the identification of routines from User Interaction (UI) logs has received significant attention. Existing approaches to this problem assume that the UI log is segmented, meaning that it consists of traces of a task that is presupposed to contain one or more routines. However, a UI log usually takes the form of a single unsegmented sequence of events. This paper presents an approach to discover candidate routines from unsegmented UI logs in the presence of noise, i.e. events within or between routine instances that do not belong to any routine. The approach is implemented as an open-source tool and evaluated using synthetic and real-life UI logs.**

***Index Terms*—Robotic process automation, robotic process mining, user interaction log.**

I. Introduction

Robotic Process Automation (RPA) allows organizations to improve their processes by automating repetitive sequences of interactions between a user and one or more software applications (a.k.a. routines). With this technology, it is possible to automate data entry, data transfer, and verification tasks, particularly when such tasks involve multiple applications. To exploit this technology, organizations need to identify routines that are prone to automation [1]. This can be achieved via interviews or manual observation of workers. In large organizations, however, this approach is not always cost-efficient as routines tend to be scattered across the process landscape. In this setting, manual routines identification efforts can be enhanced via automated methods, for example methods that extract frequent patterns from these User Interaction (UI) logs of working sessions of one or more workers [2].

Existing methods in this space [3]–[6] assume that the event log consists of a set of traces of a task that is presupposed to contain one or more routines. When the log is segmented, the identification of candidate routines boils down to discovering frequent sequential patterns from a collection of sequences, a problem for which a range of algorithms exist.

In practice, though, UI logs are not segmented. Instead, a recording of a working session consists of a single sequence of actions encompassing many instances of one or more routines, interspersed with other events that may not be part of any routine. Traditional approaches to sequential pattern mining, particularly those that are resilient to noise (irrelevant events) are not applicable to such unsegmented logs.

This paper addresses this gap by proposing a method to automatically split an unsegmented UI log into a set of segments, each representing a sequence of steps that appears repeatedly in the unsegmented UI log. Once the log is segmented, sequential pattern mining techniques are used to discover candidate routines. The method has been evaluated on synthetic and real-life UI logs in terms of its ability to rediscover routines contained in a log and in terms of scalability.

The paper is structured as follows. Section II provides the necessary background and an overview of related work. Section III describes the approach, while Section IV reports the results of the evaluation. Finally, Section V concludes the paper and spells out directions for future work.

II. Background and Related Work

The problem addressed by this paper can be framed in the broader context of Robotic Process Mining (RPM) [2]. RPM is a family of methods to discover repetitive routines performed by employees during their daily work, and to turn such routines into software scripts that emulate their execution. The first step in an RPM pipeline is to record the interactions between one or more workers and one or more software applications [7]. The recorded data is represented as a UI log – a sequence of user interactions (herein called UIs), such as selecting a cell in a spreadheet or editing a text field in a form. The UI log may be filtered to remove irrelevant UIs (e.g. misclicks). Next, it may be decomposed into segments. The discovered segments are scanned to identify routines that occur across these segments. Finally, the resulting routines are analysed to identify those that are automatable and to encode them as RPA scripts.

The problem of routines identification from UI logs in the context of RPM has attracted significant attention [7]. However, existing approaches for routines identification from UI logs either take as input a segmented UI log [3], [5], [6], [8], or they assume that the log can be trivially segmented by breaking it down at each point where one among a set of "start

events" appears [4]. These start events, which act as delimiters between segments, need to be designated by the user.

Another technique for routines identification [1] attempts to identify candidate routines from textual documents – an approach that is suitable for earlier stages of routines identification and could be used to determine which processes or tasks could be recorded and analyzed in order to identify routines.

Below, we review the literature related to UI log segmentation and identification of routines from (segmented) logs.

A. UI Log Segmentation

Given a UI log, i.e. a sequence of UIs, segmentation consists in identifying non-overlapping subsequences of UIs, namely *segments*, such that each subsequence represents the execution of a task performed by an employee from start to end. In other words, segmentation searches for repetitive patterns in the UI log. In an ideal scenario, we would observe only one unique pattern (the task execution) repeated a finite number of times. However, in reality, this scenario is unlikely to materialise. Instead, it is reasonable to assume that an employee performing X-times the same task would do some mistakes or introduce variance in how the task is performed.

The problem of segmentation is similar to periodic pattern mining on time series. While several studies addressed the latter problem over the past decades [9], [10], most of them require information regarding the length of the pattern to discover, or assume a natural period to be available (e.g. hour, day, week). This makes the adaptation of such techniques to solve the problem of segmentation challenging, unless periodicity and pattern length are known a-priori.

Under the same class of problems, we find web session reconstruction [11], whose goal is to identify the beginning and the end of web navigation sessions in server log data, e.g. streams of clicks and web pages navigation [11]. Methods for session reconstruction are usually based on heuristics that rely either on IP addresses or on time intervals between events. The former approach is not applicable in our context (routines in UI logs cannot be segmented based on IP addresses), while the latter approach assumes that users make breaks in-between two consecutive segments – in our case, two routine instances.

Lastly, segmentation also relates to the problem of correlation of event logs for process mining. In such logs, each event should normally include an identifier of a process instance (case identifier), a timestamp, an activity label, and possibly other attributes. When the events in an event log do not contain explicit case identifiers, they are said to be uncorrelated. Various methods have been proposed to extract correlated event logs from uncorrelated ones. However, existing methods in this field either assume that a process model is given as input [12] or that the underlying process is acyclic [13]. Both of these assumptions are unrealistic in our setting: a process model is not available since we are exactly trying to extract that information from the log (by identifying the routines), and a routine may contain repetitions.

B. Routines Identification

Once the UI log is segmented, the segments are scanned to identify routines. Dev and Liu [3] have noted that the problem of routines identification from (segmented) UI logs can be mapped to that of frequent pattern mining, a well-known problem in the field of data mining [14]. Indeed, the goal of routines identification is to identify repetitive (frequent) sequences of interactions, which can be represented as symbols. In the literature, several algorithms are available to mine frequent patterns from sequences of symbols. Depending on their output, we can distinguish two types of frequent pattern mining algorithms: those that discover only exact patterns [15], [16] (hence vulnerable to noise) and those that allow frequent patterns to have gaps within the sequence of symbols [17], [18] (hence noise-resilient). Depending on their input, we can distinguish between algorithms that operate on a collection of sequences of symbols and those that discover frequent patterns from a single long sequence of symbols [16]. The former algorithms can be applied to segmented UI logs while the latter can be applied directly to unsegmented ones. However, techniques that identify patterns from a single sequence of symbols only scale up when identifying exact patterns.

III. Approach

In this section, we describe our approach for identifying candidate routines in UI logs. As input, the approach takes a preprocessed and normalized UI log and outputs a set of candidate routines. The approach follows the RPM pipeline [2], and consists of two macro steps. First, the normalized UI log is decomposed into segments. Then, candidate routines are identified by mining frequent sequential patterns from the segments. In this paper, we refer to these two macro steps as *segmentation* and *candidate routines identification*, respectively. The approach is summarized in Fig. 1. Next, we describe the steps in detail, including the required UI log preprocessing and normalization.

A. UI Log Preprocessing and Normalization

Before we proceed, we give some formal definitions necessary to support the subsequent discussions.

*Definition 1 (**User Interaction (UI)**):* A *user interaction* (UI) is a tuple $u = (t, \tau, P, Z, \phi)$, where: t is a UI timestamp; τ is a UI type (e.g. click button, copy cell); P_τ is a set of UI parameters (e.g. button name, worksheet name, url, etc.); Z is a set of UI parameters values; and $\phi : P_\tau \rightarrow Z$ is a function that maps UI parameters onto values.

*Definition 2 (**UI Log**):* A UI Log Σ is a sequence of user interactions $\Sigma = \langle u_1, u_2, \ldots, u_n \rangle$, ordered by timestamp, i.e. $u_{i|t} < u_{j|t}$ for any $i, j \mid 1 \leq i < j \leq n$. In the remainder of this paper, we also call a UI log simply log.

Ideally, the UIs recorded in a UI log should capture only the execution of the task under recording. However, a log often contains also UIs that do not bring any value to the recorded task, and that should not have been executed in the first place. Some common examples of such UI *noise* include: an employee replying to incoming emails and/or browsing the web while executing a different task; or an employee committing mistakes, e.g. filling a text field with an incorrect value, or copying an incorrect text or cell in a spreadsheet. To reduce the impact of noise on routines identification, we preprocess the log. The preprocessing we apply consists

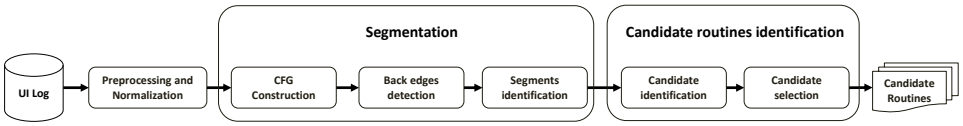

Fig. 1: Outline of the proposed approach

in identifying and removing redundant UIs that are clearly overwritten by successive UIs, such as the case of a double *CTRL+C* performed in sequence on different text fields. To identify such patterns of UIs, we rely on regular expressions by applying the methods described in [19].

After the preprocessing, the vast majority of UIs in the log are unique, because they differ by their payload. Even UIs capturing the same action within the same task execution (or different task executions) appear to be different. To discover each task execution (from start to end) recorded in the UI log, we need to detect all the UIs that, even having different payloads, correspond to the same action within the task execution.

To do so, we need to introduce the concepts of *UI data parameter*, *UI context parameter*, and *normalized UI*. Given a UI, its parameters can be divided into two types: *data parameter* and *context parameter*. The *data parameters* store the data values that are used during the execution of a task, e.g. the value of text fields or copied content. The *data parameters* usually have different values per task execution. By contrast, the *context parameters* capture where the UI was performed, e.g. the application and the location within the application. The *context parameters* are likely to have always the same values even in different task executions. For example, a UI of type *copy* includes the following parameters: *target-application* (e.g. the browser), *user*, *element id* (e.g. the id of a text field in the browser), *copied content*. Here, *target-application* and *element id* are context parameters, while *copied content* is a data parameter. Naturally, different type of UIs are characterized by different context parameters, e.g. a UI in a spreadsheet has the following context parameters: *spreadsheet name* and *cell location*.[1]

*Definition 3 (**Normalized UI**):* Given a UI $u = (t, \tau, P_\tau, Z, \phi)$, we define its normalization as $\bar{u} = (t, \tau, \bar{P}_\tau, \bar{Z}, \phi)$; where $\bar{Z} \subseteq Z$ contains only the values of the parameters in $\bar{P}$, where $\bar{P}$ is a set of context parameters. Given two normalized UIs, $u_1 = (t_1, \tau, P_\tau, \bar{Z}_1, \phi_1), u_2 = (t_2, \tau, P_\tau, \bar{Z}_2, \phi_2)$, the equality relation $u_1 = u_2$ holds iff $\forall p \in P_\tau \Rightarrow \phi_1(p) = \phi_2(p)$.

Given a UI log $\Sigma = \langle u_1, u_2, \ldots, u_n \rangle$, we normalize it by normalizing all the recorded UIs. The resulting normalized UI log is $\bar{\Sigma} = \langle \bar{u}_1, \bar{u}_2, \ldots, \bar{u}_n \rangle$. Table I and II show, respectively, a small fragment of a UI log and its normalized version. Intuitively, in a normalized UI log, the chances that two executions of the same routine have the same sequence (or set) of normalized UIs are high, because normalized UIs only have context parameters. In the next steps, we leverage such a characteristic of the normalized UI log to identify its segments (i.e. start and end of each executed task), and the routine(s) within the segments.

[1]The context parameters are selected by the domain expert.

	UI Timestamp	UI Type	UI Parameters and Values		
			P_1: Application	P_2: Element Label	P_3: Element Value
1	2019-03-03T19:02:18	Click button	Web	New Record	–
2	2019-03-03T19:02:23	Edit field	Web	Full Name	Albert Rauf
3	2019-03-03T19:02:27	Edit field	Web	Date	11-04-1986
4	2019-03-03T19:02:39	Edit field	Web	Phone	+ 61 043 512 4834
5	2019-03-03T19:02:47	Click button	Web	Submit	–
6	2019-03-03T19:02:58	Click button	Web	New Record	–
7	2019-03-03T19:03:13	Edit field	Web	Date	20-06-1987
8	2019-03-03T19:03:24	Edit field	Web	Phone	+61 519 790 1066
9	2019-03-03T19:03:43	Edit field	Web	Full Name	Audrey Backer
10	2019-03-03T19:04:10	Click button	Web	Submit	–

TABLE I: Fragment of a UI log

	UI Timestamp	UI Type	UI Parameters and Values	
			P_1: Application	P_2: Element Label
1	2019-03-03T19:02:18	Click button	Web	New Record
2	2019-03-03T19:02:23	Edit field	Web	Full Name
3	2019-03-03T19:02:27	Edit field	Web	Date
4	2019-03-03T19:02:39	Edit field	Web	Phone
5	2019-03-03T19:02:47	Click button	Web	Submit
6	2019-03-03T19:02:58	Click button	Web	New Record
7	2019-03-03T19:03:13	Edit field	Web	Date
8	2019-03-03T19:03:24	Edit field	Web	Phone
9	2019-03-03T19:03:43	Edit field	Web	Full Name
10	2019-03-03T19:04:10	Click button	Web	Submit

TABLE II: Normalized UI log

B. Segmentation

Before describing in detail the segmentation step, we formally define the necessary cornerstone concepts.

*Definition 4 (**Directly-follows relation**):* Let $\bar{\Sigma} = \langle \bar{u}_1, \bar{u}_2, \ldots, \bar{u}_n \rangle$ be a normalized UI log. Given two normalized UIs, $\bar{u}_x, \bar{u}_y \in \bar{\Sigma}$, we say that $\bar{u}_y$ directly-follows $\bar{u}_x$, i.e. $\bar{u}_x \rightsquigarrow \bar{u}_y$, iff $\bar{u}_{x|t} < \bar{u}_{y|t} \wedge \nexists \bar{u}_z \in \Sigma \mid \bar{u}_{x|t} \geq \bar{u}_{z|t} \geq \bar{u}_{y|t}$.

*Definition 5 (**Control-Flow Graph (CFG)**):* Given a normalized UI log, $\bar{\Sigma} = \langle \bar{u_1}, \bar{u_2}, \ldots, \bar{u_n} \rangle$, let $\bar{A}$ be the set of normalized UIs in $\bar{\Sigma}$. A Control-Flow Graph (CFG) is a tuple $G = (V, E, \hat{v}, \hat{e})$, where: V is the set of vertices of the graph, each vertex maps one normalized UI in $\bar{A}$; $E \subseteq V \times V$ is the set of edges of the graph, and each $(v_i, v_j) \in E$ represents a directly-follows relation between the UIs mapped by v_i and v_j; $\hat{v}$ is the graph *entry vertex*, such that $\forall v \in V \nexists (v, \hat{v}) \in E \wedge \nexists (\hat{v}, v) \in E$; while $\hat{e} = (\hat{v}, v_0)$ is the graph *entry edge*, such that v_0 maps $\bar{u_1}$. We note that $\hat{v} \notin V$, and $\hat{e} \notin E$, since they are artificial elements of the graph.

*Definition 6 (**CFG Path**):* Given a CFG $G = (V, E, \hat{v}, \hat{e})$, a CFG path is a sequence of vertices $p_{v_1, v_k} = \langle v_1, \ldots, v_k \rangle$ such that for each $i \in [1, k-1] \Rightarrow v_i \in V \cup \{\hat{v}\} \wedge \exists (v_i, v_{i+1}) \in E \cup \{\hat{e}\}$.

*Definition 7 (**Strongly Connected Component (SCC)**):* Given a graph $G = (V, E, \hat{v}, \hat{e})$, a strongly connected component (SCC) of G is a pair $\delta = (\bar{V}, \bar{E})$, where $\bar{V} = \{v_1, v_2, \ldots, v_m\} \subseteq V$ and $\bar{E} = \{e_1, e_2, \ldots, e_k\} \subseteq E$ such that $\forall v_i, v_j \in \bar{V} \exists p_{v_i, v_j} \mid \forall v \in p \Rightarrow v \in \bar{V}$. Given an SCC $\delta = (\bar{V}, \bar{E})$, we say that δ is

non-trivial iff $|\bar{V}| > 1$. Given a graph G, Δ_G denotes the set of all the non-trivial SCCs in G.

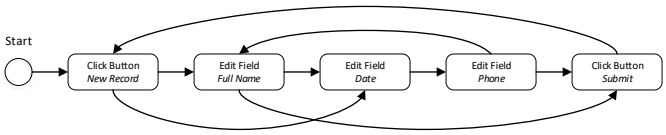

Fig. 2: Example of a Control-Flow Graph

The segmentation step starts with the construction of the CFG of the input normalized UI log. It is common that a CFG contains loops, since a loop represents the start of a new execution of the task recorded in the UI log. Indeed, in an ideal scenario, once a task execution ends with a certain UI (a vertex in the CFG), the next UI (i.e. the first UI of the next task execution) should already be mapped in the CFG and a loop will be generated. In such case, all the vertices contained in the loop represent the corresponding UIs that belong to the task. If several different tasks are recorded in the UI log, the graph would contain multiple disjoint loops, while if a task has repetitive subtasks there would be nested loops. Fig. 2 shows the CFG generated from the normalized UI log captured in Table II.

After the CFG is generated, we turn our attention to the identification of its back-edges, which can be detected by analysing the SCCs of the CFG, as described in Algorithm 1 and Algorithm 2. Given a CFG $G = (V, E, \hat{v}, \hat{e})$, we first build its dominator tree Θ (Algorithm 1, line 2), which captures domination relations between the vertices of the CFG. Fig. 3 shows the dominator tree of the CFG in Fig. 2.

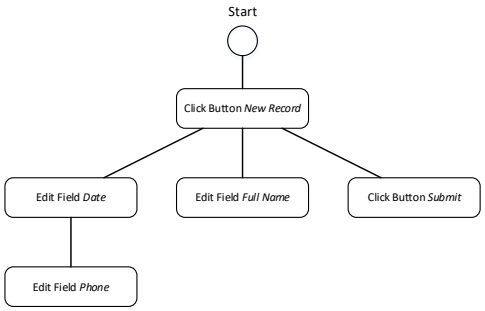

Fig. 3: Dominator tree

Then, we discover the set of all non-trivial SCCs (Δ_G) by applying the Kosaraju's algorithm [20] and removing the trivial SCCs (Algorithm 1, line 3). For each $\delta = (\bar{V}, \bar{E}) \in \Delta_G$, we discover its *header* using the dominator tree (Algorithm 2, line 1). The header of a δ is a special vertex $\hat{h} \in \bar{V}$, such that $\forall p_{\hat{v},v} \mid v \in \bar{V} \Rightarrow \hat{h} \in p_{\hat{v},v}$, i.e. the *header* $\hat{h}$ is the SCC vertex that dominates all the other SCC vertices. Once we have $\hat{h}$, we can identify the back-edges as $(v, \hat{h})$ with $v \in \bar{V}$ (line 3). Finally, the identified back-edges are stored and removed (lines 4 and 5) in order to look for nested SCCs and their back-edges by recursively executing Algorithm 2 (line 11), until no more SCCs and back-edges are found. However, if we detect an SCC that does not have a header vertex (i.e. the SCC is irreducible), we cannot identify the SCC back-edges. In such a case, we collect via a depth-first search of the CFG the edges $(v_x, v_y) \in \bar{E}$ such that v_y is topologically deeper than v_x - we call these edges *loop-edges* of the SCC (line 7). Then, out of all the loop-edges, we store (and remove from the SCC) the one having target and source connected by the longest *simple path* entirely contained within the SCC (lines 8 to 9).

Algorithm 1: Detect Back-edges

```
   input  : CFG G
   output : Back-edges Set B
1  B ← ∅;
2  Dominator Tree Θ ← computeDominatorTree(G);
3  Set Δ_G ← findSCCs(G);
4  foreach δ ∈ Δ_G do AnalyseSCC(δ, Θ, B);
5  return B;
```

Algorithm 2: Analyse SCC

```
   input  : SCC δ = (V̄, Ē), Dominator Tree Θ, Back-edges Set B
1  Header ĥ ← findHeader(δ, Θ);
2  if ĥ ≠ null then
3      Set I ← getIncomingEdges(δ, ĥ);
4      B ← B ∪ I;
5      Ē ← Ē \ I;
6  else
7      Set L ← findLoopEdges(δ);
8      Edge e ← getTheDeepestEdge(δ, L);
9      remove e from Ē;
10 Set Δ_δ ← findSCCs(δ);
11 foreach γ ∈ Δ_δ do AnalyseSCC(γ, Θ, B);
```

Given the CFG presented in Fig. 2 and its corresponding dominator tree (see Fig. 3), we immediately identify the SCC that consists of all the vertices except the *entry vertex*. Then, by applying Algorithm 2, we identify: the SCC header – *Click Button [New Record]*; and the only back-edge – (*Click Button [Submit]*, *Click Button [New Record]*), which we save and remove from the SCC. After the removal of this back-edge, we identify the nested SCC that contains all the three *Edit Field* UIs. Note that this second SCC does not have a header because it is irreducible, due to its multiple entries (*Edit Field [Full Name]* and *Edit Field [Date]*). However, by applying the depth-first search, we identify as candidate loop-edge for removal: (*Edit Field [Phone]*, *Edit Field [Full Name]*). After we remove this edge from the CFG, no SCCs are left so that Algorithm 2 stops its recursion.

At this point, we collected all the back-edges of the CFG, and we can leverage this information to start segmenting the UI log. We do so via Algorithm 3. First, we retrieve all the targets and sources of all the back-edges in the CFG and collect their corresponding UIs (lines 2 and 3). Each UI that is the target of a back-edge is an eligible segment starting point (hereinafter, *segment-start UI*), since a back-edge conceptually captures the end of a task execution, therefore its target represents the first UI of the next task execution. Following the same reasoning, each UI that is source of a back-edge is an eligible segment ending point (hereinafter, *segment-end UI*). Then, we sequentially scan all the UIs in the UI log (line 7). When we encounter a segment-start UI (line 9), and we are not already within a segment (line 10), we create a new segment (s, a list of UIs), we append the segment-start UI ($\bar{u}$), and we store it in order to match it with the correct segment-end UI (lines 11

Algorithm 3: Identify Segments

```
   input  : Normalised UI log Σ̄, Back-edges Set B
   output : Segments List Ψ
1  Set Ψ ← ∅;
2  Set T ← getTargets(B);
3  Set S ← getSources(B);
4  Boolean WithinSegment ← false;
5  Normalised UI u_0 ← null;
6  List s ← null;
7  for i ∈ [1, size(Σ̄)] do
8      Normalised UI ū ← getUI(Σ̄, i);
9      if ū ∈ T then
10         if WithinSegment = false then
11             s ← new List;
12             append ū to s;
13             u_0 ← ū;
14             WithinSegment ← true;
15         else
16             append ū to s;
17     else
18         if WithinSegment = true then
19             append ū to s;
20             if ū ∈ S ∧ (ū, u_0) ∈ B then
21                 add s to Ψ;
22                 WithinSegment ← false;
23 return Ψ;
```

to 14). Our strategy to detect segments in the UI log is driven by the following underlying assumption: a specific segment-end UI will be followed by the same segment-start UI, so that we can match segment-end and segment-start UIs exploiting back-edge's sources and targets (respectively). If the UI is not a segment-start (line 17), we check if we are within a segment (line 18) and, if not, we discard the UI, assuming that it is noise since it fell between the previous segment-end UI and the next segment-start UI. Otherwise, we append the UI to the current segment and we check if the UI is a segment-end matching the current segment-start UI (line 20). If that is the case, we reached the end of the segment and we add it to the set of segments (line 21), otherwise, we continue reading the segment.

Table III shows the segment-start and segment-end UIs, highlighted respectively in green and red, which also ideally delimits the two segments within the normalized UI log.

	UI Timestamp	UI Type	UI Parameters and Values	
			P_1: Application	P_2: Element Label
1	2019-03-03T19:02:18	Click button	Web	New Record
2	2019-03-03T19:02:23	Edit field	Web	Full Name
3	2019-03-03T19:02:27	Edit field	Web	Date
4	2019-03-03T19:02:39	Edit field	Web	Phone
5	2019-03-03T19:02:47	Click button	Web	Submit
6	2019-03-03T19:02:58	Click button	Web	New Record
7	2019-03-03T19:03:13	Edit field	Web	Date
8	2019-03-03T19:03:24	Edit field	Web	Phone
9	2019-03-03T19:03:43	Edit field	Web	Full Name
10	2019-03-03T19:04:10	Click button	Web	Submit

TABLE III: Segments identification

C. Candidate routines identification

The candidate routines identification step is based on the CloFast sequence mining algorithm [18]. To embed CloFast in our approach, we have to define the structure of the sequential patterns we want to identify. In this paper, we define a *sequential pattern* within a UI log as a sequence of normalized UIs occurring always in the same order in different segments, yet allowing gaps between the UIs belonging to the pattern. For example, if we consider the following three segments: $\langle u_1, u_y, u_2, u_3 \rangle$, $\langle u_1, u_2, u_x, u_3 \rangle$, and $\langle u_1, u_x, u_2, u_3 \rangle$; they all contain the same sequential pattern that is $\langle u_1, u_2, u_3 \rangle$. Furthermore, we define the *support* of a sequential pattern as the ratio of its occurrences and the total number of segments, and we refer to *closed* patterns and *frequent* patterns (relatively to an input threshold) as they are known in the literature. In particular, a frequent pattern is a pattern that appears in at least a number of sequences indicated by the threshold, while a closed pattern is a pattern that is not included in another pattern having exactly the same support. By applying CloFast to the set of the UI log segments, we discover all the *frequent closed* sequential patterns.

Some of these patterns may be *overlapping*, which (in this context) means they were discovered from the same portion of a segment and share some UIs. An example of overlapping patterns is the following, given three segments: $\langle u_1, u_y, u_2, u_3, u_x, u_4 \rangle$, $\langle u_1, u_y, u_2, u_x, u_3, u_4 \rangle$, and $\langle u_1, u_x, u_2, u_3, u_4 \rangle$; $\langle u_1, u_2, u_3, u_4 \rangle$ and $\langle u_1, u_x, u_4 \rangle$ are sequential patterns, but they overlap due to the shared UIs (u_1 and u_4). In practice, each UI belongs to only one routine, therefore, we are interested in discovering only non-overlapping patterns. For this purpose, we implemented an optimization that we use on top of CloFast. Given the set of patterns discovered by CloFast, we rank them by a pattern quality criterion (e.g. length, frequency), and we select the best pattern (i.e. the top rank one). Then, all its occurrences are removed from the segments, and we search again for frequent closed patterns performing the same procedure until there are no frequent closed patterns left.

In our approach, we integrated four pattern quality criteria to select the candidate routines: pattern frequency, pattern length, pattern coverage, and pattern cohesion score [3]. Pattern frequency considers how many times the pattern was observed in different segments. Pattern length considers the length of the patterns. Pattern coverage considers the percentage of the log that is covered by all the pattern occurrences. Pattern cohesion score considers the level of adjacency of the elements inside a pattern and is calculated as the difference between the pattern length and the median number of gaps between its elements. In other words, cohesion prioritizes the patterns whose UIs appear consecutively without (or with few) gaps while taking into account also the pattern length. In the next section, we compare these ranking criteria and discuss the benefits of using one or another.

IV. Evaluation

We implemented our approach as an open-source Java command-line application.[2] Our goal is threefold. First, we assess to what extent our approach can rediscover routines that are known to be recorded in the input UI logs. Second, we analyze how the use of different candidate routines selection criteria such as frequency and cohesion impact on the quality

[2]Available at https://github.com/volodymyrLeno/RPM_Segmentator

of the discovered routines. Last, we assess the efficiency and effectiveness of our approach when applied to real-life UI logs. Accordingly, we define the following research questions:

- **RQ1.** Does the approach rediscover routines that are known to exist in a UI log?
- **RQ2.** How do the candidate routines selection criteria affect the quality of the discovered routines?
- **RQ3.** Is the approach applicable in real-life settings, in terms of both efficiency and effectiveness?

A. Datasets

To answer our research questions, we rely on a dataset of thirteen UI logs, which can be divided into three subgroups: artificial logs, real-life logs recorded in a supervised environment, and real-life logs recorded in an unsupervised environment.[3] Table IV shows the logs characteristics.

UI Log	# Routine Variants	# Task Traces	# Actions	# Actions per trace (Avg.)
CPN1	1	100	1400	14.000
CPN2	3	1000	14804	14.804
CPN3	7	1000	14583	14.583
CPN4	4	100	1400	14.000
CPN5	36	1000	8775	8.775
CPN6	2	1000	9998	9.998
CPN7	14	1500	14950	9.967
CPN8	15	1500	17582	11.721
CPN9	38	2000	28358	14.179
Student Records (SR)	2	50	1539	30.780
Reimbursement (RT)	1	50	3114	62.280
Scholarships 1 (S1)		-	693	
Scholarships 2 (S2)		-	509	

TABLE IV: UI logs characteristics

The artificial logs (CPN1–CPN9) were generated from Colored Petri Nets (CPNs) [5]. The CPNs have increasing complexity, from low (CPN1) to high (CPN9). These logs are originally noise-free and segmented. We removed the segment identifiers to produce unsegmented logs.

The *Student Records* (SR) and *Reimbursement* (RT) logs record the simulation of real-life scenarios. The SR log simulates the task of transferring students' data from a spreadsheet to a Web form. The RT log simulates the task of filling reimbursement requests with data provided by a claimant. Each log contains fifty recordings of the corresponding task executed by one of the authors, who followed strict guidelines on how to perform the task. These logs contain little noise, which only accounts for user mistakes, such as filling the form with an incorrect value and performing additional actions to fix the mistake. For both logs, we know how the underlying task was executed, and we treat such information as ground truth when evaluating our approach. Additionally, we created four more logs (SRRT_{+}, RTSR_{+}, $\text{SRRT}_{\|}$, $\text{RTSR}_{\|}$) by combining SR and RT. SRRT_{+} and RTSR_{+} capture the scenario where the user first completes all the instances of one task and then moves to the other task. These logs were generated by concatenating SR and RT. $\text{SRRT}_{\|}$ and $\text{RTSR}_{\|}$ capture the scenario where the user is working simultaneously on two tasks. To simulate such behavior, we interleaved the segments of SR with those of RT.

[3]The real-life logs were recorded with the Action Logger tool [7]. All the logs are available at https://doi.org/10.6084/m9.figshare.12543587

Finally, the *Scholarships* logs (S1 and S2) were recorded by two employees of the University of Melbourne who performed the same task. The logs record the task of processing scholarship applications for international and domestic students. The task mainly consists of students data manipulation with transfers between spreadsheets and Web pages. Compared to the other logs, we have no a-priori knowledge of how to perform the task in the *Scholarships* logs (no ground truth). Also, when recording the UI logs, the University employees were not instructed to perform their task in a specific manner, i.e. they were left free to perform the task as they would normally do when unrecorded.

B. Setup

To answer RQ1 and RQ2, we analyzed the quality of the segmentation and that of the discovered routines, using the first 15 logs described above (CPN1 to CPN9, SR, RT, SRRT_{+}, RTSR_{+}, $\text{SRRT}_{\|}$, $\text{RTSR}_{\|}$) against the four candidate routines selection criteria in Section III-C, i.e. frequency, length, coverage and cohesion. To assess the quality of the segmentation, we use the *normalized* Levenshtein Edit Distance (LED), where a segment and its normalized UIs represent the string and its characters, respectively. Precisely, for each discovered segment, we collect all the ground truth segments that have at least one shared UI with the discovered segment, calculate the LED between the discovered segment and the ground truth segments and assign the minimum LED to the discovered segment as its quality score. Finally, we assess the overall quality of the segmentation as the average of the LEDs of each discovered segment.

The quality of the discovered routines is measured with the Jaccard Coefficient (JC), which captures the level of similarity between discovered and ground truth routines in a less strict manner compared to LED. In fact, the JC does not penalize the order of the UIs in a routine. This follows from the assumption that a routine could be executed performing some actions in different order, and the ordering should not be penalized. The JC between two routines is the ratio $\frac{n}{m}$, where n is the number of UIs that are contained in both routines, while m is the total number of UIs in each of the two routines (i.e. the sum of the lengths of the two routines). Given the set of discovered routines and the set of ground truth routines, for each discovered routine, we compute its JC with all the ground truth routines and assign the maximum JC to the discovered routine as its quality score. Finally, we assess the overall quality of the discovered routines as the average of the JC of each discovered routine. As the ground truth, we used the segments of the artificial logs and the guidelines given to the author who performed the tasks in SR and RT.

However, we cannot rely on the JC alone to assess the quality of the discovered routines, as this measure does not consider the routines we may have missed in the discovery. Thus, we also measure the total coverage to quantify how much log behavior is captured by the discovered routines. We would like to reach high coverage with as few routines as possible. Thus, we prioritize long routines over short ones by measuring the average routine length alongside the coverage.

To answer RQ3, we tested our approach on the S1 and S2 logs and qualitatively assessed the results with the help of the employees who performed the task. Specifically, we asked them to compare the rediscovered routines and the actions (i.e. UIs) they performed while recording.

All experiments were conducted on a Windows 10 laptop with an Intel Core i5-5200U CPU 2.20 GHz and 16GB RAM.

C. Results

Table V shows the results of the segmentation. As we can see, the LED for all the CPN logs is 0.0, highlighting that all the segments were discovered correctly. On the other hand, the segments discovered from the SR, RT, $SRRT_{+}$, $RTSR_{+}$, $SRRT_{\|}$, and $RTSR_{\|}$ logs slightly differ from the original ones. The main difference between the CPN logs and those recorded in a controlled environment is that the former contain routines having always the same starting UI, while the latter contain routines with several different starting UIs.

We identified the correct number of segments from all the logs except $SRRT_{\|}$ and $RTSR_{\|}$, where we could not discern the ending UI of the routine belonging to one task and the starting UI of the routine belonging to the other task, consequently merging the two routines and discovering only half of the total number of segments (50 out of 100). From the table we can also see that the time performance of our approach is reasonable, with maximum execution time of 3.6 seconds.

UI Log	# Original Segments	# Discovered Segments	LED (avg)	Exec. Time
CPN1	100	100	0.000	0.571
CPN2	1000	1000	0.000	1.705
CPN3	1000	1000	0.000	0.835
CPN4	100	100	0.000	0.461
CPN5	1000	1000	0.000	1.025
CPN6	1000	1000	0.000	0.707
CPN7	1500	1500	0.000	1.566
CPN8	1500	1500	0.000	1.596
CPN9	2000	2000	0.000	3.649
SR	50	50	0.059	0.714
RT	50	50	0.095	1.662
$SRRT_{+}$	100	100	0.078	2.424
$RTSR_{+}$	100	100	0.078	2.221
$SRRT_{\|}$	100	50	0.331	2.296
$RTSR_{\|}$	100	50	0.331	2.536

TABLE V: Segmentation results

Table VI shows the quality of the discovered routines for each selection criterion, when setting 0.1 as minimum support threshold of CloFast. The results highlight that, overall, the routines with the highest JC and the longest length are those discovered using cohesion as selection criterion, followed closely by those discovered using length as the criterion. Even though using these criteria we do not always achieve the highest coverage, the coverage scores are very high, i.e. above 0.90 for all the logs except CPN5.

Following these results, we decided to use cohesion as the selection criterion to discover the routines from the *scholarships* logs. From the S1 log we discovered five routines variants. The first routine variant consists in manually adding graduate research student applications to the student record in the information system of the university. The application is then assessed, and the student is notified about the outcome. The second routine variant consists in lodging a ticket to verify possible duplicate applications. When a new application is entered in the information system and its data matches an existing application, the new application is temporarily put on hold, and the employee fills in and lodges a ticket to investigate the duplicate. The remaining three routine variants represent exceptional cases, where the employee executed either the first or the second variant in a different manner (i.e. by altering the order of the actions or with overlapping routines executions). To assess the results, we showed the discovered routine variant to the employee of the University of Melbourne who recorded the S1 log, and they confirmed that the discovered routines correctly capture their task executions. Also, they confirmed that the last three routine variants are alternative executions of the first routine variant.[4]

While the results from the S1 log were positive, our

UI Logs	Selection Criterion	# Discovered Routines	Routine Length	Total Coverage	JC	Exec. Time
CPN1	Frequency	1	14.00	1.00	1.000	2.643
	Length	1	14.00	1.00	1.000	1.553
	Coverage	1	14.00	1.00	1.000	3.702
	Cohesion	1	14.00	1.00	1.000	**1.530**
CPN2	Frequency	3	6.33	**0.99**	0.452	3.908
	Length	2	**14.50**	0.95	**1.000**	4.789
	Coverage	2	14.00	**0.99**	0.964	**3.166**
	Cohesion	2	**14.50**	0.95	**1.000**	3.730
CPN3	Frequency	4	5.75	0.95	0.511	4.682
	Length	3	**14.33**	0.93	**1.000**	4.324
	Coverage	3	9.67	**0.96**	0.833	**3.940**
	Cohesion	3	**14.33**	0.93	**1.000**	6.237
CPN4	Frequency	1	12.00	0.86	0.857	3.452
	Length	2	**14.00**	**1.00**	**1.000**	**2.005**
	Coverage	1	13.00	0.93	0.929	3.351
	Cohesion	2	**14.00**	**1.00**	**1.000**	3.655
CPN5	Frequency	6	1.67	**0.86**	0.206	**6.418**
	Length	7	7.29	0.83	0.849	9.715
	Coverage	4	3.75	0.80	0.462	6.587
	Cohesion	8	**7.5**	**0.86**	**0.910**	18.206
CPN6	Frequency	3	4.67	1.00	0.485	4.250
	Length	2	**10.00**	1.00	**1.000**	2.924
	Coverage	3	4.67	1.00	0.485	**2.483**
	Cohesion	2	**10.00**	1.00	**1.000**	4.678
CPN7	Frequency	7	2.43	0.91	0.257	10.118
	Length	7	**9.57**	0.88	**0.986**	8.957
	Coverage	6	3.67	0.91	0.385	**7.203**
	Cohesion	7	9.43	**0.93**	0.971	11.983
CPN8	Frequency	5	4.20	0.75	0.337	11.801
	Length	6	**10.67**	**0.91**	**0.967**	9.070
	Coverage	5	7.60	0.89	0.618	**7.354**
	Cohesion	5	**10.67**	**0.91**	**0.967**	11.250
CPN9	Frequency	5	5.20	0.82	0.401	13.784
	Length	6	**14.67**	**0.95**	**1.000**	**8.265**
	Coverage	5	6.60	0.88	0.511	8.603
	Cohesion	6	**14.67**	**0.95**	**1.000**	13.943
SR	Frequency	3	10.00	0.96	0.356	3.883
	Length	3	**28.33**	**0.98**	**0.942**	**2.592**
	Coverage	2	15.50	0.96	0.532	2.635
	Cohesion	3	**28.33**	**0.98**	**0.942**	3.252
RT	Frequency	3	18.67	0.90	0.290	**4.63**
	Length	3	**56.33**	**0.96**	**0.829**	5.215
	Coverage	2	30.50	0.45	0.446	4.709
	Cohesion	3	**56.33**	**0.96**	**0.829**	6.585
$SRRT_{+}$	Frequency	5	16.80	0.90	0.374	13.826
	Length	4	**45.25**	**0.91**	**0.929**	**7.362**
	Coverage	2	42.50	0.86	0.921	8.177
	Cohesion	4	**45.25**	**0.91**	**0.929**	10.728
$RTSR_{+}$	Frequency	5	16.80	0.90	0.374	12.176
	Length	4	**45.25**	**0.91**	**0.929**	12.477
	Coverage	2	42.50	0.86	0.921	**9.085**
	Cohesion	4	**45.25**	**0.91**	**0.929**	14.675
$SRRT_{\|}$	Frequency	3	28.00	0.90	0.313	**13.905**
	Length	5	**86.40**	**0.96**	**0.580**	28.259
	Coverage	3	55.00	0.95	0.391	14.894
	Cohesion	5	**86.40**	**0.96**	**0.580**	37.277
$RTSR_{\|}$	Frequency	3	28.00	0.90	0.313	12.428
	Length	4	**89.50**	0.90	**0.600**	23.903
	Coverage	3	55.00	**0.95**	0.391	**10.838**
	Cohesion	4	**89.50**	0.90	**0.600**	38.657

TABLE VI: Quality of the discovered routines

[4]Detailed results at https://doi.org/10.6084/m9.figshare.12543587

approach could not discover any correct routine from the S2 log. By analyzing the results, we found out that the employee worked with multiple worksheets at the same time, frequently switching between them for visualization purposes only. Such behavior recorded in the log negatively affects the construction of the CFG and its domination tree, ultimately leading to the discovery of incorrect segments and routines. This also had an impact on the execution time, indeed, while it took only 41.7 seconds to discover the routines from the S1 log, it took 426.3 seconds to discover the routines from the S2 log.

D. Limitations

Our approach relies on information recorded in a UI log to identify segments and discover routines. Thus, its effectiveness is correlated with data quality. Since a UI log is fine-grained, deviations occurring during the routine execution affect the effectiveness of our approach. In our evaluation, we observed this phenomenon to varying degrees when dealing with real-life logs. In practice, the approach can identify correct routines only if they are observed frequently in the UI log. Recurring noise affects the accuracy of the results (see the S2 log).

The approach discovers multiple variants of the same routine when the UIs of a routine occur in different orders. Post-processing the results could be beneficial in order to cluster similar routines. Further, the approach is designed for logs that capture consecutive routine executions. In practice, routine instances may sometimes overlap (like in the S2 log).

Finally, while our approach is robust against routine executions with multiple ends, it is sensitive to multiple starts. Ideally, all routine executions should start with the same UI, unless different starts are recorded in batch (e.g. first only routines with a start, then routines with another start, etc.). In general, our approach can handle logs containing multiple different routines, provided that each routine does not share any UIs with other routines, except their start UIs.

V. Conclusion and Future Work

This paper presented an approach to automatically identify routines from unsegmented UI logs. The approach starts by decomposing the UI log into segments corresponding to paths within the connected components of a Control-Flow Graph derived from the log. Once the log is segmented, a noise-resilient sequential pattern mining technique is used to extract frequent patterns. The patterns are then ranked according to four quality criteria: frequency, length, coverage, and cohesion.

The approach has been implemented as an open-source tool and evaluated using synthetic and real-life logs. The evaluation shows that the approach can rediscover routines injected into a synthetic log, and that it discovers relevant routines in real-life logs. The execution times range from seconds to a few dozen seconds even for logs with tens of thousands of interactions.

As future work, we aim at addressing the limitations discussed above. We plan to add a post-processing step to group multiple routine variants and to discover an aggregated model thereof. This could be achieved by clustering the patterns and merging them into high-level models or by adapting a local process mining technique [21]. We plan to design more sophisticated segmentation techniques to better handle mixtures of multiple routines. Finally, the approach identifies routines from a control-flow perspective insofar as it manipulates sequences of interactions, without considering their data payload. Therefore, we plan to complement the proposed approach with an approach to quantify the automatability of candidate routines based on data attributes.

Acknowledgments. Work supported by the European Research Council (PIX project) and by the Australian Research Council (DP180102839).

References

[1] H. Leopold, H. van der Aa, and H. A. Reijers, "Identifying candidate tasks for robotic process automation in textual process descriptions," in *BPMDS*. Springer, 2018, pp. 67–81.

[2] V. Leno, A. Polyvyanyy, M. Dumas, M. La Rosa, and F. M. Maggi, "Robotic process mining: Vision and challenges," *Business & Information Systems Engineering*, 2020.

[3] H. Dev and Z. Liu, "Identifying frequent user tasks from application logs," in *IUI*. Springer, 2017, pp. 263–273.

[4] A. Jimenez-Ramirez, H. A. Reijers, I. Barba, and C. Del Valle, "A method to improve the early stages of the robotic process automation lifecycle," in *CAiSE*. Springer, 2019, pp. 446–461.

[5] A. Bosco, A. Augusto, M. Dumas, M. La Rosa, and G. Fortino, "Discovering automatable routines from user interaction logs," in *BPM Forum*. Springer, 2019.

[6] J. Gao, S. J. van Zelst, X. Lu, and W. M. van der Aalst, "Automated robotic process automation: A self-learning approach," in *OTM Confederated International Conferences*. Springer, 2019, pp. 95–112.

[7] V. Leno, A. Polyvyanyy, M. La Rosa, M. Dumas, and F. M. Maggi, "Action logger: Enabling process mining for robotic process automation," in *BPM Demos*. CEUR, 2019.

[8] C. Linn, P. Zimmermann, and D. Werth, "Desktop activity mining-a new level of detail in mining business processes," in *Workshops der INFORMATIK 2018-Architekturen, Prozesse, Sicherheit und Nachhaltigkeit*. Köllen Druck+ Verlag GmbH, 2018.

[9] H. Cao, N. Mamoulis, and D. W. Cheung, "Discovery of periodic patterns in spatiotemporal sequences," *IEEE Transactions on Knowledge and Data Engineering*, vol. 19, no. 4, pp. 453–467, 2007.

[10] Y. Zhu, M. Imamura, D. Nikovski, and E. Keogh, "Matrix profile vii: Time series chains: A new primitive for time series data mining," in *ICDM*. IEEE, 2017, pp. 695–704.

[11] M. Spiliopoulou, B. Mobasher, B. Berendt, and M. Nakagawa, "A framework for the evaluation of session reconstruction heuristics in web-usage analysis," *Informs journal on computing*, vol. 15, no. 2, pp. 171–190, 2003.

[12] D. Bayomie, A. Awad, and E. Ezat, "Correlating unlabeled events from cyclic business processes execution," in *CAiSE*. Springer, 2016, pp. 274–289.

[13] D. R. Ferreira and D. Gillblad, "Discovering process models from unlabelled event logs," in *BPM*. Springer, 2009, pp. 143–158.

[14] J. Han, H. Cheng, D. Xin, and X. Yan, "Frequent pattern mining: current status and future directions," *Data mining and knowledge discovery*, vol. 15, no. 1, pp. 55–86, 2007.

[15] S. D. Lee and L. De Raedt, "An efficient algorithm for mining string databases under constraints," in *International Workshop on Knowledge Discovery in Inductive Databases*. Springer, 2004, pp. 108–129.

[16] E. Ohlebusch and T. Beller, "Alphabet-independent algorithms for finding context-sensitive repeats in linear time," *Journal of Discrete Algorithms*, vol. 34, pp. 23–36, 2015.

[17] J. Wang and J. Han, "Bide: Efficient mining of frequent closed sequences," in *ICDE*. IEEE, 2004, pp. 79–90.

[18] F. Fumarola, P. F. Lanotte, M. Ceci, and D. Malerba, "Clofast: closed sequential pattern mining using sparse and vertical id-lists," *Knowledge and Information Systems*, vol. 48, no. 2, pp. 429–463, 2016.

[19] V. Leno, M. Dumas, M. La Rosa, F. M. Maggi, and A. Polyvyanyy, "Automated discovery of data transformations for robotic process automation," *ArXiv*, vol. abs/2001.01007, 2020.

[20] M. Sharir, "A strong-connectivity algorithm and its applications in data flow analysis," *Computers & Mathematics with Applications*, vol. 7, no. 1, pp. 67–72, 1981.

[21] B. Dalmas, N. Tax, and S. Norre, "Heuristic mining approaches for high-utility local process models," *T. Petri Nets and Other Models of Concurrency*, vol. 13, pp. 27–51, 2018.