Discovering Executable Routine Specifications From User Interaction Logs

Volodymyr Leno[a,b], Adriano Augusto[b], Marlon Dumas[a], Marcello La Rosa[b], Fabrizio Maria Maggi[c], Artem Polyvyanyy[b]

[a] *University of Tartu, Liivi 2, 50409, Tartu, Estonia*
[b] *University of Melbourne, Parkville, VIC, 3010, Australia*
[c] *Free University of Bozen-Bolzano, Piazza Universita, 1, 39100, Italy*

Abstract

Robotic Process Automation (RPA) is a technology to automate routine work such as copying data across applications or filling in document templates using data from multiple applications. RPA tools allow organizations to automate a wide range of routines. However, identifying and scoping routines that can be automated using RPA tools is time consuming. Manual identification of candidate routines via interviews, walk-throughs, or job shadowing allow analysts to identify the most visible routines, but these methods are not suitable when it comes to identifying the long tail of routines in an organization. This article proposes an approach to discover automatable routines from logs of user interactions with IT systems and to synthetize executable specifications for such routines. The approach starts by discovering frequent routines at a control-flow level (candidate routines). It then determines which of these candidate routines are automatable and it synthetizes an executable specification for each such routine. Finally, it identifies semantically equivalent routines so as to produce a set of non-redundant automatable routines. The article reports on an evaluation of the approach using a combination of synthetic and real-life logs. The evaluation results show that the approach can discover automatable routines that are known to be present in a UI log, and that it identifies automatable routines that users recognize as such in real-life logs.

Email addresses: leno@ut.ee (Volodymyr Leno), a.augusto@unimelb.edu.au (Adriano Augusto), marlon.dumas@ut.ee (Marlon Dumas), marcello.larosa@unimelb.edu.au (Marcello La Rosa), maggi@inf.unibz.it (Fabrizio Maria Maggi), artem.polyvyanyy@unimelb.edu.au (Artem Polyvyanyy)

Keywords: Robotic Process Automation, Robotic Process Mining, UI log

1. 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). Using 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 amenable to automation [1]. This can be achieved via interviews, walk-throughs, job shadowing, or by examining documented procedures [1]. These approaches are not always cost-efficient in large organizations, as routines tend to be scattered across the process landscape.

To tackle this gap, several research studies have proposed techniques to analyze User Interaction (UI) logs in order to discover repetitive routines that are amenable to automation via RPA [2, 3, 4, 5, 6]. However, existing approaches in this space make various assumptions that limit their applicability.

First, all of the existing approaches for discovering frequent and/or automatable routines from UI logs assume that the UI log consists of a set of traces (segments) of a task that is presupposed to contain one or more routines. In practice, however, 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.

Second, most of the existing approaches [2, 3, 4] discover frequent routines and/or automatable routines, but they do not produce an executable routine specification.

Third, existing approaches do not take into account the fact that the same routine may be performed differently (albeit equivalently) by different workers, or sometimes even by the same worker. In other words, existing approaches may produce redudant routines as output.

This article addresses these gaps by presenting an approach to discover automatable routines from unsegmented UI logs. The approach splits the unsegmented UI log into a set of segments, each representing a sequence of steps that appears frequently in the unsegmented UI log. It then applies sequential pattern mining techniques to find candidate routines for automation and evaluates their automatability. For each automatable routine, the approach synthesizes an executable routine specification, which can be compiled into an RPA bot. This bot

can then be executed by an RPA tool to replicate the underlying routine automatically.

The proposed approach has been implemented as an open-source prototype called Robidium [7]. Using this implementation, we have evaluated the proposed approach on synthetic and real-life UI logs in terms of its execution times and its ability to accurately discover routines from an UI log.

This article is an extended and revised version of a conference paper [8]. The conference version focused on the discovery of frequently repeated routines from unsegmented UI logs (i.e. candidate routines). This article extends this initial approach in two ways. First, this article presents an approach to post-process the identified candidate routines in order to assess their automatability and, in case a routine is fully automatable, to generate an executable routine specification. Second, this article proposes a method to identify semantically equivalent routines, so as to produce a non-redundant set of automable routines.

This article provides a concrete realization of a high-level architecture for discovering automatable routines from UI logs, sketched in [9]. To this end, the article proposes concrete techniques to implement each of the building blocks in [9], except for the UI log recording step, which is documented in [10].

The article is structured as follows. Section 2 provides an overview of related work. Section 3 describes the approach, while Section 4 reports the results of the evaluation. Finally, Section 5 concludes the paper and discusses the directions for future work.

2. Related work

The problem addressed by this article is denominated as Robotic Process Mining (RPM) in [9]. 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 [10]. The recorded data is represented as a UI log – a sequence of user interactions (herein called UIs), such as selecting a cell in a spreadsheet 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 (segmentation). The discovered segments are then scanned to identify routines that occur frequently across these segments. Finally, the resulting frequent routines (a.k.a. candidate routines) are analyzed in order to identify those that are automatable and to derive executable routine specifications.

In this section, we review previous research related to the three core research challenges of RPM identified in [9]: UI log segmentation, discovery of frequent (candidate) routines and discovery of automatable routines.

2.1. 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 materialize. Instead, it is reasonable to assume that an employee performing X-times the same task would make 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 [11, 12], 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 [13], 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 page navigation) [13]. Methods for session reconstruction are usually based on heuristics that rely on structural organization of web sites or time intervals between events. The former approach covers only the cases when all the user interactions are performed in the web applications, 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 [14] or that the underlying process is acyclic [15]. Both of these assumptions are unrealistic in our setting: a process model is not available since we are precisely trying to identify the routines in the log, and a routine may contain repetition.

Recent work on UI log segmentation [16] proposes to use trace alignment between the logs and the corresponding interaction models to identify the segments. In practice, however, such interaction models are not available beforehand. In this article, we outline a segmentation approach that does not require any models as inputs nor does it require that the user specifies one or more explicit delimiters between segments (e.g. that the user specifies that a given symbol X represents the start and/or the end of a segment).

2.2. Frequent Routine Discovery

Dev and Liu [17] have noted that the problem of routine identification from (segmented) UI logs can be mapped to that of frequent pattern mining, a well-known problem in the field of data mining [18]. Indeed, the goal of routine 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 [19, 20] (hence vulnerable to noise), and those that allow frequent patterns to have gaps within the sequence of symbols [21, 22] (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 [20]. 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. While such approaches discover the frequently repeated routines, they do not analyze whether they are automatable. In other words, these approaches focus on the discovery of the control-flow models instead of executable specifications.

The identification of frequent routines from sequences of actions is related to the problem of Automated Process Discovery (APD) [23], which has been studied in the field of process mining. Recent works [24, 2] show that RPA can benefit from process mining. In particular, the work in [2] proposes to apply traditional APD techniques to discover process models of routines captured in UI logs. However, traditional APD techniques discover control-flow models, while, in the context of RPA, we seek to discover executable specifications that capture the mapping between the outputs and the inputs of the actions performed during a routine.

2.3. Discovery of Automatable Routines

The discovery of automatable sequences of user interactions has been widely studied in the context of Web form and table auto-completion. For example, Excel's Flash Fill feature detects string patterns in the values of the cells in a spreadsheet and uses these patterns for auto-completion [25]. However, auto-completion techniques focus on identifying repetitions of keystrokes (sequences of characters). In this article, we look at routines that involve transfering data across fields in one or more applications as well as editing field values.

The discovery of data transfer routines that are amenable for RPA automation has been addressed in [3]. This latter paper proposes a technique to discover sequences of actions such that the inputs of each action in the sequence (except the first one) can be derived from the data observed in previous actions. However, this technique can only discover perfectly sequential routines, and is hence not resilient to variability in the order of the actions, whereas in reality, different users may perform the actions in a routine in a different order.

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

In [6] the authors present an approach to automatically discover routines from UI logs and automate them in the form of scripts. This approach, however, assumes that all the actions within a routine are automatable. In practice, it is possible that some actions have to be performed manually, and they can not be automated.

The approach presented in [4] aims at extracting rules from segmented UI logs that can be used to fill in forms automatically. However, this approach only discovers branching conditions that specify whether a certain activity has to be performed or not (e.g., check the box of the form). It focuses only on the copy-paste operations and does not identify more complex manipulations.

In previous work [5], we mapped the problem of discovering routines related to the data transferring to the problem of discovering data transformations. In this paper, we reuse this idea and extend it to tackle the problem of assessing if and to what extent a frequent (candidate) routine is automatable, and if such, producing an executable specification.

3. Approach

In this section, we describe our approach for discovering executable routine specifications from User Interaction (UI) logs. We adhere to the RPM pipeline proposed by Leno et al. [9], which we implemented in five macro steps (see Figure 1): i) *preprocessing and normalization*; ii) *segmentation*; iii) *candidate routine identification*; iv) *automatability assessment*; v) *routines aggregation*.

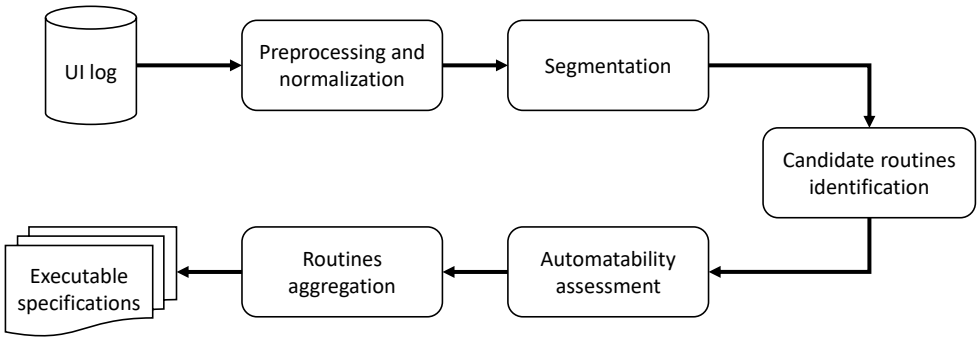


Figure 1: Outline of the proposed approach

Our approach takes as input a UI log, which is a chronologically ordered sequence of UIs between a worker and computer-based applications. In this paper, we assume that the applications used by the worker are either spreadsheet management applications or web browser applications. A UI log is usually recorded during the execution of the worker's daily tasks using specialized logging tools, for example, the *Action Logger* tool [10].

An example of a UI log is provided in Table 1. Each row of Table 1 captures one UI (e.g., clicking a button or copying the content of a cell). Each UI is characterized by a *timestamp*, a *type*, and a set of *parameters*, or *payload* (e.g., application, button's label and value of a field). The payload of a UI is not standardized, and depends on the UI type and application. Consequently, the UIs recorded in the same log may have different payloads. For example, the payload of UIs performed within a spreadsheet contains information regarding the spreadsheet name and the location of the target cell (e.g., cell row and column). In contrast, the payload of the UIs performed in a web browser contains information regarding the webpage URL, the name and identifier of the UI's target HTML element and its value (if any); – see Table 1 rows 1 and 2.

Row	UI Timestamp	UI Type	Payload					
			P_1	P_2	P_3	P_4	P_5	P_6
1	2019-03-03T19:02:18	Click button (Web)	https://unimelb.edu.au	New Record	newRecord	button	–	–
2	2019-03-03T19:02:20	Select cell (Excel)	StudentRecords	Sheet1	A	2	Albert Rauf	–
3	2019-03-03T19:02:23	Copy cell (Excel)	StudentRecords	Sheet1	A	2	Albert Rauf	Albert Rauf
4	2019-03-03T19:02:25	Select field (Web)	https://unimelb.edu.au	Full Name	name	“”	–	–
5	2019-03-03T19:02:26	Paste (Web)	https://unimelb.edu.au	Full Name	name	“”	Albert Rauf	–
6	2019-03-03T19:02:28	Edit field (Web)	https://unimelb.edu.au	Full Name	name	text	Albert Rauf	–
7	2019-03-03T19:02:30	Select cell (Excel)	StudentRecords	Sheet1	B	2	11/04/1986	–
8	2019-03-03T19:02:31	Copy cell (Excel)	StudentRecords	Sheet1	B	2	11/04/1986	11/04/1986
9	2019-03-03T19:02:34	Select field (Web)	https://unimelb.edu.au	Date	date	“”	–	–
10	2019-03-03T19:02:37	Paste (Web)	https://unimelb.edu.au	Date	date	“”	11/04/1986	–
11	2019-03-03T19:02:40	Edit field (Web)	https://unimelb.edu.au	Date	date	text	11-04-1986	–
12	2019-03-03T19:07:30	Select cell (Excel)	StudentRecords	Sheet1	C	2	+61 043 512 4834	–
13	2019-03-03T19:07:33	Copy cell (Excel)	StudentRecords	Sheet1	C	2	+61 043 512 4834	+61 043 512 4834
14	2019-03-03T19:07:40	Select field (Web)	https://unimelb.edu.au	Phone	phone	“”	–	–
15	2019-03-03T19:07:46	Paste (Web)	https://unimelb.edu.au	Phone	phone	“”	+61 043 512 4834	–
16	2019-03-03T19:07:48	Edit field (Web)	https://unimelb.edu.au	Phone	phone	text	043-512-4834	–
17	2019-03-03T19:07:50	Select cell (Excel)	StudentRecords	Sheet1	D	2	Germany	–
18	2019-03-03T19:07:52	Copy cell (Excel)	StudentRecords	Sheet1	D	2	Germany	Germany
19	2019-03-03T19:07:55	Select field (Web)	https://unimelb.edu.au	Country of residence	country	“”	–	–
20	2019-03-03T19:07:57	Paste (Web)	https://unimelb.edu.au	Country of residence	country	“”	Germany	–
21	2019-03-03T19:07:59	Edit field (Web)	https://unimelb.edu.au	Country of residence	country	text	Germany	–
22	2019-03-03T19:08:02	Edit field (Web)	https://unimelb.edu.au	Student status	status	select	Domestic	–
23	2019-03-03T19:08:05	Edit field (Web)	https://unimelb.edu.au	Student status	status	select	International	–
24	2019-03-03T19:08:08	Click button (Web)	https://unimelb.edu.au	Submit	submit	submit	–	–
25	2019-03-03T19:08:12	Click button (Web)	https://unimelb.edu.au	New Record	newRecord	button	–	–
26	2019-03-03T19:08:15	Select cell (Excel)	StudentRecords	Sheet1	B	3	20/06/1987	–
27	2019-03-03T19:08:18	Copy cell (Excel)	StudentRecords	Sheet1	B	3	20/06/1987	20/06/1987
28	2019-03-03T19:08:21	Select field (Web)	https://unimelb.edu.au	Date	date	“”	–	–
29	2019-03-03T19:08:26	Paste (Web)	https://unimelb.edu.au	Date	date	“”	20/06/1987	–
30	2019-03-03T19:08:28	Edit field (Web)	https://unimelb.edu.au	Date	date	text	20-06-1987	–
31	2019-03-03T19:08:32	Select cell (Excel)	StudentRecords	Sheet1	C	3	+61 519 790 1066	–
32	2019-03-03T19:08:34	Copy cell (Excel)	StudentRecords	Sheet1	C	3	+61 519 790 1066	+61 519 790 1066
33	2019-03-03T19:08:36	Select field (Web)	https://unimelb.edu.au	Phone	phone	“”	–	–
34	2019-03-03T19:08:38	Paste (Web)	https://unimelb.edu.au	Phone	phone	“”	+61 519 790 1066	–
35	2019-03-03T19:08:39	Edit field (Web)	https://unimelb.edu.au	Phone	phone	text	519-790-1066	–
36	2019-03-03T19:08:40	Select cell (Excel)	StudentRecords	Sheet1	A	3	Audrey Backer	–
37	2019-03-03T19:08:41	Copy cell (Excel)	StudentRecords	Sheet1	A	3	Audrey Backer	Audrey Backer
38	2019-03-03T19:08:42	Select field (Web)	https://unimelb.edu.au	Full Name	name	“”	–	–
39	2019-03-03T19:08:44	Paste (Web)	https://unimelb.edu.au	Full Name	name	“”	Audrey Backer	–
40	2019-03-03T19:08:46	Edit field (Web)	https://unimelb.edu.au	Full Name	name	text	Audrey Backer	–
41	2019-03-03T19:08:50	Select cell (Excel)	StudentRecords	Sheet1	D	2	Germany	–
42	2019-03-03T19:08:52	Copy cell (Excel)	StudentRecords	Sheet1	D	2	Germany	Germany
43	2019-03-03T19:08:58	Select cell (Excel)	StudentRecords	Sheet1	D	3	Australia	–
44	2019-03-03T19:09:01	Copy cell (Excel)	StudentRecords	Sheet1	D	3	Australia	Australia
45	2019-03-03T19:09:05	Select field (Web)	https://unimelb.edu.au	Country of residence	country	“”	–	–
46	2019-03-03T19:09:08	Paste (Web)	https://unimelb.edu.au	Country of residence	country	“”	Australia	–
47	2019-03-03T19:09:10	Edit field (Web)	https://unimelb.edu.au	Country of residence	country	text	Australia	–
48	2019-03-03T19:09:14	Edit field (Web)	https://unimelb.edu.au	Student status	status	select	Domestic	–
49	2019-03-03T19:09:20	Click button (Web)	https://unimelb.edu.au	Submit	submit	submit	–	–
…	…	…	…	…	…	…	…	…

Table 1: Fragment of a user interaction log

Our approach analyzes the log to identify and output a collection of *executable routine specifications*. Each routine specification is a pair (c, Λ), where c is a sequence of UIs, or a *candidate routine*, and Λ is a set of *data transformation steps*. Each *data transformation step* is a triplet that specifies: i) variables from which the data was read, ii) variables to which the data was written, and iii) a function capturing the data transformation (if any occurs). Such routine specifications can be compiled into software bots that can be deployed on a tool like UiPath, [1] which would be able to automatically replicate the routine.

In the following, we describe step-by-step how we generate a collection of executable routine specifications from an input UI log.

3.1. Preprocessing and Normalization

Before diving into the details of this step, we formally define the concepts of a *user interaction* and *user interaction log*, which we will refer to throughout this and the following sections.

Definition 1 (User interaction (UI)). *A* user interaction (UI) *is a tuple* $u = (t, \tau, P_\tau, Z, \phi)$*, where:* t *is a timestamp;* τ *is a UI type;* P_τ *is a set of parameters, or* payload*;* Z *is a set of parameter values; and* $\phi : P_\tau \rightarrow Z$ *is a value assignment function.*

Table 2 shows UIs and their associated payloads recorded by the Action Logger tool [10]. The UIs are logically grouped, based on their type, into three groups: *navigation*; *read*; and *write* UIs. We assume that every UI is an *instantiation* of one of the UI types from Table 2, with every parameter assigned with a specific value.

Definition 2 (User interaction log). *A user interaction log* Σ *is a sequence of UIs* $\Sigma = \langle u_1, u_2, \ldots, u_n \rangle$*, ordered by their timestamps, i.e.,* $u_{i|t} < u_{j|t}$ *for any* $i, jsuchthat 1 \leq i < j \leq n$*.*

Ideally, UIs recorded in a log should only relate to the execution of the task(s) of interest. However, in practice, a log often also contains UIs that do not contribute to completing the recorded task(s). We can consider such UIs to be *noise*. Examples of noise UIs include a worker browsing the web (e.g., social networking) while executing a task that does not require to do that, or a worker committing

[1] A commercial tool available at www.uipath.com

UI Group	UI Type	Parameter Names					
		P1	P2	P3	P4	P5	P6
Navigation	Create New Tab (Web)	ID					
	Select Tab (Web)	URL	ID	Title			
	Close Tab (Web)	URL	ID	Title			
	Navigate To (Web)	URL					
	Add Worksheet (Excel)	Workbook name	Worksheet name				
	Select Worksheet (Excel)	Workbook name	Worksheet name				
	Select Cell (Excel)	Workbook name	Worksheet name	Cell column	Cell row	Value	
	Select Range (Excel)	Workbook name	Worksheet name	Range columns	Range rows	Value	
	Select Field (Web)	URL	Name	ID	Value		
Read	Copy (Web)	URL	Name	ID	Value	Copied content	
	Copy Cell (Excel)	Workbook name	Worksheet name	Cell column	Cell row	Value	Copied content
	Copy Range (Excel)	Workbook name	Worksheet name	Range columns	Range rows	Value	Copied content
Write	Paste Into Cell (Excel)	Workbook name	Worksheet name	Cell column	Cell row	Value	Pasted content
	Paste Into Range (Excel)	Workbook name	Worksheet name	Range columns	Range rows	Value	Pasted content
	Paste (Web)	URL	Name	ID	Value	Pasted content	
	Click Button (Web)	URL	Name	ID	Type		
	Click Link (Web)	URL	Inner text	Href			
	Edit Field (Web)	URL	Name	ID	Type	Value	
	Edit Cell (Excel)	Workbook name	Worksheet name	Cell column	Cell row	Value	
	Edit Range (Excel)	Workbook name	Worksheet name	Range columns	Range rows	Value	

Table 2: User interaction types and their parameters

mistakes (e.g., filling a text field with an incorrect value or copying a wrong cell of a spreadsheet). While we cannot detect the former kind of noise without a context-aware noise filter, we can identify the latter type of noise. Given that noise in a log may negatively affect the segmentation step, we attempt to remove it. Specifically, the filter we implemented removes UIs whose effects are overwritten by subsequent UIs, and certain navigation UIs that a software robot would not need to replicate. To identify and remove such UIs, we rely on three search-and-replace rules defined as regular expressions that operate as follows.

1. Remove UIs of type *select cell*, *select range*, *select field* (e.g., Table 1, rows 2, 4, 7);

2. Remove UIs of type *copy* that are not eventually followed by UI of type *paste* before another UI of type *copy* occurs (e.g., Table 1, row 42);

3. Remove UIs of type *edit cell*, *edit range*, and *edit field* that are followed by another UI of the same type that targets the same cell or field and overwrites its content before a UI of type *copy* occurs (e.g., Table 1, row 22).

We note that, given an unsegmented log, it is impossible to apply the third rule straightforward, as removing the first UI of type *edit* (considered redundant) may be an error if the second UI of type *edit* belongs to a successive task execution. Therefore, we postpone the application of the third rule after the segmentation step. The filtering rules are applied recursively on the log until no more UIs are removed and the log is assumed to be free of *detectable* noise. Devising and applying more sophisticated noise filtering algorithms would probably benefit the

approach presented in this study. However, the design of such algorithms is outside the scope of this paper, and we leave it as possible future work.

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

Given a UI, its payload can be divided into *data parameters* and *context parameters*. The former store the data values used during the execution of tasks, e.g., the value of text fields or copied content. Consequently, *data parameters* usually have different values in different task executions. In contrast, the latter capture the context in which UIs were performed, e.g., the application and the location within the application. Therefore, *context parameters* of the same UI within a task are likely to have the same values across different task executions. For example, the payload of a UI of type *copy cell* has the following parameters (see also Table 2): *workbook name* (the Excel file name); *worksheet name* (within the Excel file); *cell column* (i.e., the column of the cell in the worksheet that was selected for the UI); *cell row* (i.e., the row of the cell in the worksheet that was selected for the UI); *value* (i.e., current value of the cell selected for the UI); *copied content* (the content copied as the result of the UI). Here, *workbook name*, *worksheet name*, *cell column/row* are *context parameters*, while *copied content* and *value* are *data parameters*. Different context parameters characterize different UI types. For example, a UI of type *click button* performed in a web browser has only these context parameters: *URL*; *name* (i.e., the label of the button); *ID* (of the button, as an element in the HTML page); and *type*. Often, context parameters are determined by the type of UI. To reduce the chance of possible automated misinterpretations, we allow the user to configure the context parameters of various UI types manually.

To segment an input UI log, we rely on the context parameters of the UIs. We call a UI whose payload has been reduced to its context parameters a *normalized UI*.

Definition 3 (Normalized UI). *Given a UI* $u = (t, \tau, P_\tau, Z, \phi)$*, the UI* $\bar{u} = (t, \tau, \bar{P}_\tau, \bar{Z}, \phi)$ *is its normalized version, where* $\bar{Z}$ *contains only the values of the parameters in* $\bar{P}_\tau$*, where* $\bar{P}_\tau$ *is a set of context parameters.*

Two normalized UIs $u_1 = (t_1, \tau, \bar{P}_\tau, \bar{Z}_1, \phi_1)$ and $u_2 = (t_2, \tau, \bar{P}_\tau, \bar{Z}_2, \phi_2)$ are *equivalent*, denoted by $u_1 = u_2$ iff $\forall p \in \bar{P}_\tau \Rightarrow \phi_1(p) = \phi_2(p)$.

A log in which all the UIs have been normalized is a *normalized log*, and we refer to it with the notation $\bar{\Sigma} = \langle \bar{u_1}, \bar{u_2}, \ldots, \bar{u_n} \rangle$. Table 1 and Table 3 show, respectively, a fragment of a log and its normalized version. Intuitively, in a normalized log, the chances that two executions of the same task have the same sequence (or set) of normalized UIs are high because they have only context parameters. We leverage such a characteristic of the normalized log to identify its segments (i.e., start and end of each executed task), and then the routine(s) within the segments.

Row	UI Timestamp	UI Type	Payload			
			P_1	P_2	P_3	P_4
1	2019-03-03T19:02:18	Click button (Web)	http://www.unimelb.edu.au	New Record	newRecord	button
2	2019-03-03T19:02:23	Copy cell (Excel)	StudentRecords	Sheet1	A	–
3	2019-03-03T19:02:26	Paste (Web)	http://www.unimelb.edu.au	Full Name	name	–
4	2019-03-03T19:02:28	Edit field (Web)	http://www.unimelb.edu.au	Full Name	name	text
5	2019-03-03T19:02:31	Copy cell (Excel)	StudentRecords	Sheet1	B	–
6	2019-03-03T19:02:37	Paste (Web)	http://www.unimelb.edu.au	Date	date	–
7	2019-03-03T19:02:40	Edit field (Web)	http://www.unimelb.edu.au	Date	date	text
8	2019-03-03T19:07:33	Copy cell (Excel)	StudentRecords	Sheet1	C	–
9	2019-03-03T19:07:40	Paste (Web)	http://www.unimelb.edu.au	Phone	phone	–
10	2019-03-03T19:07:48	Edit field (Web)	http://www.unimelb.edu.au	Phone	phone	text
11	2019-03-03T19:07:50	Copy cell (Excel)	StudentRecords	Sheet1	D	–
12	2019-03-03T19:07:55	Paste (Web)	http://www.unimelb.edu.au	Country of residence	country	–
13	2019-03-03T19:08:02	Edit field (Web)	http://www.unimelb.edu.au	Country of residence	country	text
14	2019-03-03T19:08:05	Edit field (Web)	http://www.unimelb.edu.au	Student status	status	select
15	2019-03-03T19:08:08	Click button (Web)	http://www.unimelb.edu.au	Submit	submit	submit
16	2019-03-03T19:08:12	Click button (Web)	http://www.unimelb.edu.au	New Record	newRecord	button
17	2019-03-03T19:08:17	Copy cell (Excel)	StudentRecords	Sheet1	B	–
18	2019-03-03T19:08:21	Paste (Web)	http://www.unimelb.edu.au	Date	date	–
19	2019-03-03T19:08:28	Edit field (Web)	http://www.unimelb.edu.au	Date	date	text
20	2019-03-03T19:08:35	Copy cell (Excel)	StudentRecords	Sheet1	C	–
21	2019-03-03T19:08:38	Paste (Web)	http://www.unimelb.edu.au	Phone	phone	–
22	2019-03-03T19:08:39	Edit field (Web)	http://www.unimelb.edu.au	Phone	phone	text
23	2019-03-03T19:08:40	Copy cell (Excel)	StudentRecords	Sheet1	A	–
24	2019-03-03T19:08:42	Paste (Web)	http://www.unimelb.edu.au	Full Name	name	–
25	2019-03-03T19:08:43	Edit field (Web)	http://www.unimelb.edu.au	Full Name	name	text
26	2019-03-03T19:08:45	Copy cell (Excel)	StudentRecords	Sheet1	D	–
27	2019-03-03T19:08:47	Paste (Web)	http://www.unimelb.edu.au	Country of residence	country	–
28	2019-03-03T19:08:49	Edit field (Web)	http://www.unimelb.edu.au	Country of residence	country	text
29	2019-03-03T19:08:52	Edit field (Web)	http://www.unimelb.edu.au	Student status	status	select
30	2019-03-03T19:08:53	Click button (Web)	http://www.unimelb.edu.au	Submit	submit	submit
…	…	…	…	…	…	…

Table 3: Normalized user interaction log after preprocessing

3.2. Segmentation

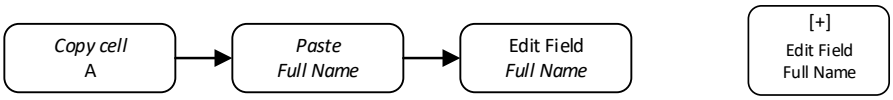
A log may capture long working sessions, where a worker performs multiple instances of one or more tasks. The next step of our approach decomposes the log into *segments* that identify the start and the end of each recorded task in the log. Given a normalized log, we generate its control-flow graph (CFG). A CFG is a graph where each vertex represents a different normalized UI, and each edge captures a directly-follows relation between the two normalized UIs represented by the source and the target vertices of the edge. A CFG has an explicit source vertex representing the first normalized UI recorded in the log.

Given a log, the directly follows relation on UI is defined as follows.

Definition 4 (Directly-follows relation). *Let $\bar{\Sigma} = \langle \bar{u}_1, \bar{u}_2, \ldots, \bar{u}_n \rangle$ be a normalized log. Given two 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 $\bar{u}_{x|t} < \bar{u}_{y|t} \wedge \nexists \bar{u}_z \in \bar{\Sigma} \mid \bar{u}_{x|t} \leq \bar{u}_{z|t} \leq \bar{u}_{y|t}$.*

Definition 5 (Control-Flow Graph (CFG)). *Given a normalized log, $\bar{\Sigma} = \langle \bar{u_1}, \bar{u_2}, \ldots, \bar{u_n} \rangle$, let $\bar{A}$ be the set of all the 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 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.*

It is likely that a CFG is cyclic, since a loop represents the start of a new execution of the task recorded in the 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 have already been mapped to a vertex of the CFG, and a loop will be generated. In such a case, all the vertices in the loop represent the UIs performed during the execution of the task. If several different tasks are recorded in sequence in the same log, we would observe several disjoint loops in the CFG, while if a task has repetitive subtasks, we would observe nested loops in the CFG. Figure 3 shows the CFG generated from the log captured in Table 3, we note that for simplicity we collapsed some vertices as shown in Figure 2.



(a) Before (b) After

Figure 2: Collapsed vertices in Figure 3

Once the CFG is generated, we turn our attention to identifying its back-edges (i.e., its loops). By identifying the CFG back-edges and their UIs, we extract the start and end UIs of the repeated task. These UIs are used to mark the boundaries between task executions. The back-edges of a CFG can be identified by analyzing the CFG Strongly Connected Components (SCCs). Given a graph, an SCC is a subgraph where for all its pairs of vertices, there exist a set of edges connecting the pair of vertices such that all the sources and targets of these edges belong to the subgraph.

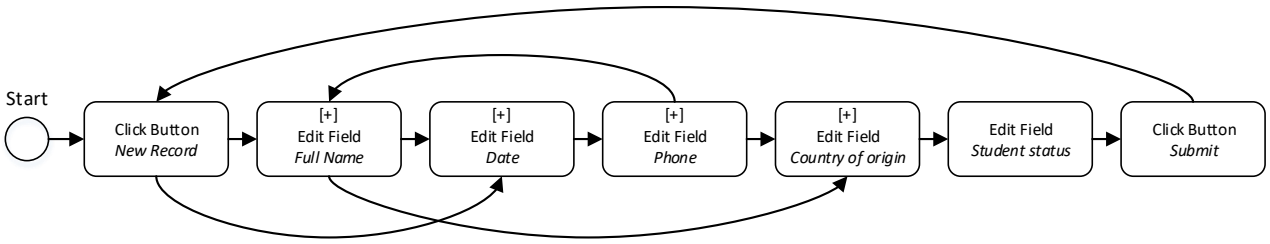


Figure 3: Example of a Control-Flow Graph

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.*

Algorithm 1 and Algorithm 2 describe how we identify the SCCs of the CFG. 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. Figure 4 shows the dominator tree of the CFG in Figure 3. Then, we discover the set of all non-trivial SCCs (Δ_G) by applying the Kosaraju's algorithm [26] 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 dominator tree 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* $\hat{h}$ (a.k.a. the SCC entry) 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 (formally, 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

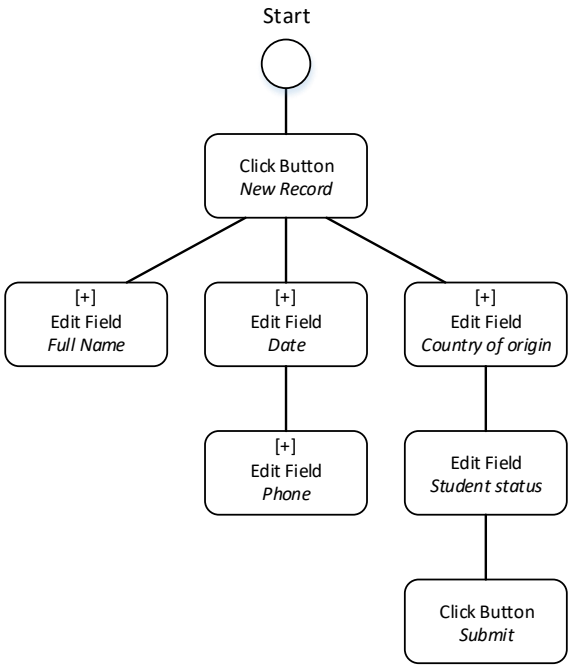


Figure 4: Dominator tree

source connected by the longest *simple path* entirely contained within the SCC (lines 8 to 9).

Given the CFG presented in Figure 3 and its corresponding dominator tree (see Figure 4), we 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 edits of *Full Name*, *Date*, and *Phone* fields. 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 Algorithm 2 terminates.

At this point, we collected all the back-edges of the CFG. Next, we use them to segment the log. We do so by applying 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 mapped onto a back-edge target is an eligible

Algorithm 1: Back-edges detection

input : CFG G
output : Back-edges Set B

$B \leftarrow \varnothing$;
Dominator Tree $\Theta \leftarrow$ computeDominatorTree(G);
Set $\Delta_G \leftarrow$ findSCCs(G);
foreach $\delta \in \Delta_G$ **do** AnalyseSCC(δ, Θ, B) ;
return B;

Algorithm 2: Analyse SCC

input : SCC $\delta = (\bar{V}, \bar{E})$, Dominator Tree Θ, Back-edges Set B

Header $\hat{h} \leftarrow$ findHeader(δ, Θ);
if $\hat{h} \neq$ *null* **then**
    Set $I \leftarrow$ getIncomingEdges(δ, $\hat{h}$);
    $B \leftarrow B \cup I$;
    $\bar{E} \leftarrow \bar{E} \setminus I$;
else
    Set $L \leftarrow$ findLoopEdges(δ);
    Edge $e \leftarrow$ getTheDeepestEdge(δ, L);
    remove e **from** $\bar{E}$;
Set $\Delta_\delta \leftarrow$ findSCCs(δ);
foreach $\gamma \in \Delta_\delta$ **do** AnalyseSCC(γ, Θ, B) ;

segment starting point (from now on, *segment-start UI*). A back-edge conceptually captures the end of a task execution, while its target represents the first UI of the next task execution. By applying the same reasoning, each UI mapped onto the source of a back-edge is an eligible segment ending point (hereinafter, *segment-end UI*). Then, we sequentially scan all the UIs in the log (line 7). When we encounter a segment-start UI (line 9), and we are not already within a segment (see 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 (line 11 to 14). Our strategy to detect segments in the 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 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 this 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.

Algorithm 3: Segmentation

```
input  : Normalized UI log Σ̄, Back-edges Set B
output : Segments List Ψ
Set Ψ ← ∅;
Set T ← getTargets(B);
Set S ← getSources(B);
Boolean WithinSegment ← false;
Normalized UI u_0 ← null;
Queue s ← ∅;
for i ← 1 to size(Σ̄) do
    Normalized UI ū ← getUI(Σ̄, i);
    if ū ∈ T then
        if WithinSegment = false then
            s ← ∅;
            append ū to s;
            u_0 ← ū;
            WithinSegment ← true;
        else
            append ū to s;
    else
        if WithinSegment = true then
            append ū to s;
            if ū ∈ S ∧ (ū, u_0) ∈ B then
                Ψ ← Ψ ∪ {s};
                WithinSegment ← false;
return Ψ;
```

Table 4 shows the segment-start and the segment-end UIs (highlighted in green and red, respectively), which delimits two segments within the normalized UI log of our running example (see also Table 3).

3.3. Candidates routines identification

Once the log has been segmented, we move to the identification of the candidate routines. The identification step is based on the CloFast sequence mining algorithm [22]. To integrate 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 always occurring 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 segments containing the pattern and the total number of segments. We refer to *closed*

Row	UI Timestamp	UI Type	Payload			
			P_1	P_2	P_3	P_4
1	2019-03-03T19:02:18	Click button (Web)	http://www.unimelb.edu.au	New Record	newRecord	button
2	2019-03-03T19:02:23	Copy cell (Excel)	StudentRecords	Sheet1	A	–
3	2019-03-03T19:02:26	Paste (Web)	http://www.unimelb.edu.au	Full Name	name	–
4	2019-03-03T19:02:28	Edit field (Web)	http://www.unimelb.edu.au	Full Name	name	text
5	2019-03-03T19:02:31	Copy cell (Excel)	StudentRecords	Sheet1	B	–
6	2019-03-03T19:02:37	Paste (Web)	http://www.unimelb.edu.au	Date	date	–
7	2019-03-03T19:02:40	Edit field (Web)	http://www.unimelb.edu.au	Date	date	text
8	2019-03-03T19:07:33	Copy cell (Excel)	StudentRecords	Sheet1	C	–
9	2019-03-03T19:07:40	Paste (Web)	http://www.unimelb.edu.au	Phone	phone	–
10	2019-03-03T19:07:48	Edit field (Web)	http://www.unimelb.edu.au	Phone	phone	text
11	2019-03-03T19:07:50	Copy cell (Excel)	StudentRecords	Sheet1	D	–
12	2019-03-03T19:07:55	Paste (Web)	http://www.unimelb.edu.au	Country of residence	country	–
13	2019-03-03T19:08:02	Edit field (Web)	http://www.unimelb.edu.au	Country of residence	country	text
14	2019-03-03T19:08:05	Edit field (Web)	http://www.unimelb.edu.au	Student status	status	select
15	2019-03-03T19:08:08	Click button (Web)	http://www.unimelb.edu.au	Submit	submit	submit
16	2019-03-03T19:08:12	Click button (Web)	http://www.unimelb.edu.au	New Record	newRecord	button
17	2019-03-03T19:08:17	Copy cell (Excel)	StudentRecords	Sheet1	B	–
18	2019-03-03T19:08:21	Paste (Web)	http://www.unimelb.edu.au	Date	date	–
19	2019-03-03T19:08:28	Edit field (Web)	http://www.unimelb.edu.au	Date	date	text
20	2019-03-03T19:08:35	Copy cell (Excel)	StudentRecords	Sheet1	C	–
21	2019-03-03T19:08:38	Paste (Web)	http://www.unimelb.edu.au	Phone	phone	–
22	2019-03-03T19:08:39	Edit field (Web)	http://www.unimelb.edu.au	Phone	phone	text
23	2019-03-03T19:08:40	Copy cell (Excel)	StudentRecords	Sheet1	A	–
24	2019-03-03T19:08:42	Paste (Web)	http://www.unimelb.edu.au	Full Name	name	–
25	2019-03-03T19:08:43	Edit field (Web)	http://www.unimelb.edu.au	Full Name	name	text
26	2019-03-03T19:08:45	Copy cell (Excel)	StudentRecords	Sheet1	D	–
27	2019-03-03T19:08:47	Paste (Web)	http://www.unimelb.edu.au	Country of residence	country	–
28	2019-03-03T19:08:49	Edit field (Web)	http://www.unimelb.edu.au	Country of residence	country	text
29	2019-03-03T19:08:52	Edit field (Web)	http://www.unimelb.edu.au	Student status	status	select
30	2019-03-03T19:08:53	Click button (Web)	http://www.unimelb.edu.au	Submit	submit	submit
…	…	…	…	…	…	…

Table 4: Segments identification

patterns and *frequent* patterns (relatively to an input threshold) as they are known in the literature. Specifically, a frequent pattern is a pattern that appears in at least a number of occurrences 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 log segments, we discover all the *frequent closed* sequential patterns.

Some of these patterns may be *overlapping*, which (in our context) means that they 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, and we select the best pattern (i.e., the top one in the ranking). We integrated four pattern quality criteria to select the candidate routines: pattern frequency, pattern length, pattern coverage, and pattern cohesion score [17]. 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. Finally, pattern cohesion score considers the level of adjacency of the elements inside a pattern. It 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.

For the candidate routine that we identified as the best pattern for a given quality criterion, we collect and remove all its occurrences from the log. An occurrence of a candidate routine is called a *routine instance*. Formally, a routine instance is a sequence of (non-normalized) UIs, e.g., $r = \langle u_1, u_2, u_3, u_4 \rangle$. After the removal of all the instances of the best candidate routine from the log, we repeat this identification step until no more candidate routines are identified. At the completion of this step, we obtain a set of candidate routines, referred to as $\mathscr{C}_\Sigma$, such that, for each candidate routine $c_i \in \mathscr{C}_\Sigma$, we can retrieve the set of its routine instances, referred to as $\mathscr{R}_{c_i}$.

Considering our running example, with reference to Table 4, assuming that the two routine instances that we identified in the previous step (by detecting their segment-start and segment-end UIs) frequently occur in the original log (a snapshot of which is captured in Table 1), and choosing length as a selection criterion, at the end of this step, we would discover two candidate routines, each consisting of 15 normalized UIs (as shown in Table 4). An example of a routine instance for each of the two candidate routines can be easily observed in the original log, Table 1 rows 1 to 24 and 25 to 49 (excluding the UIs filtered in the first step of our approach).

3.4. Automatability assessment

The candidate routines in $\mathscr{C}_\Sigma$ (and their instances, $\mathscr{R}_{c_i}$) that we identified in the previous step represent behavior recorded in the log that frequently repeats itself, thus it is the candidate for automation. However, the fact that a routine is frequently observed in a log is not a sufficient condition to guarantee its automatability. Let us consider the following example; a worker fills in and submits 100 times the same web-form, doing it always with the same sequence of actions but inputting manually-generated data (e.g., received over a phone call or copied from a hard-copy document). In such a scenario, although we would identify the filling and submission of the web-form as a candidate routine, we would not be able to automate it because we cannot automatically generate the data in input to the web-forms. On the other hand, if the data in input to the web-forms was copied from

another digital document, for example a spreadsheet, we could probably automate the routine.

Considering such a context, the next step of our approach is to assess the degree of automatability of the discovered candidate routines. To do so, given a candidate routine $c_i \in \mathscr{C}_\Sigma$, we check whether all its UIs are deterministic. We consider a UI to be deterministic if a software robot can replicate its execution. This is possible when: i) the input data of a UI can be determined automatically; or ii) the input data of a UI can be provided as input by the user when deploying the software robot. According to such constraints, we can provide the following rules to check whether a UI is deterministic or not.

1. UIs belonging to the *navigation* group (see Table 2) are always deterministic because they do not take in input any data; except the *select cell*, *select field*, and *select range* UIs which are removed during the filtering of the log (as described in Section 3.2);

2. UIs belonging to the *read* group are always deterministic because the only input they require is the source of the copied content (e.g., row and column of a cell), which is either constant or can be inputted by the user when deploying the software robot in UiPath;

3. UIs belonging to the *write* group that are of type *click* are always deterministic because they do not take in input any data, except the information regarding the element to be clicked which is always constant for a given candidate routine (by construction);

4. UIs belonging to the *write* group that are of type *paste* are always deterministic because they always retrieve data from the same source (i.e., the system clipboard).

5. UIs belonging to the *write* group that are of type *edit* are the only ones that are not always deterministic. In fact, these UIs are deterministic only if it is possible to determine the updated value of the edited elements (e.g., the value of a cell in a spreadsheet or of a text field in the web browser after the UI is executed). Furthermore, it has also to be possible to determine the target of the editing, although this is usually constant (if a web element) or can be inputted by the user when deploying the software robot in UiPath.

Algorithm 4 shows how we check these five rules given as input a candidate routine c_i and its routine instances $\mathscr{R}_{c_i}$, and how we compose the corresponding

Algorithm 4: Routine automatability assessment

input : Candidate Routine c_i, Routine Instances Set $\mathscr{R}_{c_i}$
output : Routine Specification (c_i, Λ)

```
1  Set Λ ← ∅;
2  Set E ← { "edit cell", "edit range", "edit field" };
3  Queue D ← ∅;
4  foreach Normalized UI ū ∈ c_i do
5      if getType(ū) ∉ E then
6          append ū to D;
7      else
8          k ← checkUIofTypeEdit(ū, c_i, R_{c_i});
9          Boolean d ← getDeterministic(k);
10         if d = true then
11             append ū to D;
12             Λ ← Λ ∪ getTransformationStep(k);
13 return (c_i, Λ)
```

routine specification of the input c_i. The algorithm starts by initializing the set E as a collection of *edit* UI types (*edit cell*, *edit range*, *edit field*). Then, it iterates over all the normalized UIs in the input c_i by checking their types. If the type of a normalized UI $\bar{u}$ is not in E (line 6), i.e., one of the rules 1 to 4 applies, we add it to the queue D, which stores all the deterministic UIs we identified. Otherwise, rule 5 applies. While rules 1 to 4 are simple checks on the UI types, the complexity of rule 5 required us to operationalize it through a separate algorithm, i.e., Algorithm 5, which is called within Algorithm 4 (line 8). Algorithm 5 returns a pair (d, λ), where d is a *boolean* (true if the input normalized UI is deterministic), and λ is a *data transformation step* required to automate $\bar{u}$ and therefore available only if $\bar{u}$ is deterministic. Once all the normalized UIs in the input c_i have been checked, Algorithm 4 outputs the *routine specification* of c_i, as the pair (c_i, Λ), where Λ is the set of all the *data transformation steps* we collected by executing Algorithm 5 (line 8).

Before moving to the final step of our approach, we describe how Algorithm 5 verifies whether an input (normalized) UI of type *edit* ($\bar{u}$) is deterministic. In essence, Algorithm 5 checks whether the value of the element edited by the execution of $\bar{u}$ can be deterministically computed from the UIs observed before $\bar{u}$ (in all the routine instances in $\mathscr{R}_{c_i}$). To do so, the algorithm looks for a possible data transformation function to compute the value of the edited element from the payloads of the UIs observed before $\bar{u}$. If such a data transformation function exists, $\bar{u}$ is considered to be deterministic, and the algorithm returns the identified function in the form of a data transformation step (which also includes source(s)

Algorithm 5: Check UI of Type Edit

input : Normalized UI $\bar{u}$, Candidate Routine c_i, Routine Instances Set $\mathscr{R}_{c_i}$
output : Boolean d, Transformation Step λ

```
1  Boolean d ← false;
2  Set C ← { "copy cell", "copy range", "copy field" };
3  Set E ← { "edit cell", "edit range", "edit field" };
4  Set P ← { "paste into cell", "paste into range", "paste" };
5  Set T ← ∅;
6  Set Π ← ∅;
7  Set K ← ∅;
8  Integer n ← getPosition(ū, c_i);
9  foreach r ∈ R_{c_i} do
10     UI u_1 ← get(r, n);
11     K ← K ∪ {u_1};
12     t_1 ← getTargetElement(u_1);
13     o ← getParameterValue(u_1, "Value");
14     Queue S ← ∅;
15     Queue I ← ∅;
16     for i ← n to 1 do
17         UI u_2 ← get(r, i);
18         Π ← Π ∪ {(r, u_2)};
19         if getType(u_2) ∈ P then
20             t_2 ← getTargetElement(u_2);
21             if t_2 = t_1 then
22                 for j ← i to 1 do
23                     UI u_3 ← get(r, j);
24                     if getType(u_3) ∈ C then
25                         s ← getTargetElement(u_3);
26                         append s to S;
27                         append getParameterValue(u_3, "Value") to I;
28                         break
29         else
30             if getType(u_2) ∈ E then
31                 t_2 ← getTargetElement(u_2);
32                 if t_2 = t_1 then
33                     push t_2 to S;
34                     push getParameterValue(u_2,"Value") to I;
35                     break
36     T ← T ∪ {(I, o)};
37 Transformation χ ← discoverTransformation(T);
38 if χ ≠ null then
39     d ← true;
40     λ ← (S, target, χ);
41 else
42     Set D ← discoverDependencies(K, Π);
43     if D ≠ ∅ then
44         d ← true;
45         S ← getSources(D);
46         χ ← extractTransformation(D);
47         λ ← (S, target, χ);
48 return (d, λ)
```

and target of the data transformation function). In the following, we walk through Algorithm 5.

We start by assuming that the UI in input is not deterministic, and we try to prove the opposite. We initialize to false the boolean variable which we will output at the end of the algorithm (line 1), and we create the necessary data structures (line 2 to 7). Given the input candidate routine c_i and the normalized UI $\bar{u}$, we extract the index of $\bar{u}$ within c_i (line 8). Then, for each routine instance $r \in \mathscr{R}_{c_i}$, we do what follows.

We get the instance of the normalized UI $\bar{u}$[2] by retrieving the UI of index n from r (line 10), and we store this UI (u_1) in the set K (line 11). We read the payload of u_1 to retrieve the target element (t_1, line 12), t_1 can be the ID of a web browser element or the location of a cell in a spreadsheet. Also, we read the payload of u_1 to retrieve the value of the target element after the editing (o, line 13). We initialize two queues, S (which stands for *sources*) and I (which stands for *inputs*). Queue S stores the ID or location of the (source) element(s) that produced the data used by the *edit* UI instance u_1; while queue I stores the data that was used by the *edit* UI instance u_1.

After this initialization, we iterate over all the UI instances preceding u_1 in r. Such an iteration goes backward from u_1 (position n in r) till the first UI instance in r (position 1) – line 16 to 35, unless we identify another UI instance of type *edit* performed on the same target element t_1 (see lines 30 to 32). In the iteration captured between line 16 to 35, we do the following.

We store all the preceding UI instances (u_2) into the set Π, alongside the routine instance they belong to (i.e., we store a pair (r, u_2) in Π). For each encountered u_2 of type *paste*, we check its target element and we compare it to the target element of u_1. If they are the same, we again traverse backward the routine instance from the *paste* UI until we find a *copy* UI u_3 (line 19 to 28).[3]. Then, we retrieve the target element of u_3 and we append it to queue S, and we add the copied value of u_3 to queue I (lines 26 and 27).

For each encountered u_2 of type *edit* (line 30), we check its target element and we compare it to the target element of u_1. If they are the same (line 32), we push the *target element* of u_2 to the front of queue S, and we push the *data content* of the target element after the editing performed by u_2 to the front of the queue I

[2]We recall that a UI instance contains all the parameters, both context and data ones.

[3]Our filtering approach, described in Section 3.2 guarantees that there exists a u_3 of type *copy* preceding the *paste* UI

(line 33 and 34). When we reach this point, we also stop the iteration over all the UI instances preceding u_1 because the value of the target element after performing u_1 is obtained from the last *edit* UI performed on the same target element and any other UI (i.e., *paste* UIs) between u_2 and u_1.

Finally, before moving to the next routine instance (i.e., returning to line 9), we store the input data and the output data observed in the current routine instance for the normalized UI $\bar{u}$ in the set T, which collects all the input and output data observed for *all* the instances of $\bar{u}$ (see line 36).

After performing all the above steps for each routine instance $r \in \mathscr{R}_{c_i}$, and collecting all the required data to identify a possible data transformation function into the sets T, K, and Π, we look for the data transformation function by leveraging two state-of-the-art tools: Foofah [27] and TANE [28]. First, we try to identify the data transformation function using Foofah, then – if Foofah fails – we use TANE.

Foofah requires in input two series of data values, one referred to as *input* and one referred to as *output*. We generate the two series from the pairs (I, O) that we collected in T, which capture examples of data transformations. From these examples, Foofah tries to synthesize an optimal data transformation function to convert input(s) to output.[4] We note that we run Foofah under the assumption that the output series is noise- and error-free, i.e., the analyzed data transformations are supposed to be correct.

However, Foofah suffers from two limitations: it is inefficient when the size of the input and output series is large; it cannot discover conditional data transformation functions (where different manipulations are applied depending on the input). Hence Foofah cannot deal with heterogeneous data. To address these limitations, we group the data transformation examples into equivalence classes, where each class represents a different structural pattern of the input data. To create these equivalence classes, for each data sample in the input data series, we discover its symbolic representation describing its structural pattern by applying *tokenization*. The tokenization that we apply replaces each maximal chained subsequence of symbols of the same type (either digits or letters) with a special token character ($\langle d \rangle +$ or $\langle a \rangle +$, resp.), and leaves any other symbol unaltered. For each equivalence class, we discover a data transformation function by providing to Foofah one randomly selected data transformation example from the equivalence class. The use of equivalence classes allows us to remove the heterogeneity of the input data and to facilitate the application of Foofah, which will operate only on a single

[4] For more details about Foofah refer to [27].

data transformation example.

If Foofah cannot identify a data transformation function (line 41), we turn to TANE, which can discover semantical data transformation functions (also known as *functional dependencies* [28]). TANE requires in input a table where each row contains $n-1$ input data values and an output data value in column n (this is conceptually similar to the input and output series required by Foofah). TANE analyzes each row of such a table to check if there exists any dependency between the values in the first $n-1$ columns and the value in column n.[5] An example of a semantical data transformation function discovered by TANE would be: if the value of column i is X, then the value of column n is always Y.

In our context, the input table for TANE is a table where each row represents the output data observed in all the UIs preceding $\bar{u}$ in a routine instance, and the last element of the row is the output data of the $\bar{u}$ instance in that routine (i.e., the value of the element edited by the execution of $\bar{u}$ in that routine instance). To build such a table, we require in input all the instances of $\bar{u}$ (which we stored in the set K) as well as all the instances of any UI preceding $\bar{u}$ (which we stored in the set Π). If TANE identifies a semantical data transformation function (line 43), we set $\bar{u}$ as deterministic (through the boolean d), and we compose the data transformation step using the output of TANE (see lines 44 to 47).

Table 5 shows an example of the dependency table that we would build from the log captured in Table 1 (assuming that the full-length UIs log contains nine instances of the routine showed in rows 1 to 24). Giving Table 5 in input to TANE, it would identify that the value of the last column (i.e., the type of student, domestic or international) can be deterministically generated by observing the value of column four (i.e., *country of residence*).

Full name	Date	Phone	Country of residence	Target
Albert Rauf	11-04-1986	043-512-4834	Germany	International
John Doe	11-03-1986	024-706-5621	Australia	Domestic
Steven Richards	18-06-1986	088-266-0827	Australia	Domestic
Hilda Diggle	31-07-1993	073-672-5593	New Zealand	International
Luca Bianchi	19-10-1998	029-211-4904	Italy	International
Igor	13-08-1993	040-656-3417	Ukraine	International
Ben Stanley	03-12-1991	244-557-2104	Australia	Domestic
Olga Mykolenchuk	11-04-2000	956-045-0703	Ukraine	International
Daniel Brown	06-04-1994	032-660-0403	New Zealand	International

Table 5: Example of a dependency table.

[5]For more details about TANE refer to [28].

If also TANE does not discover any data transformation function, it means that we are not able to automatically determine the value of the element edited by the execution of $\bar{u}$, consequently we assume that $\bar{u}$ is not deterministic. Otherwise, we output the data transformation step discovered.

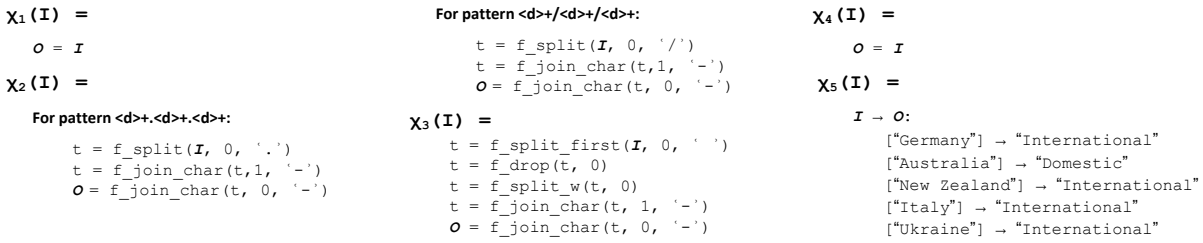


Figure 5: Transformation functions discovered from the running example

Transformation step	Sources	Target	Transformation function
λ_1	Cell A	Full Name	χ_1
λ_2	Cell B	Date	χ_2
λ_3	Cell C	Phone	χ_3
λ_4	Cell D	Country	χ_4
λ_5	Country	Status	χ_5

Table 6: Transformation steps

Considering our running example, Figure 5 shows the data transformations functions discovered by Foofah (t1 to t4) and by TANE (t5) when running Algorithm 5 on an hypothetical extended version of the UI log in Table 1 and giving as input the routine shown in rows 1 to 24 (Table 1) along all its instances, and the *edit* UIs at rows 6, 11, 16, 21, 23 (respectively, for identifying the data transformation functions from t1 to t5). Each data transformation function shows how input data is turned into output data. Although some rules are intuitive to interpret (e.g., t1 and t5), others may appear slightly cryptic. We refer to Foofah [27] and TANE [28] original studies for an extensive description of the set of rules that the two tools are capable to discover.

Finally, the data transformation functions are integrated into the data transformation steps, which also include the instantiation of the input and the output of the function, as shown in Table 6.

3.5. Routines aggregation

When a routine can be performed by executing a set of UIs without following a strict order, we may observe multiple execution variants of the same routine in the

log. For example, if a worker needs to copy the *first name*, the *family name*, and the *phone number* of a set of customers from a spreadsheet to different web-forms, she may choose to copy the data of each customer in any order (e.g., *first name*, *phone number*, and *family name*, or *family name*, *phone number*, *first name*). In such a scenario, the UI log would record several different execution variants of the same routine. Routine execution variants do not bring any additional value, rather they just generate redundancy within the log leading to the discovery of different routine specifications that would actually execute (once deployed as software bots) the same routine. Considering these routine specification as duplicates, this final step focuses on their removal.

To identify duplicate routine specifications, we start by generating for each routine discovered in the previous step its *data transformation graph*.

Definition 8 (Data Transformation Graph). *Given a routine specification (c_i, Λ), its* data transformation graph *is a graph $G_\Lambda = (D_\Lambda, L_\Lambda)$, where: D_Λ is the set of vertices of the graph, and each vertex $d \in D_\Lambda$ maps one data transformation step $\lambda \in \Lambda$; $L_\Lambda \subseteq D_\Lambda \times D_\Lambda$ is the set of edges of the graph, and each edge $(d_i, d_j) \in L_\Lambda$ represents a dependency between two data transformation steps capturing the fact that the target of the data transformation step mapped by d_i is (one of) the source(s) of the data transformation step mapped by d_j.*

Figure 6 shows the data transformation graph of the routine we discovered in the previous step in our running example.

Data transformation graphs can be used to check whether two routine specifications are equivalent, in fact, two routine specifications, (c_i, Λ_1) and (c_j, Λ_2), are equivalent if and only if the following two relations hold: i) their data transformation graphs are the same, i.e., $D_{\Lambda_1} = D_{\Lambda_2}$ and $L_{\Lambda_1} = L_{\Lambda_2}$; ii) their candidate routines c_i and c_j contain the same set of UIs, and all the UIs of type *click button* appear in the same order in both c_i and c_j.

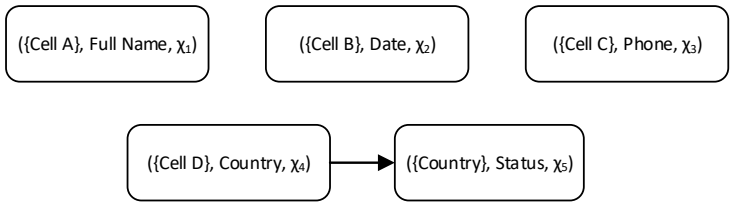


Figure 6: Data transformation graph example

By comparing each pair of routine specifications, we first create sets of equivalent routine specifications, and, for each set, we discard all the routine specifications but one. Ideally, we would like to retain the best routine specification of each set, however, we need to define what it means to be the *best* one. We can select the best routine specification by relying on different quantitative metrics, such as frequency, length, or duration of the candidate routine of a routine specification. For example, we can choose frequency as a selection criterion and retain from each set the routine specification whose candidate routine is the most frequent in the UI log.

Intuitively, the most frequent candidate routine represents the common routine execution, so that one may be tempted to use that criterion by default. However, the most frequent routine execution is not necessarily the optimal execution. For example, length or duration could represent better selection criteria. Length prioritizes short candidate routines over long ones, assuming that a candidate routine should comprise as few steps as possible. Duration prioritizes execution times over the number of steps. The duration of a candidate routine can be estimated as the average execution time of each routine instance of the candidate routine that is recorded in the UI log. Note, however, that the duration could be not always reliable since during the routine execution, the worker might perform activities that do not appear in the log or that are not relevant for the routine execution, thus involuntarily increasing the observed execution time of the routine. For this reason, we implemented a combination of length and frequency to select the best routine specification from each set. Precisely, we use length first and then compare the frequencies of the candidate routines having the same length.

4. Evaluation

We implemented our approach as an open-source Java command-line application[6] and also embedded this in the open-source tool Robidium [7]. Using the command-line application, we conducted a series of experiments to analyze the applicability of our approach in real-life settings. Specifically, we assessed to what extent our approach can rediscover routines that are known to be recorded in the input UI logs, and analyzed whether our approach is able to correctly identify automatable and not automatable user interactions within such routines.

Accordingly, we define the following research questions:

[6] Available at https://github.com/volodymyrLeno/RPM_Miner

- **RQ1.** Does the approach discover candidate routines that are known to exist in a UI log?
- **RQ2.** Does the approach discover automatable routines that are known to be present in a UI log?

4.1. Datasets

To answer our research questions, we rely on a dataset of 13 logs. These logs 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.[7] Table 7 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 7: UI logs characteristics.

The artificial logs (CPN1–CPN9) were generated from Colored Petri Nets (CPNs) in [3]. The CPNs used have increasing complexity, from low (the net used to generate CPN1) to high (the net used for CPN9). The underlying routines are characterized by a varying amount of non-deterministic user interactions injected. They involve simple data transformations, mostly in the form of copy-pasting. The logs generated were originally noise-free and segmented. We removed the segment identifiers to produce unsegmented logs.

[7]The real-life logs were recorded with the Action Logger tool [10]. All the logs are available at https://doi.org/10.6084/m9.figshare.12543587

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. While the routines captured in the logs are fully automatable, they include complex transformations to test the automatability assessment step of the approach.

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

4.2. Setup

To measure the quality of the discovered candidate routines, we use the Jaccard Coefficient (JC), which captures the level of similarity between discovered and ground truth routines. JC does not penalize the order of the interactions in a routine, which follows from the assumption that a routine could be executed by performing some actions in a different order. The JC between two routines is the ratio $\frac{n}{m}$, where n is the number of user interactions that are contained in both routines, while m is the total number of user interactions present in 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 use the segments of the artificial logs and the guidelines given to the author who performed the tasks in SR and RT.

The JC alone is not enough 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 its coverage.

We assess the quality of the automatable routines discovery by measuring precision, recall and F-score. For each discovered routine, we compute the corresponding confusion matrix, where *true positives* (TP) are correctly identified automatable user interactions, *true negatives* (TN) are correctly identified non-automatable user interactions, *false positives* (FP) are the user interactions that were wrongly marked as automatable, and *false negatives* (FN) are the user interactions that were wrongly marked as non-automatable. From the constructed confusion matrix, we calculate precision, recall and F-score as follows:

$$Precision = \frac{TP}{TP+FP}, \tag{1}$$

$$Recall = \frac{TP}{TP+FN}, \tag{2}$$

$$F\text{-}score = 2 \cdot \frac{Precision \cdot Recall}{Precision + Recall}. \tag{3}$$

We report the averages of these metrics for all the discovered routines in the log. We also report the average ratio of automatable user interactions for the routines in the log.

The results for the S1 and S2 logs were qualitatively assessed with the help of the University of Melbourne employees who performed the task. Specifically, we asked them to compare the rediscovered routines with the actions 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, using cohesion as a routine selection criterion with the minimum support threshold set to 0.1 and the minimum coverage threshold equal to 0.05.

4.3. Results

Table 8 shows the quality of the discovered routine candidates. Although the synthetic logs only contain the user interactions that belong to routines, we achieved perfect coverage for three logs only, namely CPN1, CPN4 and CPN6.

This is because some execution patterns were observed very rarely. Since the SR and RT logs contain noise, the coverage cannot be 1 in these two cases. For six logs out of eleven logs, the discovered routines match with the ground truth. Overall, the JC is very high, above 0.95 for all the logs except CPN5. The underlying model of the CPN5 log consists of multiple branches, generating 36 different executions. Considering the fact that some execution patterns are not frequent enough, we discovered only partial routines. As can be seen clearly, for this log we also achieved the lowest coverage (0.84). For the RT log we found two routines consisting of an identical set of actions. These routines were not merged though, because they are characterized by different transformation functions.

UI Log	**# Routines discovered**	**Length (Max)**	**Length (Avg.)**	**Total coverage**	**JC**
CPN1	1	14	14.00	1.00	1.000
CPN2	2	15	14.50	0.95	1.000
CPN3	3	19	14.33	0.93	1.000
CPN4	4	14	14.00	1.00	1.000
CPN5	8	8	7.38	0.84	0.880
CPN6	2	11	10.00	1.00	1.000
CPN7	7	10	9.43	0.93	0.971
CPN8	6	18	10.67	0.91	0.967
CPN9	6	18	14.67	0.95	1.000
SR	2	31	30.00	0.917	0.967
RT	2	61	61.00	0.903	0.967

Table 8: Candidates identification

Table 9 shows the quality of the automatable routines discovery. We correctly identified all the automatable and not automatable user interactions for the CPN3, CPN6 and SR logs. The routines recorded in the CPN3 and SR logs are fully automatable. Although the RT log contains automatable routines only, our approach failed to discover some of the underlying transformations, and, therefore, incorrectly marked some interactions as not automatable. Some of the user interactions of the synthetic logs were wrongly identified as automatable. Although the data values of such interactions can be deterministically computed, the locations of the edited elements were completely random as it was intended in the corresponding models. Thus, in practice, such interactions are not automatable. The routines discovered from the CPN5 log are characterized by the lowest number of automatable user interactions, and we achieved the lowest recall for this log (0.805). Overall, F-score is high, above 0.85 for all the logs, except CPN7 and CPN8. For

these logs we also achieved the lowest recall, meaning that some interactions of the corresponding routines were wrongly identified as not automatable. Although in the CPN models used to generate the artificial logs, some of the interactions are not deterministic, they are automatable in the context of the discovered routines. For example, for the CPN9 log we discovered six routines that correspond to the different branches within the model. For all the executions of a branch we use the same data values, and hence, the corresponding user interactions are automatable.

UI Log	**RAI (Avg.)**	**Precision (Avg.)**	**Recall (Avg.)**	**F-score (Avg.)**
CPN1	1.000	0.928	1.000	0.963
CPN2	0.931	0.926	1.000	0.961
CPN3	1.000	1.000	1.000	1.000
CPN4	0.786	1.000	0.846	0.917
CPN5	0.728	0.812	1.000	0.896
CPN6	0.742	1.000	1.000	1.000
CPN7	0.546	0.907	0.805	0.841
CPN8	0.612	0.897	0.823	0.845
CPN9	0.741	0.951	0.886	0.916
SR	1.000	1.000	1.000	1.000
RT	0.967	1.000	0.967	0.983

Table 9: Automatable routines discovery

From the S1 log we discovered five fully automatable routines. The first routine consists in manually adding graduate research student applications to the student record in the university's student management system. The application is then assessed, and the student is notified of the outcome. The second routine consists in lodging a ticket to verify possible duplicate applications. When a new application is entered in the 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 routines represent exceptional cases, where the employee either executed the first or the second routine in a different manner (i.e., by altering the order of the actions or overlapping routines executions). These routines were not identified as duplicate because they are characterized by different sequences of button clicks.

To assess the results, we showed the discovered routines 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 routines are alternative executions of the first routine.[8]

While the results from the S1 log were positive, our 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. 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.

Table 10 shows the execution time for each step of the approach. As we can see, the most computationally heavy step is the automatability assessment. For all the logs, this step took the largest amount of time, except for the CPN5, S1, and S2 logs. While the execution time is still reasonably low for all the artificial logs, it substantially increases for the SR and RT logs. In these two logs, the automatability assessment took 99 percent of the total computation time. This is caused by the fact that the underlying transformations in these two logs were very complex, often involving regular expressions or long sequences of manipulations. In contrast, all the transformations in the CPN1-CPN9 logs were simple copy-paste operations. Overall, for the synthetic logs, the approach took no more than 42 seconds. The aggregation step required the smallest amount of time. For the CPN1 log, we discovered only one routine, and, therefore, we did not have to apply any aggregation. For the S1 and S2logs, the most time taking step was the segmentation. The CFGs constructed for these logs were very complex, with a high number of loops. This significantly increased the time to identify back-edges in such CFGs and, therefore, the total time of segmentation.

4.4. Threats to validity

The reported evaluation has a number of threats to validity. First, a potential threat to internal validity is the fact that the context parameters (i.e. the attributes in the log that capture the notion of "user interaction") were manually selected. These context parameters are required as one of the inputs of the proposed method (in addition to the UI log). To mitigate this threat, the parameters were first selected by each of the two authors of the paper independently, then cross-checked to reach a mutual agreement, and then validated by the other authors based on their understanding of the event logs in question.

Another possible threat to internal validity is the limited use of parameter values to configure the approach at hand. To ensure we do not miss any significantly

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

UI Log	Execution time (sec)				
	Segmentation	**Candidates identification**	**Automatability assessment**	**Aggregation**	**Total**
CPN1	0.337	2.766	7.148	–	10.251
CPN2	2.521	4.482	17.408	0.010	24.421
CPN3	1.570	9.781	15.545	0.010	26.906
CPN4	0.637	5.013	18.409	0.009	24.068
CPN5	1.169	20.223	19.761	0.008	41.161
CPN6	1.376	3.811	6.102	0.010	11.299
CPN7	2.497	15.196	17.594	0.010	35.297
CPN8	2.469	14.605	17.399	0.010	34.483
CPN9	2.877	12.272	18.798	0.013	33.960
SR	0.801	8.077	845.255	0.013	854.146
RT	2.022	8.657	1066.041	0.011	1076.731
S1	29.052	14.066	21.400	0.011	64.529
S2	403.903	152.474	–	–	556.377

Table 10: Execution time

important behavior in the logs, we used very low support and coverage, equal to 0.1 and 0.05, respectively.

A potential threat to external validity is given by the use of a limited number of real-life logs (four). These logs focus on one type of task that can be automated via RPA, namely data transferring. These logs, however, exhibit different characteristics in terms of the complexity of the captured processes and log size. To mitigate this threat, we additionally performed a more extensive evaluation on a battery of artificial logs. For two real-life logs, we had no information about the underlying processes. Therefore we evaluated the results qualitatively with the workers responsible for their execution. To ensure the full reproducibility of the results, we have released all the logs, both real-life and artificial, used in our experiments. The only exceptions are the S1 and S2 logs as they contain sensitive information.

5. Conclusion

This paper presented an approach to discover automatable routines from 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. These paths represent sequences of actions that are repeated multiple times within the event log, possibly with some variations. Once the log is

segmented, a noise-resilient sequential pattern mining technique is used to extract frequent patterns that corresponds to the candidate routines. Next, the candidate routines are assessed for their amenability to automation. For each routine, a corresponding executable specification is synthesized, which can be compiled into an RPA script. Finally, the approach identifies semantically equivalent routines in order to produce a non-redundant set of automatable routines.

The approach has been implemented as an open-source tool, namely Robidium. This article reported on an evaluation of the fit-for-purpose and computational efficiency of the proposed approach. The evaluation shows that the approach can rediscover routines injected into synthetic logs, and that it discovers relevant routines in real-life logs. For most logs, the execution time does not exceed one minute. The only exceptions related to logs where we deliberately injected complex data transformations or where the routine instances overlap in the UI log.

The proposed approach makes a number of limiting assumptions. First, the effectiveness of the approach is sensitive to noise, e.g. clicks that are not related to the routine itself or clicks resulting from user mistakes. 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 frequently observed in the log. Recurring noise affects the accuracy of the results. To address this limitation, we will investigate the use of alternative segmentation and sequential pattern discovery techniques that incorporate noise tolerance mechanisms. Another avenue is to discover sequential patterns using the approach outlined in this article and then to filter out patterns that are *chaotic* in the sense that their occurrence does not affect the probability of other patterns occurring subsequently nor vice-versa. This latter approach has been studied in the context of event log filtering for process mining in [29].

Second, the approach is designed for logs that capture consecutive routine executions. In practice, routine instances may sometimes overlap (cf. the S2 real-life log in the evaluation). A possible avenue to address this limitation is to search for overlapping frequent patterns directly in the unsegmented log, instead of first segmenting it and then finding patterns in the segmented log. This approach has been previously investigated in the context of so-called Local Process Mining (LPM), where the goal is to discover process models capturing frequently repeated (and possibly overlapping) behavior in an unsegmented sequence of events [30].

When assessing the automatability of a routine, the propsoed approach assumes that the values of the edited fields are entirely derived from the (input) fields that are explicitly accessed (e.g., via copy operations) during the routine's

execution. Hence, it will fail to identify automatable user interactions in the case where a worker visually reads from a field (without performing a *copy* operation on it) and writes what they see into another field. An avenue for addressing this limitation is to complement the proposed method with optical character recognition techniques over screenshots taken during the UI log recording, so as to be able to detect that some of the outputs of a routine come from fields that have not been explicitly accessed via a copy-to-clipboard operation.

Furthermore, the proposed approach is unable to discover conditional behavior, where the transformation function for the target field depends on the value of another field. Consider, for example, a routine that involves copying delivery data. If the delivery country is USA, then the month comes before the day (MM/DD/YYYY), otherwise the day comes before the month. Here, the transformation function depends on a condition of the form "country = USA", which the proposed approach is unable to discover. In a similar vein, the proposed approach is able to discover transformations that depend on the structural pattern of the value of the input field(s), but it fails to distinguish the patterns that, although having the same syntactical structure, have different semantics. Following the example above, our approach will put both date types into the same equivalence class. Addressing this limitation would require the development of more sophisticated data transformation discovery techniques, beyond the capabilities of Foofah.

Finally, the method to detect if two routines are semantically equivalent assumes that all button clicks in a UI are effectful, meaning that their presence and the order in which they occur affect the outcome of the routine. In practice, some clicks may have no effect on the routine's outcome. For example, some clicks may simply serve to pop up a help box, while others may just serve to move from one page to another in a listing. To address this limitation, we foresee extensions of the proposed method where the alphabet of the UI log is extended with a richer array of actions, and where the routine discovery approach can be configured via a language for the specification of action effects.

Acknowledgments. The authors thank Stanislav Deviatykh for his help in the prototype implementation. This research is supported by the Australian Research Council (DP180102839) and the European Research Council (project PIX).

References

[1] H. Leopold, H. van der Aa, H. A. Reijers, Identifying candidate tasks for robotic process automation in textual process descriptions, in: Enterprise,

business-process and information systems modeling, Springer, 2018, pp. 67–81 (2018).

[2] A. Jimenez-Ramirez, H. A. Reijers, I. Barba, C. Del Valle, A method to improve the early stages of the robotic process automation lifecycle, in: International Conference on Advanced Information Systems Engineering, Springer, 2019, pp. 446–461 (2019).

[3] A. Bosco, A. Augusto, M. Dumas, M. L. Rosa, G. Fortino, Discovering automatable routines from user interaction logs, in: Proceedings of the Business Process Management Forum, Springer, 2019 (2019).

[4] J. Gao, S. J. van Zelst, X. Lu, W. M. van der Aalst, Automated robotic process automation: A self-learning approach, in: OTM Confederated International Conferences" On the Move to Meaningful Internet Systems", Springer, 2019, pp. 95–112 (2019).

[5] V. Leno, M. Dumas, M. L. Rosa, F. M. Maggi, A. Polyvyanyy, Automated discovery of data transformations for robotic process automation, ArXiv abs/2001.01007 (2020).

[6] S. Agostinelli, M. Lupia, A. Marrella, M. Mecella, Automated generation of executable RPA scripts from user interface logs, in: Business Process Management: Blockchain and Robotic Process Automation Forum - BPM 2020, Vol. 393 of Lecture Notes in Business Information Processing, Springer, 2020, pp. 116–131 (2020).

[7] V. Leno, S. Deviatykh, A. Polyvyanyy, M. L. Rosa, M. Dumas, F. M. Maggi, Robidium: Automated synthesis of robotic process automation scripts from UI logs, in: Proceedings of the Best Dissertation Award, Doctoral Consortium, and Demonstration & Resources Track at BPM 2020, Vol. 2673 of CEUR Workshop Proceedings, CEUR-WS.org, 2020, pp. 102–106 (2020).

[8] V. Leno, A. Augusto, M. Dumas, M. L. Rosa, F. M. Maggi, A. Polyvyanyy, Identifying candidate routines for robotic process automation from unsegmented UI logs, in: 2nd International Conference on Process Mining, ICPM 2020, Padua, Italy, October 4-9, 2020, IEEE, 2020, pp. 153–160 (2020).

[9] V. Leno, A. Polyvyanyy, M. Dumas, M. L. Rosa, F. M. Maggi, Robotic process mining: Vision and challenges, Business & Information Systems Engineering (2020).

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

[11] H. Cao, N. Mamoulis, D. W. Cheung, Discovery of periodic patterns in spatiotemporal sequences, IEEE Transactions on Knowledge and Data Engineering 19 (4) (2007) 453–467 (2007).

[12] Y. Zhu, M. Imamura, D. Nikovski, E. Keogh, Matrix profile vii: Time series chains: A new primitive for time series data mining, in: 2017 IEEE International Conference on Data Mining (ICDM), IEEE, 2017, pp. 695–704 (2017).

[13] M. Spiliopoulou, B. Mobasher, B. Berendt, M. Nakagawa, A framework for the evaluation of session reconstruction heuristics in web-usage analysis, Informs journal on computing 15 (2) (2003) 171–190 (2003).

[14] D. Bayomie, A. Awad, E. Ezat, Correlating unlabeled events from cyclic business processes execution, in: Proceedings of the 28th International Conference on Advanced Information Systems Engineering (CAiSE), Springer, 2016, pp. 274–289 (2016).

[15] D. R. Ferreira, D. Gillblad, Discovering process models from unlabelled event logs, in: Proceedings of the 7th International Conference on Business Process Management (BPM), Springer, 2009, pp. 143–158 (2009).

[16] S. Agostinelli, Automated segmentation of user interface logs using trace alignment techniques (extended abstract), in: C. D. Ciccio, B. Depaire, J. D. Weerdt, C. D. Francescomarino, J. Munoz-Gama (Eds.), Proceedings of the ICPM Doctoral Consortium and Tool Demonstration Track 2020, Vol. 2703 of CEUR Workshop Proceedings, CEUR-WS.org, 2020, pp. 13–14 (2020).

[17] H. Dev, Z. Liu, Identifying frequent user tasks from application logs, in: Proceedings of IUI 2017, Springer, 2017, pp. 263–273 (2017).

[18] J. Han, H. Cheng, D. Xin, X. Yan, Frequent pattern mining: current status and future directions, Data mining and knowledge discovery 15 (1) (2007) 55–86 (2007).

[19] S. D. Lee, 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 (2004).

[20] E. Ohlebusch, T. Beller, Alphabet-independent algorithms for finding context-sensitive repeats in linear time, Journal of Discrete Algorithms 34 (2015) 23–36 (2015).

[21] J. Wang, J. Han, Bide: Efficient mining of frequent closed sequences, in: Proceedings of the 20th international conference on data engineering, IEEE, 2004, pp. 79–90 (2004).

[22] F. Fumarola, P. F. Lanotte, M. Ceci, D. Malerba, Clofast: closed sequential pattern mining using sparse and vertical id-lists, Knowledge and Information Systems 48 (2) (2016) 429–463 (2016).

[23] A. Augusto, R. Conforti, M. Dumas, M. La Rosa, F. M. Maggi, A. Marrella, M. Mecella, A. Soo, Automated discovery of process models from event logs: Review and benchmark, IEEE Trans. Kn. Data Eng. 31 (4) (2019) 686–705 (2019).

[24] J. Geyer-Klingeberg, J. Nakladal, F. Baldauf, F. Veit, Process mining and robotic process automation: A perfect match, in: Proceedings of the Dissertation Award, Demonstration, and Industrial Track at BPM 2018, Vol. 2196 of CEUR Workshop Proceedings, CEUR-WS.org, 2018, pp. 124–131 (2018).

[25] S. Gulwani, Automating string processing in spreadsheets using input-output examples, in: Proceedings of the 38th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 2011, 2011, pp. 317–330 (2011).

[26] M. Sharir, A strong-connectivity algorithm and its applications in data flow analysis, Computers & Mathematics with Applications 7 (1) (1981) 67–72 (1981).

[27] Z. Jin, M. R. Anderson, M. J. Cafarella, H. V. Jagadish, Foofah: Transforming data by example, in: S. Salihoglu, W. Zhou, R. Chirkova, J. Yang, D. Suciu (Eds.), Proceedings of the 2017 ACM International Conference on Management of Data, SIGMOD Conference 2017, Chicago, IL, USA, May 14-19, 2017, ACM, 2017, pp. 683–698 (2017).

[28] Y. Huhtala, J. Kärkkäinen, P. Porkka, H. Toivonen, TANE: an efficient algorithm for discovering functional and approximate dependencies, Comput. J. 42 (2) (1999) 100–111 (1999).

[29] N. Tax, N. Sidorova, W. M. P. van der Aalst, Discovering more precise process models from event logs by filtering out chaotic activities, J. Intell. Inf. Syst. 52 (1) (2019) 107–139 (2019).

[30] N. Tax, N. Sidorova, R. Haakma, W. M. P. van der Aalst, Mining local process models, J. Innov. Digit. Ecosyst. 3 (2) (2016) 183–196 (2016).