Discovering Automatable Routines From User Interaction Logs

Antonio Bosco[1,2], Adriano Augusto[1,3], Marlon Dumas[3], Marcello La Rosa[1], and Giancarlo Fortino[2]

[1] University of Melbourne, Australia
{antonio.bosco,a.augusto,marcello.larosa}@unimelb.edu.au
[2] University of Calabria, Italy
giancarlo.fortino@unical.it
[3] University of Tartu, Estonia
marlon.dumas@ut.ee

Abstract. The complexity and rigidity of legacy applications in large organizations engender situations where workers need to perform repetitive routines to transfer data from one application to another via their user interfaces, e.g. moving data from a spreadsheet to a Web application or vice-versa. Discovering and automating such routines can help to eliminate tedious work, reduce cycle times, and improve data quality. Advances in Robotic Process Automation (RPA) technology make it possible to automate such routines, but not to discover them in the first place. This paper presents a method to analyse user interactions in order to discover routines that are fully deterministic and thus amenable to automation. The proposed method identifies sequences of actions that are always triggered when a given activation condition holds and such that the parameters of each action can be deterministically derived from data produced by previous actions. To this end, the method combines a technique for compressing a set of sequences into an acyclic automaton, with techniques for rule mining and for discovering data transformations. An initial evaluation shows that the method can discover automatable routines from user interaction logs with acceptable execution times, particularly when there are one-to-one correspondences between parameters of an action and those of previous actions, which is the case of copy-pasting routines.

1 Introduction

The complexity and rigidity of legacy application landscapes in large organizations engender situations where workers need to perform repetitive routines to transfer data from one application to another via their user interfaces, e.g. moving data from a spreadsheet application to a Web application or vice-versa. Discovering and automating such routines can not only lead to the elimination of tedious work, but it can also reduce cycle times and improve data quality by ensuring that all data are transferred correctly.

Robotic Process Automation (RPA) tools [1] allow us to automate such routines by recording scripts that encode sequences of interactions with Web and desktop applications, such as opening a file, selecting a field in a form or a cell in a spreadsheet, and copy-pasting data across fields/cells. While these tools allow us to automate a range of routines, they do not allow us to determine which routines to automate in the first place.

This paper presents a method to analyse User Interaction logs (UI logs) in order to discover sequences of actions (herein called *routines*) that are fully deterministic are

hence automatable using RPA tools. In this context, we say that a routine is automatable if its first action is always triggered when a condition is met (the routine's *activation condition*) and the value of each parameter of each action can be computed from the values of parameters of previous actions (i.e. all actions are deterministic).

The proposed method takes as input a UI log consisting of a set of user interaction sessions (herein called *routine traces*). Each routine trace consists of a sequence of interactions (herein called actions for short). Each action has a type (e.g. select, copy, paste, etc.) and a set of parameters (e.g. the identifier of the UI element upon which the action is performed, and the inputs and outputs of the action). Given a UI log, our method outputs a set of routine specifications. A routine specification is a tuple consisting of an activation condition and a sequence of action specifications. An action specification, in turn, is a tuple consisting of an action type and a set of functions to compute the action's parameters from the parameters of previous actions.

The method starts by compressing the UI log into a Deterministic Acyclic Finite State Automaton (DAFSA). It then applies an algorithm to decompose biconnected graphs (of which a DAFSA is an exemplar) into Single-Entry Single-Exit (SESE) regions. Some of these regions correspond to sequences of actions. For each such sequence, the method checks if every action is deterministic. If so, it tries to discover an activation condition using a rule mining technique. If, on the other hand, an action in the middle of the sequence is not deterministic, the sequence is split into subsequences for which the method tries to discover activation conditions separately. A routine specification is generated for each (sub)sequence for which an activation condition is found.

The paper reports on a synthetic evaluation aimed at testing if the method can extract automatable routines with acceptable execution times.

The rest of the paper is structured as follows. Section 2 introduces a running example. Next, Section 3 describes our proposed approach, while Section 4 discusses its evaluation. Finally, Section 5 discusses related work while Section 6 summarizes the contributions and outlines directions for future work.

2 Running Example

Below, we introduce a real-life scenario to illustrate our approach. The example is inspired by work performed by the Service Improvement Team at the University of Melbourne, which applies RPA to automate various student-facing processes such as student admission.We specifically consider the task of updating the student residential data, manually performed by a university officer. Students' data is stored both in a student management system (accessible via Web interface) and on local Excel files for backup purposes. We assume that the university's student admission office is not interested in recording on its backup files the residential address of international students.

Table 1 shows an extract of an UI log of this task in the format generated by a *UI logger* we have developed. [4] The extract in Table 1 captures the sequence of actions performed by one employee to complete the task for a domestic student. The employee is already logged into the student management system, and she starts the task by clicking on the *students* button on the Web interface. Then, she enters the student ID in the *ID student* text field and presses the *enter* key to confirm. The Web interface displays a

[4] Available at: `https://github.com/apromore/RPA_UILogger`

	Action Type	Action Parameters			
		Param-1	**Param-2**	**Param-3**	**Param-4**
1	Click button	Target:Web	Label: STUDENTS		
2	Fill the text field	Target:Web	Label: ID Student	Value: 010234	
3	Press key	Target:Web	Label: ENTER		
4	Click button in row	Target:Web	Label: Update	ID Row: 010234	
5	Fill the text field	Target:Web	Label: Address	Value: 19 Parkville St, Burnley VIC 3121	
6	Fill the text field	Target:Web	Label: Country	Value: Australia	
7	Open file	Target:Excel	Name: 010234	Path: C:/Students/Australia/	Extension: .xls
8	Copy (Ctrl+C)	Target:Web	From: Address	Value: 19 Parkville St	
9	Paste (Ctrl+V)	Target:Excel	Row: 5	Column: A	Value: 19 Parkville St
10	Save file (Ctrl+S)	Target:Excel			
11	Click button	Target:Web	Label: Confirm Backup		

Table 1: Routine trace from the UIL, domestic student scenario.

list of students, including the one searched (see Fig. 1). Next to each student entry, two options are available: *update* and *open*. The employee clicks on the *update* button, since she intends to update the residential data of the student. A new window opens with the student details, including the residential data, i.e. address and country. The employee types the new address and the country and, in the case of a domestic student (i.e. country is Australia), she opens the corresponding backup Excel file to copy part of the address (the street only) from the Web interface to the Excel file. She then saves the file changes and confirms the update on the Web interface by clicking button *confirm backup*.

In the case of an international student (Table 2), the update of the residential address on the student management system follows the same sequence of actions, but no backup is required. Thus, after the update on the Web interface, the employee clicks on the *No Backup* button, a dialogue box pops up to double check the selection, she clicks on the *ok* button, and finally clicks on the "Confirm" button to apply the update.

	Action Type	Action Parameters		
		Param-1	**Param-2**	**Param-3**
1	Click button	Target:Web	Label: STUDENTS	
2	Fill the text field	Target:Web	Label: ID Student	Value: 010236
3	Press Key	Target:Web	Label: ENTER	
4	Click button in Row	Target:Web	Label: Update	ID Row: 010236
5	Fill the text field	Target:Web	Label: Address	Value: 106 Tantau Ave, Cupertino CA 95014
6	Fill the text field	Target:Web	Label: Country	Value: USA
7	Click button	Target:Web	Label: No Backup	
8	Click button	Target:Web	Label: Ok	
9	Click button	Target:Web	Label: Confirm	

Table 2: Routine trace from the UIL, international student scenario.

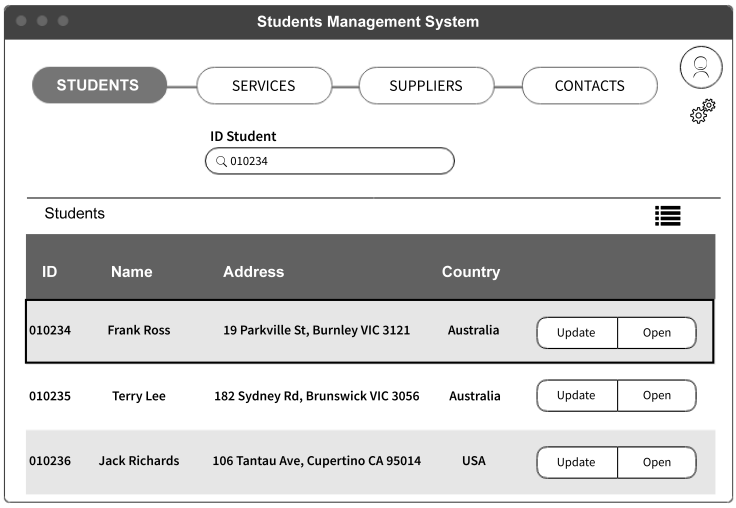

Fig. 1: Web interface of the student management system.

This example shows that a given task (e.g. updating the student residential data) may be performed via different sequences of actions (routines). In this case, there is one routine for the domestic students and another for international ones. The automation of a task requires one to identify the boundaries of each routine from the UI log, and within these routines to determine which actions are deterministic and can thus be automated.

3 Approach

In this section, we give a detailed description of the three steps composing our approach (see Fig. 2). Given as input a UI log, as first step, we parse the UI log into a DAFSA, and we extract from this latter the *flat-polygons* (the candidate automatable routines), which represent actions sequences of different length. In the second step, each of the candidate automatable routines is analysed by checking whether each of its actions is deterministic or not, i.e. the action could be executed in a systematic way by an RPA script (e.g. a software bot). The output of the second step is a set of action specifications. An action specification is a tuple consisting of: an action; and a set of functions to automatically determine all the action parameters values. Last, in the third step we extract from the candidate automatable routines, the maximal sequences of deterministic actions, and for each of them we discover the activation condition of the first action. The final output of our approach is a set of routine specifications. Each routine specification being a tuple consisting of: an activation condition to automate the routine; and a sequence of action specifications.

3.1 Definitions

The proposed approach takes as input UI logs that record multiple executions of a routine by one or several users. A UI log consists of routine traces, each one corresponding to the recording of one execution of the routine by a user. Each routine trace consists

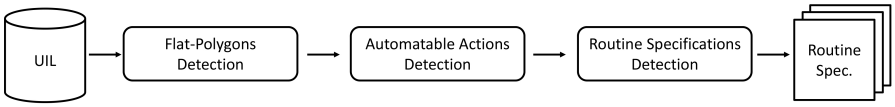

Fig. 2: Approach overview.

of actions that a given user performs sequentially using one or more applications (e.g. a browser and a spreadsheet application). These concepts are formalized below.

Definition 1. ***[Action]*** *An action a is defined as $a = (\tau, P, V, \phi)$, where: τ is the action type (e.g. click button, open file, press key, etc.); P is the set of action parameters (e.g. button name, file name, key name, etc.); V is the set of the values assigned to the action parameters; $\phi : P \to V$ is the function matching each action parameter to its value. Given two actions $a_1 = (\tau_1, P_1, V_1, \phi_1)$ and $a_2 = (\tau_2, P_2, V_2, \phi_2)$, a_1 and a_2 are equal if and only if (iff) they are of the same type and have the same set of parameters, i.e. $a_1 = a_2 \iff \tau_1 = \tau_2 \wedge P_1 = P_2$. Note that, two actions having the same set of parameters but with different assigned values are still considered equal.*

Definition 2. ***[Routine Trace and User Interaction Log]*** *A Routine Trace ρ is a sequence of actions $\rho = \langle a_1, a_2, \ldots, a_n \rangle$, we define the operator $\in$ for routine traces such that given an action $\hat{a}$, $\hat{a} \in \rho$ iff $\exists i \in [1,n] \mid a_i = \hat{a}$. Also, we refer to $a_i \in \rho = \langle a_1, a_2, \ldots, a_n \rangle, 0 < i < n$ as $\rho[i]$. Given two routine traces $\rho_1 = \langle a_1, a_2, \ldots, a_n \rangle$ and $\rho_2 = \langle \hat{a}_1, \hat{a}_2, \ldots, \hat{a}_n \rangle$, ρ_1 and ρ_2 are equal iff $\forall i \in [1,n] a_i = \hat{a}_i$. An User Interaction Log (UI log) $\mathscr{L}$ is a multiset of routine traces.*

In the proposed approach, we will sometimes reason in terms of fragments of a routine, in particular prefixes and suffixes as defined below.

Definition 3. ***[Routine Trace Prefix and Suffix]*** *Given a routine trace $\rho = \langle a_1, a_2, \ldots, a_n \rangle$ we define its $i^t h$ prefix as $\rho^{\to i} = \langle a_1, a_2, \ldots, a_i \rangle \wedge 1 < i < n$; and its $i^t h$ suffix as $\rho^{i \to} = \langle a_i, a_{i+1}, \ldots, a_n \rangle \wedge 1 < i < n$. Note that, prefixes and suffixes of routine traces (and sub-traces of routine traces) are routine traces themselves.*

Given a UI log, the goal of our proposed approach is to discover automatable routines, i.e. sets of routine traces having the parameters' values of each of their actions derivable from previous actions parameters' values.

Definition 4. ***[Deterministic Action]*** *Given an action $\hat{a} = (\tau, P, V, \phi)$, a UI log $\mathscr{L}$, and the set of the routine traces $R = \{\rho \in \mathscr{L} \mid \rho^{\to i}[i] = \hat{a} \vee \rho^{j \to}[1] = \hat{a}\}$, $\hat{a}$ is deterministic in $\mathscr{L}$, iff $\forall \hat{p} \in P$ one of the following holds: i) $\phi(\hat{p})$ is constant, $\forall (\rho, a_x) \mid \rho \in R \wedge a_x = (\tau, P, V_x, \phi_x) \in \rho \wedge \hat{a} = a_x \Rightarrow \phi_x(\hat{p}) = \phi(\hat{p})$; ii) $\phi(\hat{p})$ depends on the parameters values of the actions preceding $\hat{a}$ in the routine trace, formally, $\forall (\rho, a_x) \mid \rho \in R \wedge a_x = (\tau, P, V_x, \phi_x) \in \rho \wedge \hat{a} = a_x \Rightarrow \phi(\hat{p}) = \omega(V_1, V_2, \ldots, V_{x-1})$, where $V_i, i \in [1, x-1]$ is the set of parameters values of the action $\rho[i]$. Function ω is called a* dependency function*, or* dependency *for short.* [5]

Definition 5. ***[Automatable Routine (Trace)]*** *A Routine Trace $\rho = \langle a_1, a_2, \ldots, a_n \rangle$ is automatable iff $\forall a_i = (\tau_i, P_i, V_i, \phi_i), i \in [1,n]$, a_i is deterministic.*

Beyond identifying automatable routine, we seek to produce specifications thereof for their automation. Hence, we define an automatable routine specification as follows.

[5] In general, ω can be any function.

Definition 6. ***[(Automatable) Routine Specification]*** *A routine specification is a tuple* $(C, \langle AS \rangle)$ *where C is an activation condition and* $\langle AS \rangle$ *is a sequence of action specifications. An* activation condition *is a Boolean function over a set of actions, which can be evaluated at any point in a routine of a UI log, and such that when this condition is true, an instance of the routine specification is observed. An* action specification *is a tuple* (a, Ω) *such that a is an action* (τ, P, V, ϕ)*, and* Ω *is a set of parameter mappings. A* parameter mapping *is a tuple* $(\hat{p}, \omega)$ *such that* $\hat{p} \in P$*, and* ω *is a dependency function which computes the value of* $\hat{p}$ *from the previous actions parameters' values.*

3.2 Flat-Polygons Detection

The first step of our approach consists of detecting the flat-polygons, to do so, we execute Algorithm 1. Given as input a UI log, first, we build its DAFSA (line 1).

The DAFSA of a UI log is an acyclic automaton obtained by prefix-compressing and suffix-compressing the routine traces in the UI log. In other words, if multiple traces share the same prefix, this prefix is represented as one single sequence of states in the DAFSA, and conversely, if multiple traces share the same suffix, this suffix is represented as one single sequence of states. For details on how a DAFSA can be constructed from a set of routine traces, we refer to [13]. We observe that the DAFSA is a lossless representation of the UI log (it does not add nor remove any behavior), where each path of the DAFSA captures a different routine trace of the UI log, and each edge of a path captures an action of the routine trace, meaning that the set of all the paths in the DAFSA is exactly equal to the UI log. The prefix and suffix compression as well as the lossless feature of the DAFSA are the reasons why we chose it to represent the UI log. Indeed, capturing the UI log behavior in a lossless manner is necessary to detect and analyse the deterministic behavior recorded in the UI log. While the prefix and suffix compression allow us to easily identify each routine trace variant and where the variants start or end. Indeed, each decision point of the DAFSA (i.e. a DAFSA state with multiple outgoing edges) matches a routine variant starting point.

Once we generate the DAFSA, we compute its RPST [12] (line 2).

The RPST of a DAFSA is a *tree* where: the *nodes* are the single-entry single-exit (SESE) regions of the DAFSA; and the *edges* of the tree denote the containment relations between the SESE regions. Specifically, the children of a SESE region in the tree are the SESE regions that it directly contains. Regions at the same level of the tree represent a sequence of SESE regions in the DAFSA. Each SESE region is represented by a set of DAFSA edges, depending on how these edges are related, a SESE region can be of one of four types. A *trivial* region consists of a single DAFSA edge (i.e. a routine trace action). A *polygon* is a sequence of regions (e.g. a sequence of trivial regions). A *bond* is a region where all the child regions share two common DAFSA states, one being the entry state and the other being the exit state of the bond. Any other region is a *rigid*.

The RPST allows us to detect straightforward the flat-polygons. A flat-polygon is a polygon region where all its children are trivial regions. It follows that a flat-polygon represents a sequence of actions (DAFSA edges), these sequences are the candidate automatable routines. [6] Therefore, for each bond *b* in the RPST we extract the trivials that are direct children of *b* (see line 4), and we add each of these trivial children to the set of flat-polygons. We do the same for each rigid *r* in the RPST (see line 5). Whilst,

[6] Note that, a single action (a single DAFSA edge) is the simplest candidate automatable routine.

Algorithm 1: Flat-Polygon Detection

```
   input  : UIL 𝒰
1  DAFSA D ← generateDAFSA(𝒰);
2  RPST R ← generateRPST(D);
3  Set F ← ∅;
4  for b ∈ getBonds(R) do F ← extractTrivialChildren(R, b);
5  for r ∈ getRigids(R) do F ← extractTrivialChildren(R, r);
6  for p ∈ getPolygons(R) do F ← extractFlatPolygons(b);
7  return F;
```

for each polygon (p, see line 6) in the RPST we extract the flat-polygons as follows. If *all* the children of p are trivials, p is a single flat-polygon, otherwise, we split p into sub-polygons that have either: only rigids and/or bonds children (i.e. sub-polygons being sequences of rigids and/or bonds); or only trivial children (i.e. sub-polygons being sequences of trivials), these latter are the flat-polygons. Figure 3 shows the output of

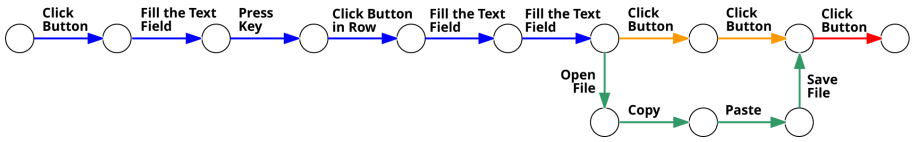

Fig. 3: Flat-polygon detection in the working example.

Algorithm 1 when the input UI log contains recordings of the two routine variants described in the running example. Note that we have multiple recordings of the two variants (i.e. UI log does not contain only the two routine traces pictured in Tables 1 and 2). The graph captured in the figure is the DAFSA of the UI log, whilst in blue, orange, green, and red the four flat-polygons detected and extracted.

3.3 Automatable Actions Detection

The second step of our approach focus on the discovery of the deterministic actions, i.e. the actions that can be automated. To do so, we execute Algorithm 2. This latter receives as input the DAFSA D and the set of flat-polygons detected in the previous step (F). For each flat-polygon f in the set F, we analyse each of its actions ($a = (\tau, P, V, \phi)$), see line 3 and 4. We retrieve all the values assigned to the parameters of the action a (for all the instances of a), and we create a map Π that associate to each parameter $p \in P$ all the values that the parameter assume in all the different routine traces containing the action a, $(p, \hat{V})$ (see line 6). [7] We need to collect all the values that each parameter p can assume because a is deterministic iff it is always possible to determine systematically its parameters values, this means that we must be able to compute the parameters values using constant or deterministic functions. The goal of Algorithm 2 is to identify, for each action of each flat-polygon, one function per action's parameter that allow us to deterministically compute the parameter value. A constant function is a function that assigns to a parameter always the same value, i.e. the parameter value is constant in all

[7] Note that $\hat{V}$ is a list of values and not a set, i.e. it can contain duplicates.

the instances of a. Whilst, for deterministic function, we mean a function that assigns the value to a parameter depending on the parameter values of actions executed before a. In particular, the deterministic functions we can discover are either based on *data transformation* or *substitution mappings*. We iterate on all the elements in Π, $(p, \hat{V})$, to identify these functions. First, we determine if the value of the parameter p is a constant function (see line 9), by checking if all values in $\hat{V}$ are equal.

Every time we discover a function for determining a parameter values, we add it to the list of the functions associated to the action (Ω) and we eliminate the parameter from Π (see lines 11- 12, 18- 19, 28- 29). Doing so, if after analyzing each parameter of a, Π is empty, it means that we found for each parameter at least one function that computes its values deterministically, and therefore a is deterministic. In such case, we add a and the set of the functions to determine its parameters' values (i.e. the action specification of a) to the map α, the output of Algorithm 2.

In our running example, an action with a constant parameter is *Press Key* (see Tables 1 and 2, action #3), where regardless of the routine trace the value of the parameter *Label* is always *ENTER*. On the other hand, if the parameter p is not a constant, we

Algorithm 2: Automatable Actions Detection

```
   input : DAFSA D, Flat-Polygons F
 1 Map α ← ∅;
 2 Set R ← ∅;
 3 for f ∈ F do
 4     for a ∈ f do
 5         Boolean transformation ← TRUE;
 6         Map Π ← getParametersValues(a);
 7         Set Ω ← ∅;
 8         for (p, V̂) ∈ Π do
 9             Function ω ← identifyAndGetConstantFunction(p, V̂);
10             if r ≠ null then
11                 Π ← Π \ (p, V̂);
12                 Ω ← Ω ∪ ω;
13             else
14                 for ρ^→i ∈ getPrefixRoutines(a,D) do
15                     R ← discoverDataTransformationFunctions(p, V̂, ρ^→i);
16                     if R = ∅ then transformation← FALSE;
17                 if transformation then
18                     Π ← Π \ (p, V̂);
19                     Ω ← Ω ∪ R;
20                 else
21                     Matrix X ← ∅;
22                     for ρ^→i ∈ getPrefixRoutines(a, D) do
23                         add rows generateMatrixRows(getActions(ρ^→i), V̂) to X;
24                     Set R ← findJripRules(X, 1.0);
25                     Decimal support ← 0;
26                     for r ∈ R do support ← support + getSupport(r);
27                     if support = 1.0 then
28                         Π ← Π \ (p, V̂);
29                         Ω ← Ω ∪ R;
30         if Π = ∅ then
31             α ← α ∪ (a, Ω);
32 return α;
```

check if each of its values in $\hat{V}$ is function of the previously executed actions' parameters' values. To do this, we collect all the routine traces prefixes ending with the action a (line 14). Each of these routine traces prefixes represents a routine trace variant that led to the execution of the action a, for each of this variants we try to discover the data transformation functions that allow us to compute the values of p. To discover these functions (line 15), we rely on the following two methods: i) we look for simple value-to-value dependencies (i.e. when the value of p always matches the value of a parameter of a previously executed action); ii) we apply Foofah, a data transfomation-by-example technique [10]; With the first method we can discover only value-to-value dependencies (the most common dependencies in RPA), hence, the deterministic function assigning a value to p would simply return the value of the matching action parameter. The second method, instead, can be used when no value-to-value dependencies are found, since Foofah can discover more complex data transformations. Foofah takes two series of data values, one called input and one called output. The input data series is the array containing all the values of one parameter of an action executed before a, whilst, the output data series is $\hat{V}$. Foofah tries to detect functions that can determine the output from the input. Consequently, Foofah allow us to discover data transformations where: each element in the output is equal to an element in the input; or equal to a sub-string thereof; or equal to a concatenation of multiple elements in the input. Despite Foofah allow us to discover more complex data transformation functions, and detect more deterministic actions, its performance greatly affects the performance of our entire approach because: i) it does not scale well for large input data series; and ii) we need to use Foofah for each parameter of a and each parameter of each action executed before a (until we find a data transformation function). To partially address Foofah scalability limitation, we perform a random sampling on the input and output data series, and input these latter to Foofah. Indeed, if Foofah cannot identify a data transformation over the subsets of the complete input and output data series, it means that it could not identify any data transformation also for the complete input and output data series.

As an example of the benefits brought by Foofah, in our running example, for the routine in Tab. 1, the action #8 that copies only part of the student address to the Excel file can be detected as deterministic only by using Foofah.

Finally, if no data transformation functions are found for the parameter p, we try to discover functions based on substitution mappings (line 21 to 26). A function based on substitution mappings is a function that maps a set of values to another (different) set of values.For instance, a substitution mapping function can assign the value v_x to p every time that a parameter value of another action (executed before a) is equal to v_y.

To find these substitution mapping functions we use JRipper [4], an implementation of a propositional rule learner. JRipper takes as input a matrix where each row is an array of *data values* and a *label*. In our case, the data values in each row corresponds to all the values of all the parameters of all the actions executed before a in a given routine trace, whilst the label in each row matches the value of p (in the given routine trace). JRipper analyses the matrix and returns the set of rules, of which we retain those having confidence 1.0 (set R, see line 24), since such rules are the only that can be considered deterministic. Each of this rules allow us to deterministically assign a value to the parameter p during a given routine execution (captured in one or more rows of the matrix). Therefore, if all the rules with confidence 1.0 can cover all the routine executions captured as the matrix rows, it means that we can use this set of

rules to deterministically assign a value to the parameter p in all the possible routine executions. To verify that the the set of rules with confidence 1.0 cover all the routine executions captured in the matrix rows, we check that the sum of the rules' supports is equal to 1.0 (see lines 26 and 27).

In our running example, a substitution mapping function is detected to determine the value of the parameter *Label* in the last *Click Button* action (in both Table 1 and 2). Indeed, every time the country value is equal to *Australia* the *Label* value of the last *Click Button* action is *confirm backup*. Whilst, every time the country value is not *Australia* the *Label* value of the last *Click Button* action is *confirm*.

3.4 Routine Specifications Detection

In the last step we identify, by applying Algorithm 3, the set of routine specifications. The algorithm receives as input the DAFSA D, the set of flat-polygons F, and the map α containing the actions specifications that we discovered with Algorithm 2. First, we extract the automatable routines from the flat-polygons, line 2. Precisely, given each flat-polygon, we check that each of its actions is deterministic (i.e. the action is in α), and we extract all the sequences of deterministic actions not interrupted by non-deterministic ones. This means that from a flat-polygon we can extract one or more automatable routines. Once detected the automatable routines (set S), we have to discover for each of them the activation condition. An activation condition of an automatable routine in S is the trigger that determines the start of the routine execution. It consists of: i) a triggering action, i.e. an action executed just before the first action of the automatable routine; and ii) a boolean condition that must be valid at the completion of the triggering action. The boolean condition can be: i) a function of parameter values of the actions executed before the triggering action (this included); or ii) can be based only on the completion of the triggering action, i.e. the boolean condition is the completion of the triggering action. We call the first case a *data-based activation condition*, whilst the second case a *trivial activation condition*.

We discover each automatable routine (ρ) activation condition as follows. First, we collect all the routine traces prefixes ending with the action a_1, being this latter the first action of the automatable routine (line 5). Then, for each of these prefixes, we collect its actions into the set B (excluding a_1, see line 6), B will contain all the actions that can be executed before a_1. We note that the execution of any of these routine traces prefixes is not a sufficient condition for the execution of a_1 (and the automatable routine thereof), because any of these prefixes could also lead to the execution of an action different than a_1 (i.e. a_1 is an outgoing edge of a decision point of the DAFSA). To analyse for what prefixes actions' parameters values a_1 is executed or not, we can use again JRipper, similarly to Algorithm 2. In this case the JRipper input is a matrix X where: each row is the array containing all the parameters values of all the actions executed before a_1 in a given routine trace, plus a boolean label set to *true* if the automatable routine was executed within the given routine trace, *false* otherwise. To build the matrix X, we collect all the paths of the DAFSA[8] (see line 8) and, for each of them ($\hat{\rho}$), we take the actions that are contained in B (see set Q, line 9). If the set Q is not empty, we add to the matrix X a row containing all the parameters values of all the actions in Q

[8] We remind that a path of the DAFSA corresponds to a routine trace in the UI log.

Algorithm 3: Routine Specifications Detection

```
   input  : Map α, Flat-Polygons F, DAFSA D
 1 Set Ξ ← ∅;
 2 Set S ← extractAutomatableFlatPolygons(α, F);
 3 for ρ ∈ S do
 4     Set B ← ∅;
 5     Action a1 ← ρ[1];
 6     for ρ^→i ∈ getPrefixRoutines(a1,D) do B ← B∪ getActions(ρ^→i−1);
 7     Matrix X ← ∅;
 8     for ρ̂ ∈ getPaths(D) do
 9         Set Q ← getActions(ρ̂) ∩B;
10         if Q ≠ ∅ then
11             Boolean label ← ρ ⊆ ρ̂;
12             add rows generateMatrixRows(Q, label) to X;
13     Set R ← findJripRules(X, 1.0);
14     Boolean activationConditionFound ← TRUE;
15     for r̂ ∈ getRowsWithLabel(X, TRUE) do
16         if ¬ existRule(R, r̂) then activationConditionFound ← FALSE;
17     if activationConditionFound then
18         Ξ ← Ξ∪ (ρ, R);
19
20     else
21         if hasTrivialCondition(ρ) then
22             R ← generateTrivialCondition(ρ);
23             Ξ ← Ξ∪ (ρ, R);
24         else
25             if |ρ| > 1 then
26                 R ← generateTrivialCondition(ρ^2→);
27                 Ξ ← Ξ∪ (ρ^2→, R);
28 return Ξ;
```

and we set the boolean label in the row as *true*, if the automatable routine ρ is contained in $\hat{\rho}$, otherwise as *false* (see lines 11 and 12).

Once we built the matrix X, we input it to JRipper, which outputs the set of rules, of which we retain those having confidence 1.0 (set R, see line 13) Each of these rules tell us which parameters values of all the actions in B triggered the automatable routine. Then, we filter the matrix X retaining only the rows having the boolean label *true* and, similarly to Algorithm 2, we check that the rules discovered with JRipper can cover all the rows of this filtered matrix, line 16. To perform this check, we cannot rely on the support of the rules, like in Algorithm 2, because we are not interested in covering all the rows of the matrix X, but only those having the boolean label *true* (i.e. we used the full matrix to discover the rules, but we want the rules to cover only the cases when the automatable routine ρ is executed). If the rules discovered by JRipper cover all the rows of X capturing an execution of ρ (line 17), it means that we found a data-based activation condition, which is represented by the set of rules discovered with JRipper. [9] When we find an activation condition for an automatable routine, we create the routine specification (the tuple: automatable routine and activation condition) and we add it to the set of routine specifications Ξ, see line 19. On the other hand, if a data-based activation condition cannot be found with JRipper (line 22), we check if exists a trivial activation condition, i.e. if ρ can be triggered by the completion of an

[9] Note that, the set of rules take into account also the triggering action.

action preceding a_1. If we find such a trivial activation condition, we use it to create the routine specification for ρ (lines 22 and 23). Otherwise, if ρ is not a single-action routine (line 25), we can always use a_1 to generate a trivial activation condition for the subroutine $\rho^{2\rightarrow}$. Indeed being ρ a sequence of deterministic actions, it follows that after the execution of its first action (a_1), the successive actions will be executed as well.

4 Evaluation

We conducted an experiment to assess the ability of our approach to correctly discover all the automatable (sub)routines recorded in a set of UI logs. To this end, we generated nine artificial UI logs (from Coloured Petri Nets, CPNs), each log containing a different number of automatable (sub)routines of varying complexity,[10] and used the characteristics of these routines (position and actions within) as a ground truth for our experiment. These logs emulate a controlled recording environment where user tasks are performed without noise (i.e. events that capture actions that are irrelevant to the task are not present).

For testing purposes, we packaged the implementation of our method as a Java command-line application, [11] and executed it on a PC with Intel Core i7-6600U@2.60GHz CPU with 12GB RAM, running Ubuntu 16.04 LTS (64-bit) with 8GB RAM and JVM 11 (4GB RAM). We conducted all tests using both methods to discover data-transformation functions, i.e. with and without Foofah.

4.1 Test Case Generation

We generated nine artificial UI logs by simulating nine CPNs designed with the CPN Tool [9]. The first six CPNs are simple and represent common real-life scenarios, capturing clear routines with a specific goal. CPN1 represents the following sequence of actions: the user opens a random file, opens a specific webpage, logs in with his credentials (assumed to be always the same and correct), awaits the response from the server, and then begins to copy data from the Web page to the opened file. All these actions are automatable, except the first one, since the input file is chosen randomly. CPN2 is a variant of CPN1 and it includes the handling of an error during the login action, i.e. the user enters wrong credentials. CPN3 is a further extension of the task captured in CPN2, where the user unsuccessfully repeats the login actions until they decide to quit the procedure. CPN4 is derived from CPN1, but in this task we injected non-deterministic actions among the automatable ones. We used these non-deterministic actions to perturb the fully automatable task in CPN1. These are: random button clicks (i.e. non data-driven clicks) and user data inputs (i.e. the login credentials are different in each routine trace). The number of non-deterministic actions increases in CPN5, where only 16% of the total actions is automatable, and no automatable routines are captured, i.e.

[10] The CPNs and the logs used for our evaluation are available at `https://doi.org/10.6084/m9.figshare.7850918.v1`

[11] The software is available at `http://apromore.org/platform/tools`, *Automatable Routines Discoverer* package. The source code can be found at `https://github.com/apromore/RPA_AutomatableRoutinesDiscoverer`

there are no two consecutive automatable actions. CPN6 is the running example presented in Section 2, which has a balanced number of automatable and non-automatable actions.

The last three CPNs have the highest complexity and the routines they represent are not easily interpretable (e.g. they do not follow a routine goal). We decided to include these latter CPNs to evaluate the robustness of our approach in the case of complex scenarios. CPN7 has only 25% of automatable actions, which are intertwined with the non-automatable actions. CPN8 extends CPN7 by adding a long deterministic subroutine to be executed when a specific condition is met. CPN9 is the most complex case; it merges CPN5 and CPN6 and captures the situation in which a user first inputs several data and then decides the sequence of actions to perform based on the input data. Using this latter CPN we can assess our approach's ability to discover activation rules based on long-dependencies, i.e. dependencies between non sequential actions.

Table 3 reports the structural complexity of each CPN in terms of size, control flow complexity, and structuredness, as well as the statistics of the UI logs simulated from the CPNs. Precisely, we reported for each UI log the number of routine variants, the number of distinct routine traces, and the number of actions recorded.

	CPN			UI Logs		
ID	**Size**	**CFC**	**Struct.**	**#Routines**	**#Traces**	**#Actions**
1	15	0	1.00	1	100	1400
2	20	2	1.00	2	1000	14804
3	20	3	1.00	6	1000	14583
4	18	4	1.00	1	100	1400
5	18	11	1.00	12	1000	8775
6	16	2	1.00	2	1000	9998
7	24	10	0.29	7	1500	14950
8	39	11	0.56	8	1500	17582
9	65	24	0.83	18	2000	28358

Table 3: Characteristics of the CPNs and UI Logs.

4.2 Results

Table 4 shows the results of our experiment. For each log generated from the corresponding CPN, we report the number of distinct automatable actions (AA) recorded, their percentage on the total number of distinct actions, the length of the longest sequence of AA (i.e. longest autom. routine length, LRL), the number of distinct AA discovered and the LRL discovered with our approach (both with and without Foofah). We note that w/ Foofah, our approach could discover all the AA as well as the LRL, except for log L3, which we discuss later. However, when disabling Foofah, we could not detect some data transformations (i.e. the non value-to-value transformations), which were necessary to identify some AA. Consequently, our approach w/o Foofah could not identify all the AA (as per L3, L6, L8, and L9), and in some cases (see L4, L6 and L9) the LRLs were shorter than those discovered w/ Foofah. Despite the Foofah variant of our approach allow us to detect more complex data transformations, and therefore discover more AA and longest autom. routines, Foofah brings an overhead in the execution time. Indeed, when enabling Foofah, our approach becomes up to 50x slower (see L9). This is due to how Foofah looks up for data transformations, its exhaustive

UIL	Recorded AA(#)	Recorded AA(%)	Recorded LRL	AA Discovered (w/ Foofah)	AA Discovered (w/o Foofah)	LRL Discovered (w/ Foofah)	LRL Discovered (w/o Foofah)	Exec. Time (s) (w/ Foofah)	Exec. Time (s) (w/o Foofah)
L1	13	92.9	13	13	13	13	13	3.0	2.8
L2	16	84.2	5	16	16	5	5	61.1	5.2
L3	17	94.4	7	15	15	6	6	224.8	7.6
L4	8	47.1	4	8	7	4	3	79.2	2.7
L5	2	16.7	1	2	2	1	1	49.6	9.4
L6	9	69.2	4	9	8	4	2	80.0	4.5
L7	4	25.0	2	4	4	2	2	279.7	8.2
L8	18	60.0	13	18	15	13	13	282.2	10.9
L9	24	68.6	5	24	22	4	3	935.2	19.3

Table 4: Experimental results on the artficially generated UI logs.

search tries all the possible combinations of data transformations before declaring that no data transformation occurred.

The only log where the approach failed to discover two routines is log L3, which contains loops. The reason for this limitation is that the approach takes a DAFSA (which is acyclic) as a starting point and the DAFSA does not capture the notion of loop. Repetition in a DAFSA shows up in the form of a polygon followed by a branching in which one of the polygons is identical to the previous polygon (and this pattern can be repeated multiple times). Our approach does detect that the polygon inside the body of the loop is a routine and it discovers its activation condition, but it fails to discover the activation condition of the task that follows the exit point of the loop, particularly when this condition depends on the number of times the loop has been executed.

5 Related Work

There are resemblances between the discovery of automatable routines from UI logs and Automated Process Discovery (APD) from event logs [3], stemming from the fact that the inputs have similar structure. However, APD focuses on discovering control-flow models, without data flow. Some approaches enhance discovered process models with branching conditions [5], but we are not aware of any work on APD that discovers data transformations. Moreover, APD approaches are not suitable for automable routine discovery because they generalize the behavior in the log, i.e. they produce models that generate traces not observed in the log. Most of these algorithms also under-approximate the log's behavior, i.e. they intentionally produce models that do not perfectly fit the log. In contrast, we seek to discover only sequences of actions that have been observed in a UI log. Automating a sequence of actions that has never been observed is risky, because the cost of letting a software bot do something that should not be done is high (e.g. it may introduce spurious data into a system leading to costly mistakes and time-consuming corrective actions). This is the reason why our approach uses a lossless representation of event logs (DAFSAs) to discover candidate automatable routines, as opposed to an automatically discovered process model.

Recent related work in APD deals with the problem of discovering process models from low-level event logs [8]. In this context, a low-level event log is one where each event refers to a step of a task in a process, e.g. a task "Contact customer" is captured via several events corresponding to steps "Retrieve the customer's contact details from CRM system", "call customer", etc. The goal of [8] and related studies is to group low-level events into coarser-grained ones in order to discover conceptual models, as opposed to automatable routines as we do.

Another related family of techniques is sequence mining [2], where the goal is to discover frequent patterns in collections of sequences. In contrast to sequence mining, automatable routine discovery takes as input UI logs consisting of actions with parameters, as opposed to sequences of symbols.

Automatable routine discovery is also related to Web usage mining and UI log mining. Web usage mining seeks to discover and analyze sequential patterns in Web data, such as Web server logs capturing user interactions with Web apps [16]. Analyzing such data can help to optimize the functionality of Web apps, optimize their navigation structure, and provide personalized content to users [11].

Research proposals in UI log mining, such as TaskTracer and TaskPredictor, have tackled the problem of analyzing UI logs generated by desktop applications to identify the current task performed by a user and to detect switches between tasks [15, 14, 7]. These techniques do not deal with the problem of identifying routines. More closely related is the work in [6], which proposes a technique to extract frequent sequences of actions from noisy UI logs. However, this technique does not extract automatable routines because it does not discover data transformations nor activation conditions.

6 Conclusion

We introduced an approach to discover automatable routines from UI logs. Each such automatable routine is characterized by an activation condition and a sequence of action specifications, each associated with data transformations that compute an action's parameters values from those of previously observed actions. An experimental evaluation allowed us to validate that the approach can re-discover repetitive routines synthetically injected in a UI log. The evaluation also allowed us to compare two alternative methods to discover data transformations: one based on an algorithm to discover one-to-one dependencies and the other based on Foofah, which can discover complex (1-to-N, N-to-1, and N-to-N) dependencies. The evaluation showed that the one-to-one dependency discovery method scales up to relatively large logs, while putting into evidence scalability limitations of the Foofah alternative.

The evaluation highlighted one of the limitations of the approach: its inability to discover activation conditions of routines that immediately follow the exit point of a loop, particularly when this condition depends on the number of executions of the loop. This is due to the fact that DAFSAs do not capture loops. A possible approach to address this limitation is to extend the DAFSA with explicit loops. This would require us to detect those loops in the first place, for example using tandem repeat detection algorithms. This extension should be such that it allows us to reason about the number of occurrences of a loop in a routine trace, so that this information can be used when discovering the activation condition (i.e. the condition under which the loop is exited).

Another limitation of the approach is its inability to deal with noise. The presence of an event in a routine trace that is not related to the trace (e.g. an error) leads to the polygon capturing that routine being broken down into two flat-polygons. As a result, our approach will either discover only sub-routines of an otherwise automatable routine, or not discover the routine at all. Some of the noise could be removed via simplification rules that take advantage of the idempotence of some UI operations (e.g. if a user mistakenly clicks twice the same cell consecutively, this can be reduced to a

single click). But more generally, it may require tailor-made noise filtering techniques. Designing such techniques is an avenue for future work.

In the embodiment of the approach presented in this paper, we used RIPPER to discover activation conditions and Foofah to discover complex data transformations. Experimenting with alternative methods to discover activation conditions and data transformations is another direction for future work.

Acknowledgements. This research is funded by the Australian Research Council (DP180102839), the Estonian Research Council (IUT20-55), and the European Research Council (project "PIX").

References

1. W. M. P. van der Aalst, M. Bichler, and A. Heinzl. Robotic process automation. *BISE*, 60(4), 2018.
2. R. Agrawal and R. Srikant. Mining sequential patterns. In *ICDE*. IEEE, 1995.
3. A. Augusto, R. Conforti, M. Dumas, M. La Rosa, F. Maria Maggi, A. Marrella, M. Mecella, and A. Soo. Automated discovery of process models from event logs: Review and benchmark. *TKDE*, 31(4), 2019.
4. W. W. Cohen. Fast effective rule induction. In *ICML*. Morgan Kaufmann, 1995.
5. M. de Leoni, M. Dumas, and L. García-Bañuelos. Discovering branching conditions from business process execution logs. In *Proceedings of FASE*. Springer, 2013.
6. Himel Dev and Zhicheng Liu. Identifying frequent user tasks from application logs. In *Proceedings of IUI 2017*, pages 263–273. Springer, 2017.
7. A. N. Dragunov, T. G. Dietterich, K. Johnsrude, M. R. McLaughlin, L. Li, and J. L. Herlocker. Tasktracer: a desktop environment to support multi-tasking knowledge workers. In *IUI*. ACM, 2005.
8. B. Fazzinga, S. Flesca, F. Furfaro, and L. Pontieri. Process discovery from low-level event logs. In *CAiSE*. Springer, 2018.
9. K. Jensen, L. M. Kristensen, and L. Wells. Coloured petri nets and cpn tools for modelling and validation of concurrent systems. *STTT*, 9(3-4), 2007.
10. Z. Jin, M. R. Anderson, M. J. Cafarella, and H. V. Jagadish. Foofah: Transforming data by example. In *SIGMOD*. ACM, 2017.
11. B. Liu. Web usage mining. In *Web Data Mining: Exploring Hyperlinks, Contents, and Usage Data*. Springer, 2007.
12. A. Polyvyanyy, J. Vanhatalo, and H. Völzer. Simplified computation and generalization of the refined process structure tree. In *Web Services and Formal Methods*. Springer, 2011.
13. D. Reißner, R. Conforti, M. Dumas, M. La Rosa, and A. Armas-Cervantes. Scalable conformance checking of business processes. In *OTM Confederated International Conferences On the Move to Meaningful Internet Systems"*. Springer, 2017.
14. J. Shen, J. Irvine, X. Bao, M. Goodman, S. Kolibaba, A. Tran, F. Carl, B. Kirschner, S. Stumpf, and T. G. Dietterich. Detecting and correcting user activity switches: algorithms and interfaces. In *IUI*. ACM, 2009.
15. J. Shen, L. Li, and T. G. Dietterich. Real-time detection of task switches of desktop users. In *IJCAI*, 2007.
16. J. Srivastava, R. Cooley, M. Deshpande, and P. Tan. Web usage mining: Discovery and applications of usage patterns from web data. *SIGKDD Explor. Newsl.*, 1(2), 2000.