An Activity Instance Based Hierarchical Framework for Event Abstraction

Chiao-Yun Li*, Sebastiaan J. van Zelst†, and Wil M.P. van der Aalst‡
Fraunhofer FIT, Birlinghoven Castle, Sankt Augustin, Germany
Process and Data Science (PADS) Chair, RWTH-Aachen University, Germany
Email: *chiao-yun.li@fit.fraunhofer.de, †sebastiaan.van.zelst@fit.fraunhofer.de, ‡wvdaalst@pads.rwth-aachen.de

***Abstract*—Process mining allows one to analyze and extract knowledge from *event data*, i.e., records of process executions stored in information systems. Most process mining techniques are directly applied to the data as recorded in the system. Applying automated process discovery techniques, i.e., a core process mining technology, directly on such data yields complex process models describing millions of different execution paths. Other techniques applied to such discovered process models and system-level data, e.g., conformance checking or performance analysis techniques, often generate complex and over-detailed results. The results obtained by directly applying process mining techniques on system-level data are, therefore, hard to interpret by a human analyst and greatly differ from the business level. Therefore, in this paper, we propose a generic hierarchical framework for event abstraction. We formalize the framework, which uses the notion of activity instances as an input and allows for hierarchical abstraction of event data. In addition, we propose an instantiation of the framework, which describes two key functions of the framework, i.e., abstract concept identification and abstract entity extraction. The framework, together with the instantiation, is evaluated both quantitatively and qualitatively. The experiments show that, without compromising the quality of results, the abstraction allows users to easier analyze a process.**

***Index Terms*—process mining, event abstraction, earth mover's distance**

I. Introduction

Organizations strive to improve their (well-defined) business processes to achieve competitive advantage. The effective improvement of processes requires an accurate understanding of the actual execution of these processes. To obtain such insights into the processes executed, *process mining* [1] allows one to analyze and exploit the information captured in *event data*, i.e., the records of executions performed in the context of a process. *Process models* visualize/explain the behavior of a process such that human analysts can interpret its described behavior. As such, many process mining techniques use process models, e.g., one may automatically discover a process model that describes the behavior of the process recorded in the event data by using *process discovery* algorithms.

According to [2], a human analyst can only process and interpret a limited number of tasks or execution paths in a process model. Hence, to facilitate the general interpretability of a process model, one typically applies *abstraction principles* when modeling a process. For example, consider a medical process. Rather than modeling all activities performed by the front-desk employee to register a patient, one typically models the *patient registration* as a single activity. However, information systems typically record all activities performed (and often additional lower-level system calls and functions). Hence, applying process mining directly on the data captured yields overly detailed and complex process models describing hundreds of activities and millions of execution paths.

To increase the interpretability of process mining results, *event abstraction techniques* are typically applied [3]. Such techniques pre-process the event data by grouping the recorded "low-level events" into a higher-level activity notion, i.e., activities at the business level. Process models discovered based on the *abstracted event log* are of a higher level of abstraction and thus allow analysts to gain a better end-to-end understanding of the process. For example, identifying the deviations that have a significant impact on the business is easier with a limited number of concepts in a process model discovered by applying existing process discovery techniques.

Nevertheless, some existing event abstraction techniques require explicit domain knowledge to perform abstraction. Furthermore, most techniques are designed to be applied on *sequences of events*. After applying the techniques once, the resulting event log describes a partial order of activity instances. Hence, if the abstracted event log is still too fine-grained, the techniques cannot be further applied on the abstracted log. Therefore, in this paper, we propose a *hierarchical framework* for event abstraction which allows for the integration of several existing event abstraction techniques. The framework is based on the notion of *activity instances*, which differs from the typical concept of *events* used in process mining. As such, this work can be seen as an effort to standardize an end-to-end framework for event abstraction that allows for various instantiations. We propose an accompanying instantiation of our framework, characterizing the two core functions of the framework, i.e., an abstract concept identification and entity extraction strategies. We evaluate the instantiation of our framework both quantitatively and qualitatively. The results indicate that, without compromising the quality of the models discovered, the abstracted process is easier to comprehend.

The remainder of this paper is structured as follows. We present preliminaries in Section II and provide a running example in Section III. The framework and its instantiation are introduced in Section IV and V, respectively. We evaluate our approach in Section VI. Section VII gives an overview of the related work. Limitations are discussed in Section VIII, followed by the conclusion in Section IX.

TABLE I: Running Example: An activity instance log L.

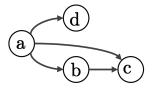
cid	*id*	*activity*	*start time*	*complete time*
1	1	apply loan (a)	2021.01.01 09:02	2021.01.01 09:02
1	2	check credit (c)	2021.01.04 09:30	2021.01.06 11:32
1	3	examine application (e)	2021.01.04 09:30	2021.01.07 09:45
1	4	decide (d)	2021.01.09 13:47	2021.01.10 10:20
1	5	decide (d)	2021.01.10 16:00	2021.01.12 08:48
1	6	decide (d)	2021.01.13 15:00	2021.01.15 09:20
1	7	notify applicant (n)	2021.01.16 10:23	2021.01.16 10:23
2	8	apply loan (a)	2021.01.01 10:06	2021.01.01 10:06
2	9	examine application (e)	2021.01.06 08:50	2021.01.10 08:45
2	10	check credit (c)	2021.01.07 11:16	2021.01.09 17:38
2	11	decide (d)	2021.01.12 11:12	2021.01.14 14:00
2	12	examine application (e)	2021.01.14 14:02	2021.01.14 14:00
2	13	decide (d)	2021.01.15 09:24	2021.01.18 10:28
2	14	notify applicant (n)	2021.01.20 16:07	2021.01.20 16:07

II. Preliminaries

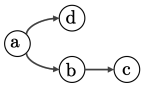
Partial Orders: Given an arbitrary set X, we write $\mathcal{P}(X)=\{X'|X'\subseteq X\}$ to denote its powerset. $|X|$ denotes the number of elements in X. A relation $\prec$ over X ($\prec\subseteq X\times X$) is a *strict partial order over* X, written $(X,\prec)$, if and only if $\forall x\in X(x\not\prec x)$ ($\prec$ is *irreflexive*), $\forall x_1,x_2\in X(x_1\prec x_2 \implies x_2\not\prec x_1)$ ($\prec$ is *anti-symmetric*), and $\forall x_1,x_2,x_3\in X(x_1\prec x_2\wedge x_2\prec x_3 \implies x_1\prec x_3)$ ($\prec$ is *transitive*). Given $(X,\prec)$ and $X'\subseteq X$, we let $(X',\prec')$ be the *induced partial order* on X', where $\prec'=\prec\cap(X'\times X')$. Given $x\in X$ and $(X,\prec)$, we define $pred_X(x,\prec)=\{x'\in X|x'\prec x\}$ and $succ_X(x,\prec)=\{x'\in X|x\prec x'\}$ for the predecessors and successors of x in X. In the remainder, we write $pred(x)$ and $succ(x)$ if X is clear from the context (or not relevant). Given $succ(x)$, we write $succ^k(x)$ for the successors that are at the *largest* distance $d\leq k$ of x in $succ(x)$. For example, given the partial order in Figure 1a, $succ^1(a)=\{b,d\}$ and $succ^2(a)=\{b,c,d\}$. Similarly, $pred^k(x)$ refers to the predecessors of x of depth $d\leq k$. Figure 1a visualizes a partial order as a graph. For simplicity, we often visualize the *transitive reduction*[1] of a partial order as in Figure 1b, where the arc (a,c) is removed yet c is still reachable from a.

Event Data: Most process mining techniques require a log describing records of executions of processes as an input. In this paper, we use the notions of *activity instance* (a record of a task executed in a process), *case* (a process instance), and *activity instance log* (a collection of process instances). Table I sketches a synthetic example log. Every row corresponds to an *instance of an activity* executed in the context of a *case*. For example, the first row indicates that the *apply loan* activity (short-hand notation: activity a) was executed at *9:02 AM, January 1st, 2021* and the second row indicates that *check credit* (short-hand notation: activity c) was executed from *9:30 AM, January 4th, 2021* until *11:32 AM, January 6th, 2021*. Each activity instance has a unique *id* (the second column). The case identifier (the first column *cid*) allows us to identify for which case the activity instance was executed. For example, the first 7 activity instances are executed for a case identified by case-identifier 1. We formalize the notions of an activity instance, a case, and an activity instance log as follows.

[1]A transitive reduction of a directed graph is a directed graph with the same vertices and as few edges as possible maintaining the same reachability.



(a) The partial order.



(b) The *transitive reduction* of the partial order.

Fig. 1: Representation of a partial order over $\{a,b,c,d\}$.

Definition 1 (Activity Instance). *An activity instance describes the (historical) execution of an activity, which is described by a set of attributes. We let $\mathcal{A}$ be the universe of activity instances, $\mathcal{U}_{act}$ the universe of activities, and $\mathcal{U}_{ts}$ the universe of timestamps. The following attributes of $a\in\mathcal{A}$ can be obtained by the projection functions:*

- $\pi_{id}\colon \mathcal{A}\to\mathbb{N}^+$ *s.t.* $\forall a,a'\in\mathcal{A}(\pi_{id}(a)=\pi_{id}(a') \implies a=a')$,
- $\pi_{act}\colon \mathcal{A}\to\mathcal{U}_{act}$, *where* $\pi_{act}(a)$ *denotes the activity of* a,
- $\pi_{st}\colon \mathcal{A}\to\mathcal{U}_{ts}$, *where* $\pi_{st}(a)$ *denotes the start time of* a,
- $\pi_{ct}\colon \mathcal{A}\to\mathcal{U}_{ts}$, *where* $\pi_{ct}(a)$ *denotes the completion time of* a *s.t.* $\pi_{st}(a)\leq\pi_{ct}(a)$.

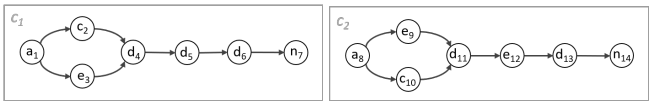
Definition 2 (Case, Activity Instance Log). *Let $\mathcal{A}$ be the universe of activity instances. A case $c\subseteq\mathcal{A}$ describes a set of activity instances executed in the context of a process instance. A case c explicitly defines a partial order $(c,\prec)$, i.e., $\forall a,a'\in c\,(a\prec a'\Leftrightarrow\pi_{ct}(a)<\pi_{st}(a'))$. Let $\mathcal{C}\subseteq\mathcal{P}(\mathcal{A})$ denote the set of all possible cases. An activity instance log $L\subseteq\mathcal{C}$ is a set of cases where $\forall c,c'\in L(c\cap c'=\emptyset)$. The set of all the possible logs of a process is denoted as $\mathcal{L}\subseteq\mathcal{P}(\mathcal{C})$.*

In the remainder, we refer to the *activity instance class* of said activity instances, e.g., the activity instance class of instances with id 4, 5 and 6 in Table I is *decide* (d). The *valid* classes of a log L are denoted as $\alpha(L)=\bigcup_{c\in L}\{\pi_{act}(a)|a\in c\}$.

III. Running Example

In this section, we introduce a running example which we use in this paper to illustrate and explain the framework. Consider a loan application process. After an applicant submits an application for a loan, the bank checks the applicant's credit and examines the application. Depending on the amount applied for and the information of the applicant, the application may be examined again and go through several underwriters. The bank notifies the applicant of the final decision, i.e., approval or rejection of the application. Table I shows an activity instance log L for the process described, where each row represents an activity instance with the corresponding attributes. For example, for case 1 (written as c_1), $c_1=\{a_1,c_2,e_3,d_4,d_5,d_6,n_7\}$[2]. Since each case defines a partial order over its activity instances, the activity instance log in Table I can be seen as a collection of partial orders. A graphical representation of L, i.e., represented as a collection

[2]For simplicity, given an activity instance a where $\pi_{id}(a)=k$ and $\pi_{act}(a)=b$, we write b_k in the rest of the paper.



(a) Case 1 (b) Case 2.

Fig. 2: Graphical representation of L (for simplicity, the example cases are depicted by the transitive reduction of the partial order defined on the activity instances.).

of partial orders, is shown in Figure 2, where each node represents an activity instance labeled with the abbreviation of its activity name and its id as subscript.

Table II shows an abstracted log after applying the proposed framework on L. The activity classes c, e, and d are abstracted to an *abstraction class* $C1$ at hierarchy level 1. Activity instances a and n form singleton abstraction classes at hierarchy level 1. In c_1, all the activity instances related to $C1$ are grouped together as an instance of $C1$. In c_2, we extract two instances of $C1$, $\{e_9, c_{10}, d_{11}\}$ and $\{e_{12}, e_{13}\}$. The collection of the cases in Table II is a log at hierarchy level 1. Finally, we apply the framework on the log abstracted and compute a log at hierarchy level 2 as shown in Table III, where all the classes at level 1 are grouped as a single class at level 2.

IV. Framework

In this section, we introduce the proposed framework. We explain the framework using the running example introduced in Section III and define the input and output for each component in the framework.

Figure 3 presents a schematic overview of the framework. A hierarchy of abstractions is obtained by iteratively applying the framework. In each iteration $i{\in}\mathbb{N}^+$, first, we discover the applicable *abstraction classes* at the next hierarchy level, i.e., at level $i{+}1$ (represented by abstraction oracle function φ). The corresponding instances of each abstraction class are extracted by the function δ. In the log at level i, the detected instances of an abstraction class, are replaced by instances of the abstracted class, i.e., generating the log at level $i{+}1$. For example, in L_1 in Table II, the instances c_2, e_3, d_4, d_5, d_6 of L in Table I are collapsed into a single instance of class $C1$ (instance id 2 in Table II). The same holds for instances e_9, c_{10}, d_{11} and instances e_{12} and d_{13}, i.e., instances 5 and 6, respectively in Table II. For each abstraction class, a sublog (*class log*) describing all the instances of the class is created.

TABLE II: A log L_1 at hierarchy level 1 abstracted from L, where $C1{=}\{c, d, e\}$.

cid	*id*	*class*	*instances*	*start time*	*complete time*
1	1	$\{a\}$	$\{a_1\}$	2021.01.01 09:02	2021.01.01 09:02
1	2	$C1$	$\{c_2, e_3, d_4, d_5, d_6\}$	2021.01.04 09:30	2021.01.15 9:20
1	3	$\{n\}$	$\{n_7\}$	2021.01.16 10:23	2021.01.16 10:23
2	4	$\{a\}$	$\{a_8\}$	2021.01.01 10:06	2021.01.01 10:06
2	5	$C1$	$\{e_9, c_{10}, d_{11}\}$	2021.01.06 08:50	2021.01.14 14:00
2	6	$C1$	$\{e_{12}, d_{13}\}$	2021.01.14 14:02	2021.01.18 10:28
2	7	$\{n\}$	$\{n_{14}\}$	2021.01.20 16:07	2021.01.20 16:07

TABLE III: A log L_2 at hierarchy level 2, abstracted from L_1, where $C2{=}\{\{a\}, \{n\}, C1\}$.

cid	*id*	*class*	*instances*	*start time*	*complete time*
1	1	$C2$	$\{\{a_1\}, C1_2, \{n_7\}\}$	2021.01.01 09:02	2021.01.16 10:23
2	2	$C2$	$\{\{a_8\}, C1_5, C1_6, \{n_{14}\}\}$	2021.01.01 10:06	2021.01.20 16:07

For example, class $C1$ describes activity instance classes c, d, e. Based on L, a class log L_{C1} is constructed, containing activity instances $2\ldots6$ for c_1 and instances $9\ldots13$ for c_2. Similarly, for singleton classes $\{a\}$, class logs containing activity instances 1 and 8 are created. Thus, the underlying behavior recorded for activity instance 2 in Table II, i.e., the instance of $C1$, is described by activity instances $2\ldots6$ in Table I, captured in the previously described class log L_{C1}.

The framework's output, i.e., a log at level $i{+}1$, is computed and can be used as the input for the next iteration. The process repeats until the level specified by the user. The total collection of the results obtained at each iteration forms a hierarchical abstraction of the input data.

First, we discover the applicable classes at each level. If we consider the activities recorded in the system as abstraction level 0, a class at level $i{>}0$ is simply a collection of classes at level $i{-}1$. The notion of class hierarchy is as follows.

Definition 3 (Hierarchy of Classes). *Let $\mathcal{U}_{act}$ be the universe of process activities. The universe of classes of a process at level $i{\in}\mathbb{N}$, i.e., $\mathcal{U}^i_{class}$, is recursively defined as $\mathcal{U}^i_{class}{=}\mathcal{U}_{act}$ if $i{=}0$ and $\mathcal{U}^i_{class}{=}\mathcal{P}(\mathcal{U}^{i-1}_{class})\backslash\{\emptyset\}$ for $i{>}0$.*

In the running example, $\mathcal{U}^0_{class}{=}\{a, c, d, e, n\}$. We identify classes $\mathcal{U}^1_{class}{=}\{\{a\}, C1, \{n\}\}$, where $C1{=}\{c, d, e\}$, and $\mathcal{U}^2_{class}{=}\{C2\}$, where $C2{=}\{\{a\}, C1, \{n\}\}$. For each class discovered, we extract the corresponding *instances* from the log. Within a case, multiple instances of the same abstraction class may be identified, e.g., instances 5 and 6 in Table II both represent an instance of $C1$. We define a hierarchy of instances as follows.

Definition 4 (Hierarchy of Instances). *Let $\mathcal{U}^i_{class}$ be the universe of classes at level $i{\in}\mathbb{N}$ and $\mathcal{U}_{ts}$ be the universe of timestamps. An instance at level i describes the presumed (historical) execution of a class at level i and consists of the executions at level $i{-}1$ if $i{>}0$. Let $\mathcal{A}$ be the universe of activity instances. The universe of instances at level i is defined as $\mathcal{A}^i{=}\mathcal{A}$ and $\forall\mathbf{i}{\in}\mathcal{A}^i(\pi_{class}(\mathbf{i}){=}\pi_{act}(\mathbf{i}))$ if $i{=}0$, else $\mathcal{A}^i{=}\mathcal{P}(\mathcal{A}^{i-1})\backslash\{\emptyset\}$ and $\forall\mathbf{i}{\in}\mathcal{A}^i(\pi_{class}(\mathbf{i}){\in}\mathcal{U}^i_{class})$. The corresponding attributes can be derived from the underlying instances at the lower-level such that, given $\mathbf{i}{\in}\mathcal{A}^i$, where $i{\in}\mathbb{N}$,*

- $\pi_{st}\colon \mathcal{A}^i{\to}\mathcal{U}_{ts}$, *where* $\pi_{st}(\mathbf{i})$ *denotes the start time of* $\mathbf{i}$ *s.t.* $\pi_{st}(\mathbf{i}){=}\pi_{st}(\mathbf{i})$ *if* $i{=}0$ *(since* $\mathbf{i}{\in}\mathcal{A}$*), else* $\pi_{st}(\mathbf{i}){=}min(\{\pi_{st}(\mathbf{i}')|\mathbf{i}'{\in}\mathbf{i}\})$,
- $\pi_{ct}\colon \mathcal{A}^i{\to}\mathcal{U}_{ts}$, *where* $\pi_{ct}(\mathbf{i})$ *denotes the completion time of* $\mathbf{i}$ *s.t.* $\pi_{ct}(\mathbf{i}){=}\pi_{ct}(\mathbf{i})$ *if* $i{=}0$, *else* $\pi_{ct}(\mathbf{i}){=}max(\{\pi_{ct}(\mathbf{i}')|\mathbf{i}'{\in}\mathbf{i}\})$.

Note that we do not impose the constraint that all the instances at level i must be in an instance at level $i{+}1$.

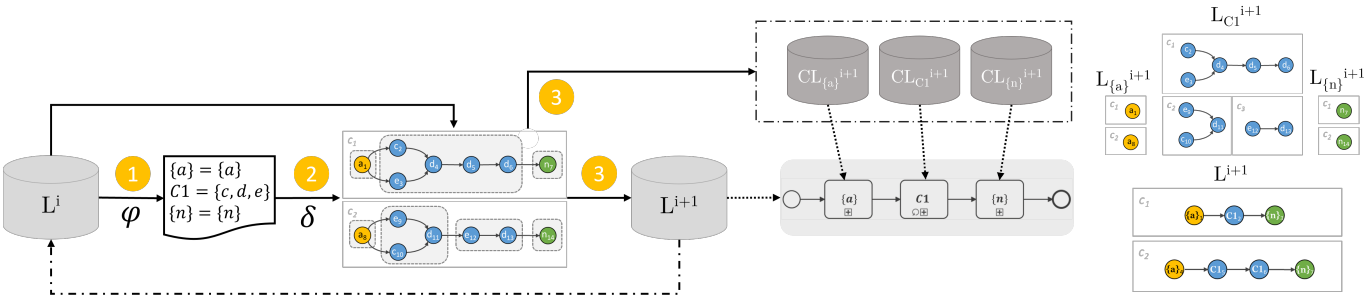


Fig. 3: Schematic Overview of Proposed Approach.

Instances in a case at level $i{\in}\mathbb{N}$ together form an instance at level $i{+}1$. In the running example, we extract four instances $\mathbf{i}_4$, $\mathbf{i}_5$, $\mathbf{i}_6$, and $\mathbf{i}_7$ at level 1 from $c_2{\in}L$ in Table I. Hence, the same case at level 1, written as c_2^1, contains these extracted instances, i.e., $c_2^1{=}\{\mathbf{i}_4, \mathbf{i}_5, \mathbf{i}_6, \mathbf{i}_7\}$. These instances describe the activity instances $\{a_8\}$, $\{e_9, c_{10}, d_{11}\}$, $\{e_{12}, d_{13}\}$, and $\{n_{14}\}$, respectively, of the underlying log L. Similarly, at level 2 (L_2 in Table III), c_2^2 describes one instance of $C2$ and consists of all instances of $c_2^1{\in}L_1$ (Table II).

Definition 5 (Hierarchy of Cases and Logs). *Let $\mathcal{A}^i$ be the universe of instances at level $i{\in}\mathbb{N}$. The universe of cases at level i, i.e., $\mathcal{C}^i$, is recursively defined as $\mathcal{C}^i{=}\mathcal{C}$ if $i{=}0$ else $\mathcal{C}^i{\subseteq}\mathcal{P}(\mathcal{P}(\mathcal{A}^{i-1}))$ s.t. $\forall c^i{\in}\mathcal{C}^i$, c^i is a derivative case of $c^{i-1}{\in}\mathcal{C}^{i-1}$ where $\forall \mathbf{i}{\in}c^i(\mathbf{i}{\subseteq}c^{i-1})$ and $\forall \mathbf{i}, \mathbf{i}'{\in}c^i(\mathbf{i}{\cap}\mathbf{i}'{=}\emptyset)$. The universe of logs at level $i{\in}\mathbb{N}$ is defined as $\mathcal{L}^i{=}\mathcal{L}$ if $i{=}0$ else $\mathcal{L}^i{\subseteq}\mathcal{P}(\mathcal{P}(\mathcal{C}^{i-1}))$ s.t. $\forall L{\in}\mathcal{L}^i$, L is the collection of all the derivative cases of cases in L^{i-1}.*

Note that an *activity*, an *activity instance*, a *case*, and an *activity instance log* can be seen as the building blocks of a hierarchy of classes, instances, cases, and logs, respectively. To extract the classes and instances at different level of the hierarchy, we define two key oracle functions, class abstraction function φ and instance extraction function δ as shown in Figure 3. First, the class abstraction function φ groups the classes at the lower-level and extracts the classes at the next level as follows.

Definition 6 (Class Abstraction Oracle). *Let $\mathcal{L}^i$ be the universe of logs at level $i{\in}\mathbb{N}$. An i^{th}-order class abstraction oracle φ^i detects the possible applicable classes at level $i{+}1$, i.e., $\varphi^i{:}\,\mathcal{L}^i{\to}\mathcal{P}(\mathcal{U}_{class}^{i+1})$, where $\mathcal{U}_{class}^{i+1}$ is the universe of classes at level $i{+}1$, such that, given $L{\in}\mathcal{L}^i$, $\forall C^i{\in}\alpha(L)(\exists C^{i+1}{\in}\varphi(L)(C^i{\in}C^{i+1}))$.*

In the running example L, we identify three classes at level 1 with $\varphi^0(L){=}\{\{a\}, C1, \{n\}\}$, where $C1{=}\{c, d, e\}$. Given the log L^1 extracted from L, we identify the classes at level 2 as $\varphi^1(L^1){=}\{C2\}$, where $C2{=}\{\{a\}, C1, \{n\}\}$. Intuitively, an instance of class C^i, where $i{\in}\mathbb{N}^+$, contains the instances of the classes at the lower-level in C^i. For example, the instances in c_1 in Table I with activities c, d, and e are collected as one instance $\mathbf{i}_2$ in Table II since $C1{=}\{c, d, e\}$. The extraction of instances is defined as follows.

Definition 7 (Instance Extraction Oracle). *Given $i{\in}\mathbb{N}$, let $\mathcal{C}^i$ be the universe of cases at level i and $\mathcal{U}_{class}^i$ and $\mathcal{A}^i$ be the universe of classes and instances at level i, respectively. An instance extraction oracle at level i δ^i is a function $\delta^i{:}\,\mathcal{C}^i{\times}\mathcal{P}(\mathcal{U}_{class}^{i+1}){\to}\mathcal{P}(\mathcal{A}^{i+1})$, s.t., given case $c{\in}\mathcal{C}^i$ and a valid class definition at level $i{+}1$ $\mathbf{C}^{i+1}{\in}\mathcal{U}_{class}^{i+1}$, an instance at level i can only be assigned to at most one instance at level $i{+}1$, i.e., $\forall \mathbf{i}, \mathbf{i}'{\in}\delta^i(c, \mathbf{C}^{i+1})(\mathbf{i}{\cap}\mathbf{i}'{=}\emptyset)$, and $\forall \mathbf{i}{\in}\delta^i(c, \mathbf{C}^{i+1})\forall \mathbf{i}''{\in}\mathbf{i}(\pi_{class}(\mathbf{i}''){\in}\mathbf{C}^{i+1})$. We overload $\delta^i(c, \mathbf{C}^{i+1})$ and write $\delta^i(L^i, \mathbf{C}^{i+1}){=}\{\delta^i(c, \mathbf{C}^{i+1})|c{\in}L^i\}$.*

With the definition, the instances of $C1$, $C2$, and $C3$ from $\varphi^0(L^0)$ can be extracted as $\delta^0(L^0, \varphi^0(L^0)){=}\{\mathbf{i}_1, \mathbf{i}_2, \mathbf{i}_3, \mathbf{i}_4, \mathbf{i}_5, \mathbf{i}_6, \mathbf{i}_7\}$, where each row in Table II represents an instance and the corresponding attributes. For example, $\mathbf{i}_5$ is an instance of $C2$ ($\pi_{class}(\mathbf{i}_5){=}C2$) extracted from e_9, c_{10}, and d_{11} ($\mathbf{i}_5{=}\{e_9, c_{10}, d_{11}\}$) in c_2^0 such that $\pi_{st}(\mathbf{i}_5){=}$2021.01.06 08:50 and $\pi_{ct}(\mathbf{i}_5){=}$2021.01.14 10:06. In Figure 3, we color the activity instances of the same class at level 1 with the same color and group the activity instances if they belong to the same instance at level 1 with a dashed line.

Finally, for every class identified, we create a *class log* containing the instances of the class. In the running example, given $\delta^0(L, \varphi^0(L))$, we extract class logs $L_{\{a\}}{=}\{\mathbf{i}_1, \mathbf{i}_4\}$, $L_{C1}{=}\{\mathbf{i}_2, \mathbf{i}_5, \mathbf{i}_6\}$, and $L_{\{n\}}{=}\{\mathbf{i}_3, \mathbf{i}_7\}$ in Table II. Given a class, the corresponding class log is defined as follows.

Definition 8 (Class Log). *Let L^i be a log at level $i{\in}\mathbb{N}$ and $\mathbf{C}^{i+1}{\subseteq}\mathcal{U}_{class}^{i+1}$ be the applicable classes at level $i{+}1$, i.e., $\mathbf{C}^{i+1}{=}\varphi(L^i)$. A class log of $C{\in}\mathbf{C}^{i+1}$, written as L_C, is the collection of the instances of the class C where $L_C{=}\{\mathbf{i}|\mathbf{i}{\in}\delta^i(L^i, \mathbf{C}^{i+1})(\pi_{class}(\mathbf{i}){=}C)\}$.*

Given the definitions, the extraction of a hierarchy of logs and class logs is trivial provided the instances. To summarize, the framework takes a log at level $i{\in}\mathbb{N}$ and discovers the applicable classes at level $i{+}1$ with abstraction oracle φ. The instances of each class identified are extracted using δ. For every class, we collect the instances of each class and generate a log, which contains the data of a subprocess represented with the class. The final output is a log at level $i{+}1$ computed from the instances extracted. The process repeats until the level that a user specified and creates a hierarchy of abstractions.

V. An Instantiation

In this section, we introduce an instantiation of the two core oracle functions in the proposed framework: (1) the class abstraction φ and (2) the instance extraction oracle δ for every abstraction level.

A. Abstracting Classes

Given a process, it is intuitive to group the classes that have *similar surrounding behavior* as a class at the higher level. We define classes to be similar if they having similar preceding and following classes.

First, we extract the classes preceding (*preceding patterns*) and following (*following patterns*) every class within a certain *window size* (w). To increase the probability that two classes group together, we let $\hat{pred}_c^w(\mathbf{i})$ and $\hat{succ}_c^w(\mathbf{i})$ be a set of w elements randomly selected from $pred_c^w(\mathbf{i})$ and $succ_c^w(\mathbf{i})$, respectively. The patterns are extracted as below.

Definition 9 (Preceding and Following Patterns). *Given a log L, let $\mathbf{C}$ be all the classes of instances in L. For every $C \in \mathbf{C}$, the preceding and following patterns of C are*

$$\mathbf{p}_C^{pred} = \bigcup_{c \in L} \{\{\pi_{class}(\mathbf{i}') | \mathbf{i}' \in \hat{pred}_c^w(\mathbf{i})\} | \mathbf{i} \in c(\pi_{class}(\mathbf{i}) = C)\},$$

$$\mathbf{p}_C^{succ} = \bigcup_{c \in L} \{\{\pi_{class}(\mathbf{i}') | \mathbf{i}' \in \hat{succ}_c^w(\mathbf{i})\} | \mathbf{i} \in c(\pi_{class}(\mathbf{i}) = C)\}.$$

We denote all the preceding and following patterns in L as $\mathbf{P}^{pred} = \bigcup_{C \in \mathbf{C}} \mathbf{p}_C^{pred}$ and $\mathbf{P}^{succ} = \bigcup_{C \in \mathbf{C}} \mathbf{p}_C^{succ}$.

The green (top) and blue (bottom) rows in Figure 4a present the preceding and following patterns of L with window size 1. The arrows in Figure 4a indicate if a pattern precedes or follows an activity in L. For example, a is followed by $\{c\}$ and $\{e\}$ and preceded with an empty set (since a is the first activity in all the cases in L). Based on the preceding and following patterns, we quantify the difference between any two classes using *earth mover's distance* (*EMD*) [4].

Definition 10 (Preceding and Following Similarity). *Let $\mathbf{P}^{pred}$ and $\mathbf{P}^{succ}$ be the preceding and following patterns of the classes $\mathbf{C}$ in a log L. The frequency that a pattern $\mathbf{p}$ precedes or follows a class $C \in \mathbf{C}$ are*

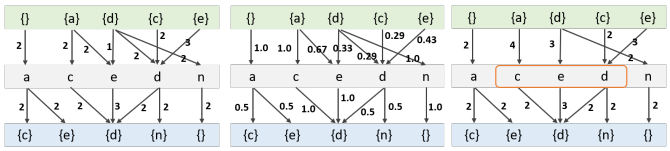
$$f^{pred}(C, \mathbf{p}) = |\bigcup_{c \in L} \{\hat{pred}_c^w(\mathbf{i}) | \mathbf{i} \in c(\pi_{class}(\mathbf{i}) = C)\}|,$$

$$f^{succ}(C, \mathbf{p}) = |\bigcup_{c \in L} \{\hat{succ}_c^w(\mathbf{i}) | \mathbf{i} \in c(\pi_{class}(\mathbf{i}) = C)\}|.$$

The probability that a pattern $\mathbf{p}$ precedes or follows a class C is $p^{pred}(C, \mathbf{p}) = f^{pred}(C, \mathbf{p}) / \Sigma_{\mathbf{p}' \in \mathbf{P}^{pred}} f^{pred}(C, \mathbf{p}')$ and $p^{succ}(C, \mathbf{p}) = f^{succ}(C, \mathbf{p}) / \Sigma_{\mathbf{p}' \in \mathbf{P}^{succ}} f^{succ}(C, \mathbf{p}')$, respectively. Let the distance between any different patterns be 0 and 1 otherwise. According to [5], the EMD *between any $C, C' \in \mathbf{C}$ of the preceding and following patterns are as follows.*

$$EMD^{pred}(C, C') = 1 - \Sigma_{\mathbf{p} \in \mathbf{P}^{pred}} (p^{pred}(C, \mathbf{p}) - p^{pred}(C', \mathbf{p}), 0),$$

$$EMD^{succ}(C, C') = 1 - \Sigma_{\mathbf{p} \in \mathbf{P}^{succ}} (p^{succ}(C, \mathbf{p}) - p^{succ}(C', \mathbf{p}), 0).$$



(a) Frequency. (b) Probability. (c) Updated.

Fig. 4: Pattern graphs for the running example L.

The labels of the arrows in Figure 4a and 4b show the frequency and probability of a pattern precedes and follows a class. For instance, in Figure 4b, there is a 50% probability that d goes to $\{d\}$ and 50% that d goes to $\{n\}$. We quantify the similarity of classes using the probability. If the average similarity of the classes, defined as $1-EMD$ for preceding and following patterns, is greater than a threshold provided by the user, we consider them *similar enough* and group them together. The corresponding arrows and frequency labels are updated as shown in Figure 4c, where the threshold is set to 0.25. Then, we compute the probability of the edges and repeat the process until no classes can be grouped. The final output, i.e., sets of classes grouped, are the classes at the next level. Note that, for the singletons classes that are not grouped, we define them as a class at the next level.

B. Extracting Instances

We propose two solutions for extracting the instances. The first one is straightforward. For each case c, we define an instance $\mathbf{i}$ of a class C at level $i \in \mathbb{N}^+$ as the collection of all the instances in c where their classes are in C, i.e., $\mathbf{i} = \{\mathbf{i}' | \mathbf{i}' \in c \wedge \pi_{class}(\mathbf{i}') \in C\}$.

The second solution proposed "cuts" the instances in a case according to the start and completion classes identified. A (sub)process often starts and completes with specific activities, e.g., receive an application, forward a case to another department, reject a purchase request, ..., etc. An instance of a class at the higher level can be extracted by "cutting" a case using the potential start and completion classes at the lower level, i.e., endpoint classes. There are two steps in the extraction. First, we identify the endpoint classes at the lower level for a class at higher level. For instance, given the running example L, suppose $C1 = \{c, d, e\}$. We identify d as the completion class of $C1$ ($\mathcal{C}_{C1}^{complete}$) and $\{c, e\}$ as the start classes ($\mathcal{C}_{C1}^{start}$).

Given the endpoint classes identified, we compute the "cutting points" of a case, which fall between the instances with the start and completion classes. For example, in Figure 2, given $\mathcal{C}_{C1}^{start} = \{e\}$ and $\mathcal{C}_{C1}^{complete} = \{d\}$, we extract three instances of $C1$, $\mathbf{i}_2 = \{c_2, e_3, d_4, d_5, d_6\}$, $\mathbf{i}_5 = \{e_9, c_{10}, d_{11}\}$, and $\mathbf{i}_6 = \{e_{12}, d_{13}\}$. In case 2, we "cut" the case between d_{11} and e_{12} based on their classes which indicates the initiation of a new instance at the higher level. We identify the cutting points based on the start and completion classes as below.

Definition 11 (Identifying Cutting Points). *Given a log L^i, where $i \in \mathbb{N}$, and the applicable classes $\mathbf{C}^{i+1}$. Let $\mathcal{C}_C^{start}$ and*

TABLE IV: Parameter setting of abstraction level (*L*), window size (*W*), similarity threshold (*S*), dependency threshold of HM, and noise threshold of IMflc (*IMflc*) and IMf.

	L	*W*	*S*	*IMflc*	*HM-Dependency Threshold*	*IMf-Noise Threshold*
Min	0	1	0	0	0	0
Max	3	10	1.0	1.0	1.0	1.0
Step	1	1	0.1	0.1	0.1	0.1

$\mathcal{C}_C^{complete}$ *be the start and completion classes of* $C \in \mathbf{C}^{i+1}$. *For every* $c \in L^i$, *we perform the following steps.*

1) *Select the instances with the start and completion classes* $\mathbf{I}=\{\mathbf{i}|\mathbf{i}\in c(\pi_{class}(\mathbf{i})\in\mathcal{C}_C^{start}\cup\mathcal{C}_C^{complete})\}$.
2) *Extract a set of timestamps that represent the cutting points* $\mathcal{T}=\{\pi_{st}(\mathbf{i})|\mathbf{i}\in c(\pi_{class}(\mathbf{i})\in\mathcal{C}_C^{start} \wedge \{\pi_{class}(\mathbf{i}')|\mathbf{i}'\in s\hat{u}cc_{\mathbf{I}}^{1}(\mathbf{i},\prec)\}\subseteq\mathcal{C}_C^{complete})\}$.
3) *Sort the timestamps extracted into a list* $\hat{\mathcal{T}}=\langle 1,2,...,|\mathcal{T}|\rangle$, *where* $\forall 1\leq i<j\leq|\mathcal{T}|(\hat{\mathcal{T}}(i)\leq\hat{\mathcal{T}}(j))$.
4) *Collect the instances s.t. every instance at level* $i{+}1$ *is a set of instances at level* i *in between the timestamps extracted* $\bigcup_{1\leq k<|\hat{\mathcal{T}}|}\{\mathbf{i}|\mathbf{i}\in c(\pi_{class}(\mathbf{i})\in C)\wedge\pi_{st}(\mathbf{i})\geq\hat{\mathcal{T}}(k)\wedge(k<|\hat{\mathcal{T}}|\rightarrow\pi_{ct}(\mathbf{i})<\hat{\mathcal{T}}(k{+}1)\wedge k=|\hat{\mathcal{T}}|\rightarrow\pi_{ct}(\mathbf{i})>max(\hat{\mathcal{T}}))\}$.

We apply the same process to all the cases in L^i *and extract all the instances of* C *at level* $i{+}1$.

Finally, with the instances extracted, the abstract log at the next level and the class logs are computed and applied as an input for the next iteration until the level that a user specified.

VI. Evaluation

In this section, we evaluate the proposed framework with the instantiation. First, we present a quantitative evaluation by comparing the quality metrics of discovered process models with and without applying abstraction. Then, we present the models discovered to showcase the applicability of the framework.

A. Quantitative Evaluation

In this section, we evaluate the quality of process models with and without abstraction by applying existing discovery algorithms on the abstract logs. To fairly compare the results, we flatten the process models discovered using the logs at different levels, i.e., an *abstract model*, by replacing each transition in the model with a model discovered using the class log of the corresponding transition (*class model*). We repeat the replacement process until all the transitions are labeled with classes at level 0, i.e., the classes, or activities, in the processes without applying abstraction. Meanwhile, we adopt the first solution proposed to extract the instances to compare with results without abstraction.

We evaluate the process models using two datasets [6], [7]. We apply Inductive Miner - infrequent & life cycle (IMflc) [8] for discovering abstract models (as it is the only discovery algorithm known that can handle life cycle transitions reasonably). The class models are discovered using different versions of the Inductive Miner, i.e., (IM) [9], (IMf) [10] and (IMd) [11], and, the Heuristic Miner (HM) [12].

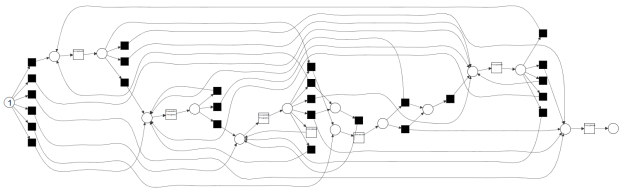


Fig. 5: The process model without abstraction, i.e., level 0, discovered using HM with dependency threshold 0.2 on [7].[4]

Using the aforementioned logs and algorithms, we conducted various experiments with different parameter settings of our framework. An overview of the total parameter setup is presented in Table IV. In Table V we present the average results obtained in terms of fitness [14], precision [15], and F1-score of the flattened models using token-based replay technique.[4]. The simplicity is evaluated for both flattened and abstract models, which are intended for human analysts [16].

We observe that the simplicity of the obtained models increases for higher levels of abstraction. For the abstract models this is to be expected, i.e., these models contain less elements. However, we also observe slight increase for the flattened models, which is more surprising. For [6], we observe an increase of F1-score for the flattened models when we increase the abstraction levels. This is mainly caused by a significant increase in precision for the models based on higher levels of abstraction. However, generally, the fitness levels of the corresponding models is lower. For [7], similar results are obtained, however, the drop in fitness is too large compared to the gain in precision, i.e., F1-scores for higher levels of abstraction are worse compared to the raw event data. Nonetheless, these result confirms the ability of abstraction techniques to counter the typical underfitting behavior of process discovery algorithms on raw event data. For the impact of the parameters, we observe that the similarity threshold and the discovery algorithm used (along with the corresponding parameters) have stronger impact to the quality of the process models comparing to other parameters.

B. Qualitative Evaluation

The complexity and density of a process model are known to impact the understandability for a human analyst [2], [17]. To demonstrate the applicability of the framework, we compare the process models with and without abstraction. Figure 5 presents an impression of a process model without abstraction. The abstract models at levels 1 to 3 are shown in Figure 6, with the class models discovered using the same parameter setting as the one without abstraction. Note that though the labels are unclear, yet the figures sketch the overall structure of the process models.

[3]See [9] for the process tree representation and [13] for Petri nets.

[4]See https://owncloud.fraunhofer.de/index.php/s/zs2RFvji47g1Ck3 for all results. Due to space limitations we only show aggregate results.

TABLE V: Average fitness, precision, and F1-Score of flattened process models per level, where process models at level 0 refer to the models discovered without applying abstraction.

		Fitness	*Precision*	*F1-Score*	*Simplicity-Flatten*	*Simplicity-Abstract*
bpi 13 [7]	Level 0	0.9697	0.7956	0.8728	0.5829	-
	Level 1	0.6858	0.9817	0.7857	0.6385	0.9382
	Level 2	0.7172	0.9802	0.8116	0.6402	0.9615
	Level 3	0.7245	0.9802	0.8177	0.6401	0.9661
Sepsis [6]	Level 0	0.8043	0.8627	0.6979	0.5437	-
	Level 1	0.7430	0.9584	0.8264	0.5857	0.9698
	Level 2	0.7441	0.9595	0.8280	0.5860	0.9753
	Level 3	0.7443	0.9598	0.8283	0.5861	0.9756

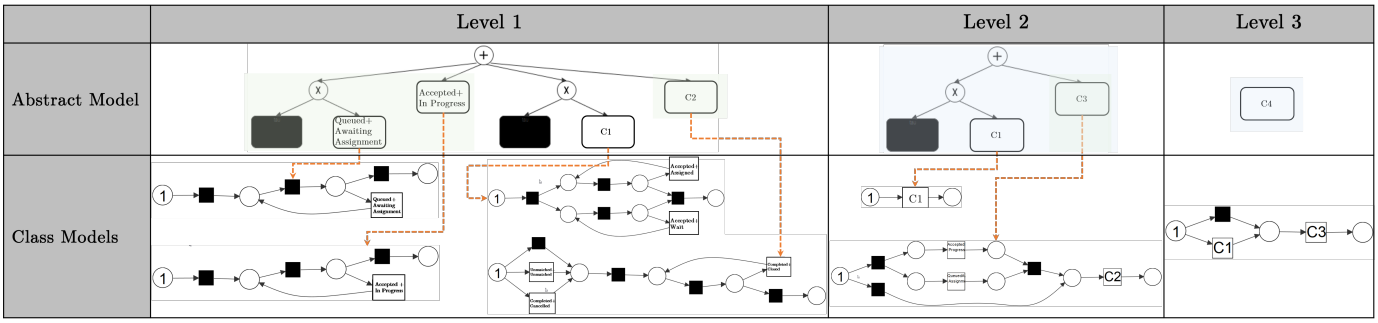


Fig. 6: Abstract and class models of [7] at hierarchy levels 1 to 3 with setting of window size 4, similarity threshold 0.9, and IMflc noise threshold 0.2. The fitness/precision (F1-Score) of the flattened models are 0.3898/1.0(0.5609), 0.7118/1.0(0.8317), and 0.7987/1.0(0.8881) at level 1 to 3, respectively.[3]

By using the divide-and-conquer strategy, compared to Figure 5, a human analyst only needs to interpret a limited number of behaviors at a time. Additionally, we observe a change of the behavior in the abstraction. In the abstract model of level 1, the class $C2$ is in parallel with two singleton classes. They are grouped as $C3$ in level 2 and the two singleton classes are before $C2$ at level 2. The quality of the flattened models, i.e., fitness, precision, and F1-score, increases with the change.

To summarize, we evaluated our framework both quantitatively and qualitatively. The quantitative evaluation shows that the precision is higher with abstraction and the quality of the process models depends mainly on the similarity threshold and the discovery algorithm applied. In qualitative evaluation, we demonstrate the applicability of the framework with the visualization of process models. Hence, we conclude that the framework enhances the overall interpretability of discovered process models without compromising the results.

VII. Related Work

In this section, we discuss the work that is most relevant to the proposed framework. For a complete overview of the abstraction techniques for process mining, we refer to [3].

To reasonably group the concepts or instances (*events* in most work) together into high-level concepts or instances, some techniques require various levels of domain knowledge provided by the user. For example, [18] assumes that the information of different levels of abstraction is provided as an attribute in the data to discover hierarchical models. The same assumption is applied in [19], which aims to predict the higher-level concepts of events using log with labeled events and does not allow for automatically discovering hierarchical models. The authors in [20] propose to compute an *activity tree*, which contains the hierarchical information. Nevertheless, among three alternatives, i.e., domain knowledge provided, randomly assigned, and flat activity trees, the best results still rely on the domain knowledge provided.

Other approaches do not require users to explicitly provide the abstraction information. De Leoni and Dündar apply clustering algorithms with the feature extracted using frequency or duration encoding within a fixed time frame determined by the user [21]. Nguyen et al. imitate how a human analyst applies a divide-and-conquer strategy to decompose a process model by searching for the "cutting points" based on graph modularity [22]. The above techniques require a certain level of parameter tuning; however, less knowledge is required compared to the techniques using explicitly abstraction information [18]–[20]. In both [23] and [24], a process model for each higher-level transition are used for abstraction. While the latter one does not require users to provide such models, both techniques utilize alignment to extract an abstracted log, which is computationally expensive.

Similar to [18] and [20], the proposed framework supports hierarchical event abstraction. However, instead of relying on domain knowledge, we propose an instantiation to extract the abstraction information. Moreover, most, or, all, techniques apply on *events*. In real-life event logs, there are different status of an activity, e.g., start, complete, suspend. To the best of our knowledge, the framework proposed in this paper is

the first to be applied on *activity instances* instead of events, which is generally more applicable to different event logs.

VIII. Discussion and Limitations

This section discusses the framework and its limitations. There are two key functions in the proposed framework, the identification of classes at different levels, i.e., φ, and the extraction of the corresponding instances, i.e., δ. The instantiation assumes that each class at level $i{\in}\mathbb{N}$ exists in only one class at level $i{+}1$. However, in real life, a class may exist in different "parts" (higher-level classes) of a process. Therefore, it would be interesting to apply an instantiation that allows for duplicated labels and explore its impact.

The instantiation of δ which splits a case based on start and completion classes at the lower level highly depends on the selection of such classes. Consider a log containing incomplete cases, for example, $L{=}\{\{a_1, b_2, d_3, a_4, c_5\}, \{a_6, b_7, d_8, c_9\}, \{a_{10}, b_{11}\}\}$ for a process $\rightarrow(\wedge(\circlearrowleft(a), \rightarrow(b, d)), c)$. We select $\{a\}$ as the start classes and $\{b, c\}$ as the completion classes of a class $C{=}\{a, b, c, d\}$. For the first case, we extract two instances $\{a_1, b_2\}$ and $\{a_4, c_5\}$, resulting from the selection of b in completion classes. As shown with the example, the selection of classes that does not represent the termination of a class at the higher level can lead to inaccurate results. In addition, we do not guarantee that all the instances at level $i{\in}\mathbb{N}$ are assigned to instances at level $i{+}1$. In the example, instance d_3 is excluded in the instances at level $i{+}1$.

In the instantiation, three parameters, i.e., *window size*, *similarity threshold*, and *the final level of abstraction*, need to be configured by the user. However, as shown in the experiments, the quality of the flattened models depends more on the similarity threshold and the discovery algorithms applied. Therefore, we may further automate the framework by removing, i.e., setting the parameters to constants, other parameters while guaranteeing not affecting the results.

IX. Conclusion

We proposed a framework for event abstraction based on *activity instances*. The framework first identifies classes and extracts instances for each abstracted class at every hierarchical abstraction level. The output, a log based on extracted classes, is then applied as input for the next iteration. We apply the earth mover's distance for comparing the similarity of classes and group the classes that are similar enough in the instantiation. We propose two solutions for the extraction of instances. First, an instance at the higher-level is the collection of all the relevant instances in a case. The second solution cuts a case based on the intuition that a (sub)process often starts and completes with certain activities. The evaluation shows the applicability of the framework without compromising the quality of the process models compared to the models discovered without applying abstraction. In certain settings, the results with abstraction outperform the ones without applying abstraction. Meanwhile, we demonstrate that the framework allows for applying different process mining techniques to provide better results. As a next step, we aim to remove the requirement that a class can only be assigned to one class at the higher-level and allow for multiple higher-level class containing the same class at the lower-level. Also, we aim to develop different strategies for the selection of start and completion classes and evaluate the corresponding impact on discovered process models.

References

[1] W. van der Aalst, "Data science in action," in *Process mining*, 2016.
[2] J. Mendling, H. A. Reijers, and J. Cardoso, "What makes process models understandable?" in *BPM*, 2007.
[3] S. J. van Zelst, F. Mannhardt, M. de Leoni, and A. Koschmider, "Event abstraction in process mining: literature review and taxonomy," *Granular Computing*, vol. 6, no. 3, 2021.
[4] Y. Rubner, C. Tomasi, and L. J. Guibas, "A metric for distributions with applications to image databases," in *Sixth International Conference on Computer Vision*, 1998.
[5] S. Leemans, A. F. Syring, and W. van der Aalst, "Earth movers' stochastic conformance checking," in *BPM*, 2019.
[6] F. Mannhardt, "Sepsis cases - event log."
[7] W. Steeman. (2013) BPI challenge 2013, closed problems.
[8] S. Leemans, D. Fahland, and W. van der Aalst, "Using life cycle information in process discovery," in *BPM*, 2016.
[9] S. Leemans, D. Fahland, and W. van der Aalst, "Discovering block-structured process models from event logs-a constructive approach," in *International conference on applications and theory of Petri nets and concurrency*, 2013.
[10] S. Leemans, D. Fahland, and W. van der Aalst, "Discovering block-structured process models from event logs containing infrequent behaviour," in *BPM*, 2013.
[11] S. Leemans, D. Fahland, and W. van der Aalst, "Scalable process discovery and conformance checking," *Software & Systems Modeling*, vol. 17, no. 2, 2018.
[12] A. Weijters, W. van der Aalst, and A. A. De Medeiros, "Process mining with the heuristics miner-algorithm," *Technische Universiteit Eindhoven, Tech. Rep. WP*, vol. 166, 2006.
[13] J. Peterson, "Petri nets," *ACM Computing Surveys*, vol. 9, no. 3, 1977.
[14] A. Berti and W. van der Aalst, "Reviving token-based replay: Increasing speed while improving diagnostics." in *ATAED@ Petri Nets/ACSD*, 2019.
[15] J. Munoz-Gama and J. Carmona, "A fresh look at precision in process conformance," in *BPM*, 2010.
[16] F. R. Blum, "Metrics in process discovery," Technical report, TR/DCC. 1–21, Tech. Rep., 2015.
[17] H. A. Reijers and J. Mendling, "A study into the factors that influence the understandability of business process models," *IEEE Transactions on Systems, Man, and Cybernetics-Part A: Systems and Humans*, vol. 41, no. 3, 2010.
[18] S. Leemans, K. Goel, and S. J. van Zelst, "Using multi-level information in hierarchical process mining: Balancing behavioural quality and model complexity," in *2nd ICPM*, 2020.
[19] N. Tax, N. Sidorova, R. Haakma, and W. van der Aalst, "Event abstraction for process mining using supervised learning techniques," in *Proceedings of SAI Intelligent Systems Conference*, 2016.
[20] X. Lu, A. Gal, and H. A. Reijers, "Discovering hierarchical processes using flexible activity trees for event abstraction," in *2nd ICPM*, 2020.
[21] M. de Leoni and S. Dündar, "Event-log abstraction using batch session identification and clustering," in *Proceedings of the 35th Annual ACM Symposium on Applied Computing*, 2020.
[22] H. Nguyen, M. Dumas, A. H. ter Hofstede, M. La Rosa, and F. M. Maggi, "Stage-based discovery of business process models from event logs," *Information Systems*, vol. 84, 2019.
[23] F. Mannhardt, M. De Leoni, H. A. Reijers, W. van der Aalst, and P. J. Toussaint, "From low-level events to activities-a pattern-based approach," in *BPM*, 2016.
[24] F. Mannhardt and N. Tax, "Unsupervised event abstraction using pattern abstraction and local process models," *arXiv preprint arXiv:1704.03520*, 2017.