Proceedings of the 20th Conference on Computer Science and Intelligence Systems (FedCSIS) pp. 231–241
DOI: 10.15439/2025F4755
ISSN 2300-5963 ACSIS, Vol. 43

Segmentation and Process Assignment of Semi-Structured Event Logs

Piotr Przymus, Krzysztof Rykaczewski, Janusz Zieliński, Łukasz Mikulski
0000-0001-9548-2388, 0000-0002-4772-6330, 0000-0002-1127-9874, 0000-0002-6711-557X
Nicolaus Copernicus University in Toruń
ul. Gagarina 11, 87-100 Toruń, Poland
Email: {piotr.przymus, krzysztof.rykaczewski, janusz.zielinski, lukasz.mikulski}@mat.umk.pl

***Abstract*—Process mining provides valuable insights by discovering process models from execution logs. However, its effectiveness depends heavily on high-quality, well-structured logs. Many real-world systems produce low-level, semi-structured logs lacking clear process identifiers, causing misalignment with their intended process models.**

This paper introduces a method for structuring raw event logs by segmenting event streams and mapping them to known processes. Using process traces from experienced users, we develop a model that infers process assignments in unstructured logs. Our approach is motivated by a modular enterprise system without predefined workflows, where dynamic processes generate low-level logs requiring interpretation.

We validate our method on a semi-synthetic business dataset and a fully synthetic dataset from PLG2. Our results demonstrate that trace segmentation improves process discovery, aligns logs with meaningful structures, and significantly enhances process mining in unstructured environments.

I. Introduction

PROCESS mining is a powerful methodology that uses event logs to gain insights into business processes, enabling organizations to analyze, monitor, and optimize their workflows. Traditionally, process mining relies on structured event logs, where each event is explicitly linked to a process instance identifier. However, in many real-world contexts, event logs are only semi-structured, lacking process identifiers yet containing timestamps, event types, and user identifiers. This limitation complicates the reconstruction of process execution and hinders effective process analysis.

This research addresses that challenge by presenting an approach to structure semi-structured event logs. Specifically, we assign process identifiers to raw event data by leveraging a set of ideal process traces performed by experienced users. These labeled traces, which include known process identifiers and types, serve as training data for a machine learning-based system. The trained model is then applied to raw event logs to infer process assignments and identify user activities linked to specific processes.

We explored a practical application of this approach in a modular enterprise management system provided to external entities. These entities use the system to execute diverse business processes spanning multiple modules. Because of the variability of clients, the heterogeneity of business processes, and the system's flexibility, incorporating process identifiers into the logs is not feasible from a business perspective. Therefore, our methodology relies exclusively on semi-structured data.

By implementing this approach, we provide a solution that enhances process mining capabilities in environments where structured event logs are unavailable. This research contributes to process mining by introducing a method for structuring semi-structured event logs, enabling more effective business process analysis, anomaly detection, and performance monitoring.

a) Replication Package: To facilitate reproducibility and further research, we provide a complete replication package containing code, data, and experimental scripts. It is publicly available at:

`https://github.com/ncusi/Segmentation_and_Process_Assignment_of_Semi-StructuredEvent_Logs`

The remainder of this paper is organized as follows. Section II discusses related work. Sections III and IV reviews necessary preliminaries on event logs, process mining, and similarity measures. Section V states the problem, and Section VI describes our methodology for structuring semi-structured event logs. In Section VII, we discuss our experimental design, and Section VIII presents the results. In Section IX, we assess threats to validity. Finally, we conclude and propose future directions in Section X.

A. Running Example (Motivation)

Consider a customer-support system where each event is logged as `[time, user, activity, ...]`. A typical log snippet might look like:

```
2025-05-10T09:12:03Z, alice, OpenComplaint, details
2025-05-10T09:14:21Z, alice, AssignAgent, agentId=42
2025-05-10T09:15:07Z, alice, ResolveComplaint, status=closed
2025-05-10T09:20:00Z, alice, OpenComplaint, details
...
```

Without a case identifier, it is unclear which `OpenComplaint` corresponds to which conversation. This can cause supervisors to miss abandoned requests or misattribute resolution times. To address this, we propose to reconstruct each case by matching segments of the event stream to known process templates.

This work was supported by the Regional Operational Programme of the Kuyavian-Pomeranian Voivodeship for 2014–2020 under the grant titled "Budowa zaplecza badawczo-rozwojowego w MGA Sp. z o.o."

II. Related Work

Workflow and process mining have been extensively studied. Classical surveys such as [1], [2], [3] provide a comprehensive overview. A critical challenge in this field involves deriving high-quality, structured event logs from raw, often semi-structured data sources. Our work sits at the intersection of several active research areas, including event-case correlation, trace segmentation, similarity-based activity matching, and process discovery under uncertainty.

A fundamental problem in process mining is event-case correlation inferring case identifiers when they are absent or ambiguous. This challenge has been addressed through various heuristics and model-aware approaches. For example, Helal and Awad [4] propose a runtime event correlation method that uses process models and task durations to group events into likely cases. Brzychczy et al. [5] introduce a rule-based approach for detecting case boundaries in time series data, leveraging domain-specific structural regularities.

The extraction of event logs from relational or unstructured sources has also been widely explored. Andrews et al. [6] present RDB2Log, a quality-aware, semi-automated log extraction framework. Hernandez-Resendiz et al. [7] extend this work by introducing techniques to convert relational data into XES-compliant event logs, facilitating downstream analysis.

The problem of reconstructing traces from flat event streams or system logs has been tackled through temporal and contextual grouping methods. In the domain of unstructured customer service data, Kecht et al. [8] apply natural language inference to extract structured traces from customer service conversations. Korzeniowski and Goczyca [9] propose an automated log-template generation method (SLT) and an interaction-extraction pipeline that discovers application-to-application dependencies in enterprise logs, demonstrating its utility on a large banking system. One of the earliest treatments of the segmentation problem appears in [10], where the authors propose an event correlation function to partition unstructured logs into coherent subsequences.

One of the earliest treatments of the segmentation problem appears in [10], where the authors propose an event correlation function to partition unstructured logs into coherent subsequences. Our approach similarly focuses on identifying process instances from flat user streams, using data-driven similarity and conformance models.

Sequence similarity measures have been applied in activity and behavior mining. Levenshtein similarity, in particular, has proved effective by Tax et al. in [11] for clustering and aligning activity sequences. To the best of our knowledge, although the Tversky similarity measure is widely used in set and pattern matching, it has not previously been applied to trace similarity in process mining. Our work leverages both Levenshtein and Tversky measures to enable efficient and flexible segmentation.

The challenge of multi-case event streams where multiple processes or objects interact has been identified by Martin et al. in [12] as a key limitation in traditional process mining. Recent methods under the umbrella of object-centric process mining [13], [14], [15] provide more expressive models, but they require richer data and frequently involve more complex analysis pipelines. Our work addresses a simpler setting where each user executes one process at a time, but case boundaries remain implicit and must be inferred.

Event log imperfections such as missing case identifiers, incomplete traces, or overlapping activities are well-documented threats to process analysis. Jans et al. [16] discuss the implications of log quality in auditing scenarios. Our method specifically targets one imperfection: the lack of case identifiers in user-specific event streams.

Streaming scenarios add complexity, requiring online segmentation and model updating. Burattin et al. [17] propose methods for control-flow discovery from live event streams. While their focus is on evolving process models, their incremental techniques are relevant to our goal of continuously analyzing user activity streams.

In contrast to prior work, we assume access to a small set of example traces per process and center our efforts on segmenting flat, user-specific event streams with no case identifiers. We combine prefix-based segmentation with syntactic and conformance-based similarity scoring, which allows us to reconstruct approximate process instances even under noisy or truncated conditions. Our results show that lightweight similarity functions outperform full model conformance for segmentation tasks, offering a practical balance between accuracy and efficiency.

a) Theoretical and Practical Implications:

- *Theory:* We introduce a novel combination of syntactic (Levenshtein, Tversky) and semantic (log skeleton) similarity within a unified segmentation framework. Our proofs of injectivity in activity encoding and the boundary properties of our similarity measures contribute formal insights for future process-mining research.
- *Practice:* The method integrates seamlessly with existing BPM tools (e.g., PM4Py) without modifying ERP system logs. Its lightweight shingle computations and XTructure-based regex checks enable real-time segmentation on production event streams.

III. Preliminaries

We follow standard process mining notation, as introduced in [18]. The goal of process mining is to derive meaningful insights into the execution of business processes from recorded event data. The key input for process mining is the *event log*, which captures sequences of events corresponding to the execution of individual process instances, referred to as *cases*.

A *trace* is a finite sequence of events that represents the observed execution of a single case. By comparing and analyzing these traces, we can uncover patterns, identify deviations, and evaluate performance characteristics of the underlying process.

A *process* P represents the system or workflow whose behavior we aim to analyze. In process mining, the focus is on gaining insights into the execution of P by studying event logs,

which contain traces that correspond to specific executions or instances of process P.

A. Event Logs and Process Mining

We now introduce the formal elements used to represent event data, traces, and structured event logs.

a) Events: Let AN be a finite set of *attribute names*, and let Val be the set of all possible attribute values. We introduce a special symbol $\perp$ (bottom), with $\perp \notin \mathsf{Val}$, to denote "undefined."

The universe of events is a (possibly infinite) set E. Each event $e \in E$ is described by a *partial* function

$$\pi_e \colon \mathsf{AN} \rightharpoonup \mathsf{Val}, \tag{1}$$

where the arrow $\rightharpoonup$ indicates that for some attribute names a the value $\pi_e(a)$ may be undefined (i.e. $\pi_e(a) = \perp$). Concretely, for every $e \in E$ and $a \in \mathsf{AN}$, we have $\pi_e(a) \in \mathsf{Val} \cup \{\perp\}$.

b) Conditions on Events: We impose the following two conditions on every event $e \in E$:

(A1) $\pi_e(time) \neq \perp$, meaning that every event has a timestamp.
(A2) There exists an attribute $a \in \mathsf{AN} \setminus \{time\}$ such that $\pi_e(a) \neq \perp$, meaning every event has at least one non-time attribute with a value not equal to $\perp$.

Definition III.1 (Event Stream)**.** *An* event stream *(or event table) ET is a finite sequence of events:*

$$ET = \langle e_1, e_2, \ldots, e_n \rangle, \quad e_i \in E, \tag{2}$$

such that there exists an attribute $a \in \mathsf{AN} \setminus \{time\}$ with $\pi_a(e_i) \neq \perp$ for all $i = 1, \ldots, n$.

c) Finite Sequences: In what follows, we adopt the standard definition of finite sequences using Kleene star notation. This notation, common in formal language theory, is used throughout this work to denote sequences (e.g., traces of events).

Notation III.2 (Kleene star)**.** *Given a set X, we define X^* to be the set of all finite sequences over X. Formally, each element of X^* is of the form $\langle x_1, x_2, \ldots, x_n \rangle$ for some $n \geq 0$ and $x_i \in X$. The empty sequence is allowed if $n = 0$.*

Let $\Sigma \subseteq \mathsf{Val}$ be a finite set of *activity names*, known as the *activity alphabet*. A *trace* is a finite sequence $\sigma \in \Sigma^* = \{\text{all finite sequences over } \Sigma\}$ representing the control-flow perspective of a case.

Definition III.3 (Case Identifier and Cases)**.** *Let $id \in \mathsf{AN}$ be a selected* case identifier *attribute. The set of cases in ET is defined as:*

$$\mathsf{Cases}(ET, id) = \{\pi_{id}(e) \mid e \in ET,\ \pi_{id}(e) \neq \perp\}. \tag{3}$$

Remark III.4. *Each element $c \in \mathsf{Cases}(ET, id)$ is interpreted as a distinct* case *of the process within ET. Concretely, "case" refers to the actual value c of the attribute id. In this way, $\mathsf{Cases}(ET, id)$ collects exactly those case-identifier values that appear in the event stream ET, ensuring each value corresponds to a unique case.*

Definition III.5 (Trace of a Case)**.** *Let $c \in \mathsf{Cases}(ET, id)$. Then the* trace of case c *is the unique sequence of events*

$$\pi_{trace}(c) = \langle e_1, \ldots, e_k \rangle \in E^* \tag{4}$$

satisfying:

i) For all i, $\pi_{id}(e_i) = c$. In other words, each e_i belongs to the case identified by c.
ii) The sequence $\langle e_1, \ldots, e_k \rangle$ is sorted by non-decreasing timestamp, i.e.

$$\pi_{e_1}(time) \leq \pi_{e_2}(time) \leq \cdots \leq \pi_{e_k}(time), \tag{5}$$

recalling that $e.a := \pi_e(a)$.
iii) $\{e_1, \ldots, e_k\} = \{e \in ET \mid \pi_{id}(e) = c\}$, so we include all *events of that case (no more, no fewer).*

Remark III.6. *For each $c \in \mathsf{Cases}(ET, id)$, the sequence $\pi_{trace}(c)$ is defined as the ordered list of all events $e \in ET$ for which $\pi_{id}(e) = c$, arranged in non-decreasing order of $time$.*

IV. Process Models, Event Logs, and Conformance Checking

A. Process Models

Definition IV.1 (Process Models)**.** *Let Σ be an* activity alphabet *(i.e., a finite set of symbols). Denote by $\mathcal{M}_P(\Sigma)$ the set (or universe) of all process models that describe the behavior of a specific process P and are defined over the activity alphabet Σ. These models belong to a given class $\mathcal{P}$ (e.g., Petri nets, BPMN models, etc.) relevant to this work. An element $M \in \mathcal{M}_P(\Sigma)$ is called a* process model *if it prescribes which traces over Σ are possible in the corresponding process.*

Remark IV.2. *Let P denote a particular business process (e.g. "Order Handling"), and let Σ be its activity alphabet. We write*

$$\mathcal{M}_P(\Sigma) \subseteq \mathcal{P} \tag{6}$$

to mean "the set of all models in the model class $\mathcal{P}$ that describe process P over Σ."

Here:

- *P is the specific process whose behavior we wish to capture.*
- *$\mathcal{P}$ is a* class of modeling formalisms *(for example, Petri nets, BPMN diagrams, log skeletons, etc.).*
- *$\mathcal{M}_P(\Sigma)$ is the subset of $\mathcal{P}$ consisting of those individual models that generate or accept exactly the traces of P over alphabet Σ.*

Definition IV.3 (Induced Language of a Model)**.** *For each process model $M \in \mathcal{M}_P(\Sigma)$, associate a* language $\mathcal{L}_P(M) \subseteq \Sigma^*$*, where Σ^* denotes the set of all finite sequences over Σ. Formally,*

$$\mathcal{L}_P(M) = \{w \in \Sigma^* \mid w \text{ is a possible execution of } M\}. \tag{7}$$

Thus, each $w \in \mathcal{L}_P(M)$ is a finite sequence of activities representing one possible execution of M.

B. Finite Multisets of Traces

Definition IV.4 (Finite Multisets of Traces)**.** *Let X be a set. A* finite multiset *over X is a function $N\colon X \to \mathbb{N}$ such that $\{x \in X \mid N(x) \neq 0\}$ is finite. We call $N(x)$ the* multiplicity *of x in the multiset N. We write $B(X)$ for the set of all finite multisets of elements of X.*

Notation IV.5 (Multisets of Traces)**.** *Since Σ^* is the set of all finite sequences over Σ, the set of all finite multisets of traces is denoted by $B(\Sigma^*)$. An element $L \in B(\Sigma^*)$ is a multiset of traces, where $L(\sigma) \in \mathbb{N}$ indicates the number of occurrences of the trace σ in L.*

C. Structured Event Log

Assume that events are drawn from a universe E, each event $e \in E$ being described by a partial function $\pi_e\colon \mathsf{AN} \rightharpoonup \mathsf{Val}$.

Definition IV.6 (Structured Event Log)**.** *Let $id \in \mathsf{AN}$ be a designated case identifier attribute. A* structured event log *is a finite set of case-trace pairs:*

$$L = \{(c_i, \pi_{trace}(c_i)) \mid c_i \in \mathsf{Cases}(ET, id)\} \subseteq \mathsf{Val} \times E^*, \quad (8)$$

such that for each pair $(c, \pi_{trace}(c)) \in L$, the corresponding trace

$$\pi_{trace}(c) = \langle e_1, \ldots, e_k \rangle \in E^* \quad (9)$$

satisfies:

(i) $\forall e_j \in \pi_{trace}(c)$ *we have* $\pi_{id}(e_j) = c$*;*
(ii) $\forall e_j \in \pi_{trace}(c)$ *we have* $\pi_{time}(e_j) \neq \perp$*;*
(iii) $\forall e_j \in \pi_{trace}(c)\ \exists a \in \mathsf{AN} \setminus \{time, id\}$ *such that* $\pi_a(e_j) \neq \perp$.

Remark IV.7. *Each element $(c, \pi_{trace}(c)) \in L$ represents a single case c together with its corresponding trace. Thus, a structured event log explicitly associates each case identifier with the complete, temporally ordered sequence of events belonging to that case. Specifically, the log has the following hierarchical structure:*

- *A finite set of case-trace pairs L;*
- *Each trace $\pi_{trace}(c) \in E^*$ is a sequence of events related to a single case c;*
- *Every event $e \in \pi_{trace}(c)$ has a timestamp, the associated case identifier c, and at least one additional meaningful attribute (e.g., activity, resource).*

Notice that each case identifier c_i appears exactly once in the set L.

From this structured event log, one can perform further abstraction steps, such as applying *event classifiers*, to derive a *simple event log* $L' \in B(\Sigma^*)$ suitable for control-flow analysis.

D. Process Models and Conformance Checking

When process models are available, conformance checking can be used to compare a candidate trace to a known process structure. This enables semantic similarity beyond syntactic matching, especially useful in process mining when control-flow behavior must be validated.

Let $\mathcal{L}_P \subseteq \Sigma^*$ be a set of example traces for a process P. From this set, we can derive a process model $\mathcal{M}_P(\Sigma)$ using one of several discovery algorithms:

- **Alpha Miner** [19]: Constructs a Petri net by detecting causal and concurrent relations from the event log.
- **Heuristics Miner** [20]: Enhances the Alpha Miner with frequency-based dependency filtering to handle noise.
- **Log Skeleton** [21]: Builds a constraint-based model capturing behavioral relations such as equivalence, always-before, and always-after.

Let $\sigma \in \Sigma^*$ be a trace and $\mathcal{M}_P(\Sigma)$ the process model discovered from $\mathcal{L}_P$. We evaluate how well σ conforms to $\mathcal{M}_P(\Sigma)$ using one of the following conformance techniques:

- **Token-Based Replay Fitness (TBR)** [22]: Evaluates trace fitness by simulating token movement in a Petri net. We use the PM4Py function `pm4py.conformance.fitness_token_based_replay`. This is applied to models discovered via Alpha Miner and Heuristics Miner.
- **Log Skeleton Conformance**: Computes trace fitness by comparing the trace to a log skeleton model using structural constraint diagnostics. We use: `pm4py.conformance_log_skeleton`.

While these model-based approaches capture process semantics, we also employ syntactic techniques that operate on string representations of event sequences. These methods allow for efficient trace comparison when no explicit model is available or required.

Process Discovery Configuration: All discovery was done with PM4Py v2.9.3:

- **Heuristics Miner:** `dependency_threshold=0.99, and_threshold=0.65, or_threshold=0.65, min_support=0.10`.
- **Alpha Miner:** `dependency_threshold=0.50` (PM4Py default).

These were chosen because Heuristics is noise-robust and frequency-based, whereas Alpha provides a simple, well-understood baseline.

E. String Representation and Similarity Measures

To support efficient and scalable process matching, we transform sequences of events into compact string representations that preserve their sequential structure. This enables the application of string-based similarity techniques, including shingling and character-edit distance.

a) Activity Encoding: To enable string-based similarity comparison, each activity in a trace is deterministically mapped to a fixed-length character code. This allows the transformation of activity sequences into strings over a finite alphabet.

Definition IV.8 (Activity Encoding)**.** *Let Σ denote the finite set of activity labels. Define an injective encoding function*

$$C\colon \Sigma \to \Gamma^n, \quad (10)$$

where Γ is a finite character alphabet, and $n \in \mathbb{N}$ is a fixed code length. The set Γ^n represents all possible strings of length n formed from alphabet Γ. This encoding assigns each activity a unique code of length n, but can be extended to $\widetilde{C}\colon \Sigma^* \to (\Gamma^n)^*$ by allowing any trace $\sigma = \langle a_1, \ldots, a_k \rangle \in \Sigma^*$ to be represented as a concatenated string $\widetilde{C}(\sigma) = C(a_1) \cdot C(a_2) \cdot \cdots \cdot C(a_k) \in (\Gamma^n)^*$, where $\widetilde{C}(\langle\rangle)$ is the empty string and $\cdot$ (dot) denotes concatenation.

Remark IV.9 (Practical Consideration:)**.** *In our implementation, we use $n = 3$ and define*

$$\Gamma = \{A, B, \ldots, Z, a, b, \ldots, z, 0, \ldots, 9\}, \tag{11}$$

with $|\Gamma| = 62$. This provides a sufficiently large encoding space for typical process alphabets, ensuring compactness and facilitating downstream operations like shingling.

We now formalize the injectivity of the activity encoding in the following proposition:

Proposition IV.10 (Uniqueness Preservation)**.** *Let $C\colon \Sigma \to \Gamma^n$ be an injective encoding function as defined. Then, for any two distinct traces $\sigma_1, \sigma_2 \in \Sigma^*$, their encoded representations $\widetilde{C}(\sigma_1)$ and $\widetilde{C}(\sigma_2)$ are distinct.*

Proof: We prove the claim by induction on the length k of the sequences.

Base Case: For $k = 0$ (the empty sequence), $\widetilde{C}(\langle\rangle)$ is the empty string, and the claim holds trivially.

Inductive Step: Assume the statement holds for all sequences of length k. Let

$$\sigma_1 = (a_1, a_2, \ldots, a_k, a_{k+1}) \quad \text{and} \quad \sigma_2 = (b_1, b_2, \ldots, b_k, b_{k+1})$$

be sequences in Σ^{k+1} such that

$$\widetilde{C}(\sigma_1) = \widetilde{C}(\sigma_2). \tag{12}$$

Writing

$$\widetilde{C}(\sigma_1) = C(a_1) \cdot C(a_2) \cdots C(a_k) \cdot C(a_{k+1}), \tag{13}$$

and

$$\widetilde{C}(\sigma_2) = C(b_1) \cdot C(b_2) \cdots C(b_k) \cdot C(b_{k+1}), \tag{14}$$

since each $C(a_i)$ is a fixed-length string (of length n), the concatenation is unambiguous. Comparing the first n characters of both sides gives

$$C(a_1) = C(b_1). \tag{15}$$

Because C is injective, it follows that

$$a_1 = b_1. \tag{16}$$

Removing the first code from both concatenations, we then have

$$\widetilde{C}(a_2, \ldots, a_{k+1}) = \widetilde{C}(b_2, \ldots, b_{k+1}). \tag{17}$$

By the induction hypothesis,

$$(a_2, \ldots, a_{k+1}) = (b_2, \ldots, b_{k+1}). \tag{18}$$

Thus,

$$(a_1, a_2, \ldots, a_{k+1}) = (b_1, b_2, \ldots, b_{k+1}). \tag{19}$$

This completes the inductive step, and therefore $\widetilde{C}$ is injective. ∎

b) Shingling: To capture local sequential patterns within the encoded string representation of traces, we apply a fixed-length sliding window operation known as *shingling*.

Definition IV.11 (Shingle Set)**.** *Let $x \in (\Gamma^n)^*$ be an encoded string representation of a trace, and let $k \in \mathbb{N}$ be a fixed window size. The* shingle set *of x is defined as:*

$$\text{Shingle}(x, k) = \{x[i : i+k-1] \mid 1 \le i \le |x| - k + 1\}, \tag{20}$$

where $|x|$ is the length of the string x, and $x[i : i + k - 1]$ denotes the substring of length k starting at position i.

c) Similarity Measures: We employ two types of similarity functions to compare string-encoded traces or their shingle sets.

Definition IV.12 (Similarity Measures)**.**

(1) ***Tversky Similarity.***
Let $A, B \subseteq \mathcal{U}$ be finite sets (e.g., sets of shingles). Define the similarity function:

$$\tau(A, B) := \frac{|A \cap B|}{|A \cap B| + \alpha|A \setminus B| + \beta|B \setminus A|}, \tag{21}$$

where $\alpha, \beta \ge 0$ are tunable parameters (commonly $\alpha = 0$, $\beta = 1$), and $|X|$ denotes the cardinality of set X. This asymmetric variant prioritizes recall over precision, treating containment as more significant than a symmetric match. Consequently, if one set is entirely contained within the other, the similarity measure reaches its maximum value, reflecting a strong inclusion relationship.

(2) ***Levenshtein Similarity.***
Let $a, b \in (\Gamma^n)^*$ be two encoded strings, and let $d(a, b) \in \mathbb{N}_0$ be the Levenshtein distance between them (i.e., the minimum number of single-character insertions, deletions, or substitutions needed to transform a into b). A common normalization for Levenshtein similarity is:

$$\ell(a, b) := 1 - \frac{d(a, b)}{\max\{|a|, |b|\}}. \tag{22}$$

Here, $\ell(a, b)$ yields 1 if the strings are identical and 0 if $d(a, b)$ is as large as the maximum length of the two strings (e.g., $d(a, b) = |b|$ if $|a| = 0$). This approach is widely used due to its intuitive interpretation.

Remark IV.13 (Alternative Normalization)**.** *In some applications, one might prefer a custom normalization to give special treatment to length differences or partial overlaps (e.g., subtracting $\big||a| - |b|\big|$ from $d(a, b)$). If that approach is desired, it must be clearly justified. For example, one might wish to penalize purely length-based mismatches less than mismatches in the overlapping region of the two strings. In such cases, the*

similarity function could be:

$$\ell'(a,b) := \begin{cases} 1, & \text{if } d(a,b) - \big||a| - |b|\big| = 0, \\ \frac{1}{d(a,b) - \big||a| - |b|\big|}, & \text{otherwise.} \end{cases}$$

However, this version is less common and can be confusing without a thorough explanation of the motivation behind it.

This is a well-known property of the above-mentioned similarity measures:

Theorem IV.14 (Boundary Conditions of Similarity Functions)**.** *For any two encoded strings $a, b \in (\Gamma^n)^*$:*

- *The Levenshtein similarity $\ell(a,b)$ satisfies $\ell(a,b) = 1$ if and only if $a = b$, and $\ell(a,b) = 0$ when the number of edits equals the length of the longer string (i.e., maximally different).*
- *For the Tversky index $\tau(A,B)$ (with a typical setting of $\alpha = 0$ and $\beta = 1$), it holds that $\tau(A,B) = 1$ if and only if $B \subseteq A$.*

Remark IV.15. *The normalization in $\ell(a,b)$ subtracts the minimum number of edits required to align the lengths of a and b, isolating the structural mismatches. This ensures that string pairs differing only in length are treated as more similar than those with substantive internal edits.*

V. Problem Statement

In many enterprise systems, users interact with the platform through various interfaces, executing tasks that correspond to different business processes (e.g., onboarding a customer, handling a complaint, reviewing a case). These interactions are recorded as user-specific event streams. However, in most operational settings, this data is captured in a semi-structured format: each event includes the time it occurred, the action taken (activity name), and the user who performed it, but does not explicitly identify which process or case it belongs to.

Let $ET_u = \langle e_1, e_2, \ldots, e_n \rangle$ denote the event stream of a user $u \in \mathsf{Val}$, where each event $e_i \in E$ satisfies:

$$\pi_{user}(e_i) = u, \quad \pi_{time}(e_i) \neq \bot, \quad \pi_{event}(e_i) \in \Sigma. \tag{23}$$

This stream captures a chronologically ordered sequence of user interactions, i.e.

$$\pi_{e_1}(time) \leq \pi_{e_2}(time) \leq \cdots \leq \pi_{e_n}(time), \tag{24}$$

but without any case identifier. As such, it is considered a *semi-structured event stream*.

To support process-aware analysis and monitoring, the business needs to understand which processes a user was involved in, when they started and ended, and what behavior was exhibited within each instance. For example, a supervisor may wish to assess whether an employee followed the correct handling procedure for different types of customer requests.

To enable this, the organization provides, for each process type P_i, $i = 1, \ldots, k$, a set of example traces:

$$\mathcal{L}_{P_i} \subseteq \Sigma^*, \tag{25}$$

which have been collected in a controlled environment. In this setting, expert users executed process cases one at a time, with clear boundaries between cases. These labeled traces serve as templates for recognizing process behavior in live, unstructured event streams.

Definition of the Process-Specific Alphabet. We define

$$\Sigma_{P_i} = \bigcup_{\sigma \in \mathcal{L}_{P_i}} \{\text{activities in } \sigma\} \subseteq \Sigma, \tag{26}$$

i.e., Σ_{P_i} is the set of activity labels that appear in the example traces for process P_i, $i = 1, \ldots, k$.

Several challenges arise in this scenario:

- **Overlapping Activities:** Activities (event names) may appear in multiple processes, i.e. $\Sigma_{P_i} \cap \Sigma_{P_j} \neq \emptyset$ for some $i \neq j$, $i, j = 1, \ldots, k$.
- **Multiple Instances:** The same user u may execute multiple process instances across different process types.
- **Sequential, Possibly Abandoned Executions:** The user performs processes sequentially (never in true parallel) but may abandon one and start another.

a) Objective: Given a semi-structured event stream ET_u and example trace sets $\mathcal{L}_{P_1}, \ldots, \mathcal{L}_{P_k}$ for known processes, the goal is to approximate the user's activity history in terms of which processes were executed and when.

We aim to segment the user stream into trace-like subsequences that are likely to correspond to process instances. Each segment is matched to one of the known processes based on similarity. At this stage, our goal is not to produce a perfect segmentation, but rather a **good estimation** of the user's behavior from a process-centric perspective.

VI. ProcessMatcher Overview

This section introduces **ProcessMatcher**, a two-phase framework for real-time detection of known business processes in semi-structured event streams. In the **offline modeling** phase, curated example traces are filtered for noise and standardized; events are encoded into fixed-length tokens and passed through XTructure to learn hierarchical prefix regexes, alongside a k-shingle extractor that produces compact signatures. Optionally, a light-weight conformance analysis validates pattern quality. In the **online segmentation** phase, incoming streams are partitioned into candidate segments and matched against the learned regexes and shingle signatures using fast approximate similarity measures, yielding polynomial-time detection.

A. Methodology

The ProcessMatcher pipeline comprises three stages:

(a) **Training Data Preparation.** Noisy or incomplete traces are removed; activity labels are normalized; events are encoded as fixed-length character codes (Definition IV.8).
(b) **Pattern Learning.** Normalized sequences feed XTructure for prefix generalization, producing a regex $\mathcal{R}_i$ for each process, and into the k-shingle extractor to compute each process's signature.

(c) **Parameter Tuning.** A grid search over shingle length k, similarity threshold θ, and prefix depth p is conducted on a held-out validation set to jointly optimize classification accuracy and runtime.

This clear separation of phases ensures reproducibility, modularity, and straightforward extensibility of individual components.

B. *XTructure Primer*

`XTructure` [23] is a lightweight, polynomial-time structure for learning and representing the syntax of semi-structured strings (e.g., activity-encoded traces) as a DFA-equivalent model. Unlike full regular-expression inference—which is NP-hard—`XTructure` builds a concise pattern model in a single pass over the data. Its design comprises three hierarchical layers:

1) **Symbol Layer:**
   - Records empirical counts of characters at each position.
   - Groups symbols into character classes (e.g. digit, letter) via a χ^2 homogeneity test when possible, or else enumerates the top-k symbols to cover a fixed fraction of observations.
2) **Token Layer:**
   - Splits input strings on delimiters into tokens.
   - Represents each token as a sequence of symbol-layer distributions, capturing both fixed and variable-length substrings within that token.
3) **Branch Layer:**
   - Accommodates multiple syntactic variants (e.g. date formats) by maintaining up to a bounded number of token-sequence "branches."
   - Creates a new branch when an incoming example's *fit score* exceeds a threshold for all existing branches, then performs branch-and-merge to enforce a global branch limit.

Once learned in a single (streaming) pass, an `XTructure` instance supports:

- **Serialization** to a compact, human-readable regular expression (an OR of its branches).
- **Sampling** of synthetic examples by drawing from its learned symbol distributions.
- **Comparison** to other patterns (regular expressions or other `XTructures`) via Monte Carlo sampling and CLT-based fitness estimation.

This combination of expressivity and efficiency makes `XTructure` orders of magnitude faster than general regex-inference techniques, yet sufficiently powerful to capture all finite-language structure observed in real-world enterprise event data.

C. *ProcessMatcher Architecture*

To identify instances of a process within semi-structured event streams, we introduce the `ProcessMatcher`, a modular component designed to match candidate trace segments to a known process P using a combination of syntactic and conformance-based techniques.

For each known process P_i, $i = 1, \ldots, k$, a separate matcher instance M_i is constructed using its set of example traces $\mathcal{L}_{P_i} \subseteq \Sigma^*$. The matcher learns a lightweight representation of the process, which is then used to detect and score segments within user-specific event streams.

The approach is divided into two phases: an offline *process modeling phase*, and an online *stream segmentation and matching phase*.

D. *Parameter Specification*

Default values and allowable ranges for the key parameters of `ProcessMatcher`:

- **Shingle length k:** default $k = 6$, range $4 \leq k \leq 10$.
- **Similarity threshold θ:** default $\theta = 0.8$, range $0.5 \leq \theta \leq 0.95$.
- **Prefix length for `XTructure`:** default $p = 18$ characters (i.e., 6 activity codes $\times$ 3 chars each), range $9 \leq p \leq 30$.
- **Tversky parameters (α, β):** default $(\alpha, \beta) = (0, 1)$, both non-negative.

The method `contains_process`(S) works as follows:

1) Compute the set of shinglets (character k-grams) from the encoded string S:
$$A = \mathrm{Sh}(S). \tag{27}$$
2) For each reference shinglet set $\mathrm{Sh}(\sigma_j)$ in $\mathcal{L}_{P_i}$, $i = 1, \ldots, k$, set
$$B = \mathrm{Sh}(\sigma_j), \tag{28}$$
then compute the Tversky index $\tau(A, B)$ given by Equation (21).
3) If any $\tau(A, B) \geq \theta$, return `true`; otherwise return `false`.

E. *XTructure Explanation*

We construct a prefix regular expression $\mathcal{R}_i$, $i = 1, \ldots, k$, from observed prefixes of length p using `XTructure`:

1) Initialize: `xpref = XTructure(max_branches=2)`.
2) For each trace $\sigma \in \mathcal{L}_{P_i}$, $i = 1, \ldots, k$, extract `prefix =` σ`[:p]` and call `xpref.learn_new_word(prefix)`.
3) After all prefixes are learned, `str(xpref)` yields a regex, e.g. `(ABC|XYZ)`, capturing the generalized prefix patterns.

F. *Phase 1: Offline Process Modeling*

Given example traces $\mathcal{L}_{P_i}$ of process P_i, $i = 1, \ldots, k$, the `ProcessMatcher` performs the following:

1) **Process Signature Construction:** A process signature is built using *shinglets*, which are sets of overlapping substrings (character n-grams) extracted from traces in $\mathcal{L}_{P_i}$, $i = 1, \ldots, k$. This signature enables fast approximate matching using the Tversky index during online presence checking.
2) **Prefix Pattern Learning:** The initial prefixes of example traces are used to learn a regular expression $\mathcal{R}_i$ that characterizes common process starts. This is done using the

`XTructure` algorithm [23], which generalizes prefixes into a compact, discriminative regex by merging observed patterns. This allows rapid detection of potential case beginnings within user streams.

3) **Optional Process Model Discovery:** If desired, a process model $\mathcal{M}_{P_i}(\Sigma)$ (e.g., Petri net, log skeleton) is discovered from $\mathcal{L}_{P_i}$ using PM4Py, $i = 1, \ldots, k$. This enables conformance-based fitness computation as an alternative to syntactic similarity.

G. Phase 2: Online Stream Segmentation and Matching

Given a user event stream $ET_u = \langle e_1, e_2, \ldots, e_n \rangle$ with $\pi_{user}(e_i) = u$ and no case identifier, we segment and label the stream with likely process instances using the following steps, executed per process matcher M_i, $i = 1, \ldots, k$:

1) **Process Presence Check:** Using the shinglet-based Tversky similarity, we determine whether the user stream contains any segment sufficiently similar to $\mathcal{L}_{P_i}$, $i = 1, \ldots, k$. This acts as a fast filter to skip irrelevant streams.
2) **Prefix Detection:** We apply the learned prefix regular expression $\mathcal{R}_i$ over the stream to detect potential case start positions. Each match defines a candidate window boundary.
3) **Similarity Scoring:** For each candidate segment $\sigma \in \Sigma^*$ (bounded by prefix matches), we compute its similarity to the process using one of several methods:
   - **Tversky Index (default)**: uses shinglets and the Tversky index.
   - **Levenshtein Distance**: normalized edit distance.
   - **Conformance Fitness**: if a process model $\mathcal{M}_{P_i}(\Sigma)$, $i = 1, \ldots, k$, is available, token-based replay or log skeleton diagnostics can be used (e.g. Conformance checking based on Log Skeleton Conformance, Heuristics Miner, and Alpha Miner).
4) **Instance Counting:** The number of likely process instances within the stream is estimated by summing the similarity scores of detected segments. Only segments whose score exceeds a predefined threshold are counted.

This approach offers a flexible, efficient method for process instance identification from flat user event streams, enabling reconstruction of activity histories even in the absence of explicit case identifiers.

Remark VI.1. *Set $\mathcal{M} = \{M_1, \ldots, M_k\}$ denotes a set of process matchers, each of which corresponds to a* different *known procedure (process) $P_i = M_i.\texttt{process}$, $i = 1, \ldots, k$. In other words, for each process P_i we have a separate matcher M_i assigned.*

Remark VI.2. *In Algoithm 1 we use function `EncodeStreamAsActivityString` which is a procedure that transforms a user's event stream ET_u into a string where each character (or substring) represents a specific activity. In practice, for each event $e_i \in ET_u$, this function:*

- *Retrieves the value of the attribute corresponding to the activity (for example, $\pi_{event}(e_i)$).*
- *Maps this value to a fixed character code according to a predefined activity encoding function $C \colon \Sigma \to \Gamma^n$. As defined earlier in Definition IV.8, C is an injective function that assigns each activity $a \in \Sigma$ a unique fixed-length code $C(a) \in \Gamma^n$, where Γ is a finite character alphabet and n is a fixed code length.*
- *Concatenates these codes to construct a string S, which represents the ordered sequence of activities in the stream.*

This string representation enables further operations, such as prefix detection, stream segmentation, and the calculation of similarity measures between segments of events, which are crucial in the process matching algorithm.

Algorithm 1 Segmenting a User Event Stream Using Process Matchers

Require: User event stream $ET_u = \langle e_1, e_2, \ldots, e_n \rangle$
Require: Process matchers $\mathcal{M} = \{M_1, \ldots, M_k\}$
Ensure: Set of labeled trace segments $\mathcal{T}$

$\mathcal{T} \leftarrow \emptyset$
$S \leftarrow$ EncodeStreamAsActivityString(ET_u)
for all $M_i \in \mathcal{M}$ $(i = 1, \ldots, k)$ **do**
  if M_i.contains_process(S) **then**
    $B \leftarrow M_i$.prefix_regex.find_all_matches(S)
    for $j = 1$ **to** $|B| - 1$ **do**
      $\sigma \leftarrow S[B[j] : B[j+1]]$
      $s \leftarrow M_i$.process_similarity(σ)
      if $s \geq \theta$ **then**
        $\mathcal{T} \leftarrow \mathcal{T} \cup \{(\sigma, M_i.\texttt{process})\}$
      end if
    end for
    $\sigma \leftarrow S[B[-1] :]$
    $s \leftarrow M_i$.process_similarity(σ)
    if $s \geq \theta$ **then**
      $\mathcal{T} \leftarrow \mathcal{T} \cup \{(\sigma, M_i.\texttt{process})\}$
    end if
  end if
end for

return $\mathcal{T}$

Proposition VI.3. *Assume that a user event stream ET_u is formed by the concatenation of process traces*

$$ET_u = \sigma_1 \cdot \sigma_2 \cdots \sigma_k, \tag{29}$$

where each $\sigma_i \in \mathcal{L}_P$ (the set of example traces for process P). Suppose further that the learned prefix regular expression $\mathcal{R}$ recognizes exactly the starting positions of any trace in $\mathcal{L}_P$. Then, applying the `ProcessMatcher` segmentation algorithm to ET_u will recover the segments $\sigma_1, \sigma_2, \ldots, \sigma_k$ exactly.

Proof: Let S be the string representation of ET_u obtained by the encoding function $\widetilde{C}$. By assumption, each process instance σ_i corresponds to an encoded substring $s_i = \widetilde{C}(\sigma_i)$

and the overall stream is

$$S = s_1 \cdot s_2 \cdots s_k. \tag{30}$$

By the correctness of the learned prefix regular expression $\mathcal{R}$, for every i we have:

$$\mathcal{R}(S[i]) = \text{true} \quad \Longleftrightarrow \quad i \text{ is the starting position of some } s_j.$$

Let $B = \{i_1, i_2, \ldots, i_k\}$ be the set of positions where $\mathcal{R}$ matches in S, with $i_1 = 1$. The segmentation algorithm then forms segments by taking

$$\sigma_j = S[i_j : i_{j+1} - 1] \quad \text{for } j = 1, \ldots, k-1, \tag{31}$$

and $\sigma_k = S[i_k : |S|]$. Since the stream S is exactly $s_1 s_2 \cdots s_k$, it follows that for each j,

$$\sigma_j = s_j = \widetilde{C}(\sigma_j). \tag{32}$$

Thus, the segmentation algorithm recovers the exact boundaries of each process instance ■

Corollary VI.4. *The segmentation algorithm will always find all process instances if the assumptions about prefixes are satisfied.*

VII. Experiment Design

To evaluate the proposed segmentation method, we conduct two experiments: one semi-synthetic (based on real process traces) and one fully synthetic (generated via process simulation). Both experiments follow a similar structure: we assume that for each process P_i a set of example traces $\mathcal{L}_{P_i}$, $i = 1, \ldots, k$, is available, and we generate user event streams by concatenating process executions. We evaluate the model's ability to identify the underlying process structure from these unstructured streams.

A. Process Setup and Trace Sources

- **Experiment 1 Semi-Synthetic (Real Traces):** We use two real-world processes from our business use case. For each process, we obtain at least 5000 example traces (of various events long). These are used both as reference traces for training the `ProcessMatcher` and as source traces for simulating user activity.
- **Experiment 2 Fully Synthetic (Simulated Processes):** We generate five synthetic process models using the PLG2 framework [24]. For each model, we generate:
  - 20 ideal (noise-free) traces for $\mathcal{L}_{P_i}$, $i = 1, \ldots, k$.
  - 100 perturbed traces using PLG2's default noise settings (e.g., skipped steps, inserted noise, reordering).

B. User Event Stream Synthesis

In both experiments, we generate synthetic user event streams by randomly sampling process traces (ideal or perturbed) and concatenating them. For each user event stream ET_u^{syn}, we apply the following:

- Each user stream contains a random number of process executions (between $[10, 40]$).
- For each selected execution:
  - In the semi-synthetic setting, traces are sampled from the set of ideal real-world traces.
  - In the fully synthetic setting, traces are sampled from the pool of perturbed traces generated via PLG2.
- In the semi-synthetic setting, sampled traces may be further perturbed by truncating their suffix, simulating premature termination of process instances.
- For each synthetic user event stream, we retain the ground truth segmentation: the start and end positions of each process instance, and the corresponding process identity.

This setup yields realistic, unstructured user streams while preserving the ability to evaluate segmentation and classification performance precisely.

C. Iteration and Distribution Variation

To evaluate robustness across different usage distributions, we repeat each experiment over multiple iterations. In each iteration, the probability of selecting each process is varied, producing both balanced and skewed distributions of process executions within streams. This simulates scenarios ranging from uniformly active users to users focused on one or two dominant processes.

D. Evaluation Metrics

Since the true structure of each synthetic user event stream is known, we evaluate performance using the following metrics: **Process Classification Accuracy:** Proportion of predicted segments assigned to the correct process. Together, these experiments allow us to evaluate both the accuracy and robustness of the method under realistic variability and controlled noise.

VIII. Results and discussion

We evaluate our method in two benchmark scenarios, described in Section VIII-A: a semi-synthetic setting based on real process traces, and a fully synthetic setting using processes generated by PLG2 [24]. In both scenarios, we assess the performance of different similarity scoring functions within the segmentation pipeline described earlier. Table I summarizes the accuracy and average processing time for each similarity scoring function, evaluated across both experimental settings.

Each experiment reports both classification accuracy how accurately the detected segments match the correct process and processing time per user stream (in seconds). We compare five similarity or conformance scoring approaches:

- **Levenshtein Similarity**: Edit distance over string-encoded traces.
- **Tversky Similarity**: Shingle-based set similarity.
- **Log Skeleton Conformance** [21]: Constraint-based model alignment.
- **Heuristic Miner + Token Replay** [20]: Frequency-based process model with token-based fitness scoring.
- **Alpha Miner + Token Replay** [19]: Basic Petri net discovery and replay fitness.

TABLE I
RESULTS: ACCURACY AND TIME ON ALL FUNCTIONS ON BOTH EXPERIMENTAL SETUPS

Scoring Function	**Fully Synthetic**		**Semi-Synthetic**	
	Accuracy (%)	Time (s)	Accuracy (%)	Time (s)
Levenshtein Similarity	**99.56**	**81.03**	**100.0**	**7.12**
Tversky Similarity	**91.14**	**94.55**	**99.43**	**14.10**
Log Skeleton Conformance	**79.55**	**11242**	**88.79**	**812**
Heuristic Miner + TBR	**60.8**	**13170**	**53.32**	**1214**
Alpha Miner + TBR	**65.5**	**5572**	**14.78**	**1273**

A. Discussion

As shown in Table I, Levenshtein similarity consistently achieves the highest segmentation and classification accuracy across both experimental settings. It also outperforms the other methods in terms of runtime efficiency, making it a strong default choice.

Tversky similarity, while slightly less accurate, remains competitive and exhibits favorable performance in both accuracy and runtime. Both syntactic similarity methods clearly outperform conformance-based approaches in this segmentation task.

Among the token-replay-based methods, the Log Skeleton model achieves the best accuracy but still falls short of the syntactic techniques. Heuristic Miner and Alpha Miner produce lower classification accuracy and efficiency, suggesting that these models are not well suited to trace-level segmentation in the absence of case structure.

a) Takeaway: For user stream segmentation tasks without explicit case identifiers, syntactic similarity-based scoring functions (Levenshtein and Tversky) provide the best balance between accuracy and computational cost.

IX. THREATS TO VALIDITY

Although the proposed approach shows promising results, several limitations must be acknowledged. We discuss potential threats to validity across four standard dimensions: internal, external, construct, and conclusion validity.

a) Internal Validity: The segmentation algorithm relies on synthetic or semi-synthetic user streams, where process boundaries are derived from known trace sets and artificially injected noise. While this setup allows for precise ground-truth evaluation, it may not capture all forms of real-world user behavior, such as interleaved or overlapping process executions. In addition, prefix learning and similarity scores are sensitive to parameter choices (e.g., shingle length, Tversky threshold), which may influence the outcomes.

b) External Validity: Our experiments are conducted on two real processes and five simulated ones. Although these span diverse scenarios, they may not fully reflect the complexity or variability of processes in other domains (e.g., healthcare, manufacturing). Moreover, user behavior in production systems can involve more irregular or non-sequential activity patterns, which are not entirely accounted for by our current assumptions.

c) Construct Validity: We measure segmentation accuracy using a ground truth built from ideal and truncated traces. However, the assumption that a truncated trace still represents a meaningful process instance may not always hold. Furthermore, our evaluation treats the best-matching process as correct, even though some traces could be ambiguous due to shared activities across multiple processes.

d) Conclusion Validity: Our evaluation metrics focus on classification accuracy and runtime per stream, providing clear comparative insights but excluding statistical significance testing. In future work, we plan to incorporate cross-validation, error analysis, and statistical robustness measures to reinforce confidence in the observed performance differences.

X. CONCLUSION AND FUTURE WORK

In this paper, we propose a method for segmenting semi-structured user event streams into process instances using a combination of syntactic similarity and model-based conformance checking. Our approach leverages known example traces for each process to construct lightweight matcher components that identify candidate segments based on learned prefixes and similarity scoring. This design enables segmentation in scenarios where case identifiers are absent and activities may overlap across processes.

We evaluated the method in two settings: (1) a semi-synthetic setup based on real-world processes, and (2) a fully synthetic benchmark generated with PLG2. The results show that syntactic similarity measures particularly Levenshtein and Tversky offer strong accuracy while remaining computationally efficient. Although semantically richer, model-based conformance approaches performed less effectively in this segmentation context, likely because of their sensitivity to partial traces and noise.

a) Future Work: Several extensions of this work are planned. First, we intend to address the issue of overlapping or conflicting segment assignments when multiple processes match the same trace window. Incorporating overlap resolution strategies such as greedy selection based on similarity scores or probabilistic modeling could yield cleaner segmentations. Second, we aim to evaluate the method on real, unlabeled user data in production environments, with expert annotations serving as ground truth. Finally, integrating sequence models (e.g., RNNs or transformers) or embedding-based representations may further enhance trace similarity beyond symbolic encoding.

Our findings highlight the potential of combining simple syntactic techniques with process mining insights to extract structured information from unstructured logs, thereby enabling more interpretable and process-aware user behavior analysis.

b) Prospects for Sequence Models: In future work, we plan to explore sequence-model approaches (RNNs, LSTMs, or Transformers). This would allow us to:

- Learn continuous embeddings of event sequences instead of manual character codes.
- Capture long-range temporal dependencies and non-linear patterns inherent in complex processes.
- Support online adaptation via incremental fine-tuning on newly observed streams without retraining shingle sets or XTructure models.

These extensions should improve robustness to noise and adaptivity to evolving business logics.

REFERENCES

[1] W. M. van der Aalst, B. F. van Dongen, J. Herbst, L. Maruster, G. Schimm, and A. J. Weijters, "Workflow mining: A survey of issues and approaches," *Data & knowledge engineering*, vol. 47, no. 2, pp. 237–267, 2003.

[2] J. Claes and G. Poels, "Process mining and the prom framework: an exploratory survey," in *Business Process Management Workshops: BPM 2012 International Workshops, Tallinn, Estonia, September 3, 2012. Revised Papers 10*. Springer, 2013, pp. 187–198.

[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, pp. 719–736, 2021.

[4] I. Helal and A. Awad, "Correlating unlabeled events at runtime," *arXiv preprint arXiv:2004.09971*, 2020.

[5] E. Brzychczy, T. Pełech-Pilichowski, and Z. Dworakowski, "Case id detection based on time series data – the mining use case," 2024. [Online]. Available: https://arxiv.org/abs/2410.23846

[6] R. Andrews, C. G. van Dun, M. T. Wynn, W. Kratsch, M. K. Röglinger, and A. H. ter Hofstede, "Quality-informed semi-automated event log generation for process mining," *Decision Support Systems*, vol. 132, p. 113265, 2020.

[7] D. Sanchez-Charles, J. Carmona, and R. Raventos, "Building event logs from relational databases in the XES format," *Applied Sciences*, vol. 12, no. 21, p. 10832, 2022.

[8] C. Kecht, A. Egger, W. Kratsch, and M. Röglinger, "Event log construction from customer service conversations using natural language inference," in *3rd International Conference on Process Mining (ICPM)*, 2021.

[9] Ł. Korzeniowski and K. Goczyła, "Discovering interactions between applications with log analysis," in *2022 17th Conference on Computer Science and Intelligence Systems (FedCSIS)*. IEEE, 2022, pp. 861–869.

[10] C. W. Günther, A. Rozinat, and W. M. van der Aalst, "Activity mining by global trace segmentation," in *Business Process Management Workshops: BPM 2009 International Workshops, Ulm, Germany, September 7, 2009. Revised Papers 7*. Springer, 2010, pp. 128–139.

[11] N. Tax, N. Sidorova, R. Haakma, and W. M. van der Aalst, "Event abstraction for process mining using supervised learning techniques," in *Proceedings of SAI Intelligent Systems Conference (IntelliSys) 2016: Volume 1*. Springer, 2018, pp. 251–269.

[12] N. Martin, B. Depaire, and A. Caris, "The use of process mining in a business process simulation context: Overview and challenges," in *2014 IEEE Symposium on Computational Intelligence and Data Mining (CIDM)*. IEEE, 2014, pp. 381–388.

[13] W. M. van der Aalst, "Object-centric process mining: dealing with divergence and convergence in event data," in *Software Engineering and Formal Methods: 17th International Conference, SEFM 2019, Oslo, Norway, September 18–20, 2019, Proceedings 17*. Springer, 2019, pp. 3–25.

[14] W. M. van der Aalst and A. Berti, "Discovering object-centric Petri nets," *Fundamenta informaticae*, vol. 175, no. 1-4, pp. 1–40, 2020.

[15] A. F. Ghahfarokhi, G. Park, A. Berti, and W. M. van der Aalst, "Ocel: a standard for object-centric event logs," in *European Conference on Advances in Databases and Information Systems*. Springer, 2021, pp. 169–175.

[16] M. J. Jans, M. Alles, and M. A. Vasarhelyi, "Process mining of event logs in auditing: Opportunities and challenges," *International Journal of Accounting Information Systems*, vol. 12, no. 1, pp. 1–20, 2011.

[17] A. Burattin, A. Sperduti, and W. M. van der Aalst, "Control-flow discovery from event streams," in *Proceedings of the IEEE Congress on Evolutionary Computation (CEC)*, 2014, pp. 2420–2427.

[18] D. Fahland, "Extracting and pre-processing event logs," *CoRR*, vol. abs/2211.04338, 2022.

[19] W. M. van der Aalst, A. J. Weijters, and L. Maruster, "Workflow mining: Discovering process models from event logs," *IEEE Transactions on Knowledge and Data Engineering*, vol. 16, no. 9, pp. 1128–1142, 2004.

[20] A. J. Weijters, W. M. van der Aalst, and A. de Medeiros, "Process mining with the heuristics miner algorithm," in *Technical Report WP*, vol. 166, 2003, pp. 1–34.

[21] S. J. van Zelst and W. M. van der Aalst, "Log skeletons: A classification approach to process discovery," in *International Conference on Advanced Information Systems Engineering (CAiSE)*. Springer, 2018, pp. 309–324.

[22] A. Rozinat and W. M. van der Aalst, "Conformance checking of processes based on monitoring real behavior," *Information Systems*, vol. 33, no. 1, pp. 64–95, 2008.

[23] A. Ilyas, J. M. da Trindade, R. Castro Fernandez, and S. Madden, "Extracting syntactical patterns from databases," in *Proceedings of the 2018 International Conference on Management of Data (SIGMOD)*. ACM, 2018, pp. 1773–1788.

[24] A. Burattin and W. M. van der Aalst, "PLG2: Multiperspective process randomization with online and offline simulations," in *International Conference on Business Process Management*. Springer, 2015, pp. 214–219.