Identifying Execution States and Boundary Conditions in Object-Centric Event Logs: A Framework and Case Study

Alessandro Berti[1], Dina Kretzschmann[1], and Wil M.P. van der Aalst[1,2]

[1] Process and Data Science (PADS), RWTH Aachen University, Aachen, Germany
[2] Celonis, Munich, Germany
{a.berti, dina.kretzschmann, wvdaalst}@pads.rwth-aachen.de

Abstract. Understanding how object-centric processes evolve requires discovering execution states directly from event data and explaining the conditions under which transitions occur. This paper introduces an unsupervised framework that infers states and their boundary conditions from sequences of events related to a business object. Each sequence is encoded as numerical vectors and projected into a low-dimensional space. On top of the projected points we train a self-organizing map (SOM) and associate each event, via its surrounding context, to a best-matching neuron, which we interpret as an execution state. Transitions arise from consecutive events mapping to different neurons, and boundary conditions are characterized by the event patterns and contextual signals that precede such changes. The approach uncovers non-obvious states that are not necessarily tied to any single data attribute, and it supports both within-state comparisons (which events tend to co-occur in a state) and between-state analyses (which event patterns drive movement across states). We provide a prototypal implementation and a case study in an inventory management context.

Keywords: Object-Centric Process Mining · Execution State Discovery · Boundary Conditions · Self-Organizing Maps (SOM) · Inventory Management

1 Introduction

Business processes are coordinated sets of activities that transform inputs into products and services across organizational roles, systems, and data. *Business Process Management* (BPM) provides the discipline, methods, and tools to design, analyze, monitor, and continuously improve these processes for goals like efficiency, compliance, and resilience. Within BPM, *Process Mining* (PM) exploits timestamped event data from operational systems to discover process models, check conformance, and analyze performance, bridging execution and prescription. Classical PM uses a case-centric view where each event belongs to a single case, simplifying modeling but obscuring many-to-many interactions between events and business objects (e.g., an event concerning an order, material, and delivery). *Object-Centric Process Mining* (OCPM) addresses this by

representing events with links to all affected objects. *Object-Centric Event Logs* (OCELs) thus retain the relational structure of processes, exposing richer yet more heterogeneous and context-dependent behavioral streams. Object-centric process models, computed on OCELs, are often difficult to read because multiple case notions intertwine. Events simultaneously refer to different objects and object types, so alternative "cases" overlap and cross-cut one another, yielding dense models with many parallel paths and loops.

This complexity is amplified when the log contains *long-lived objects*, objects whose lifecycles comprise hundreds or thousands of events. In inventory management, for instance, a single material (SKU) may accumulate years of activity: repeated *goods issue* and *goods receipt* events, *purchase suggestion* $\rightarrow$ *purchase order* cycles, stock corrections, reservations, and returns. A model that tries to summarize the full lifecycle of such an object easily conflates episodic phenomena (e.g., a short promotion) with structural patterns (e.g., the replenishment policy), making interpretation difficult. As an example, long runs of *goods issue* without compensating receipts signal a drift toward understock, whereas frequent receipts with weak demand indicate emerging overstock; these signals are hard to isolate in a global object-centric model.

To make such processes understandable, the notion of *state* comes into play. States abstract from the fine-grained event stream and capture the situational posture of an object at a point in time, while remaining grounded in what actually happens in the log. We define an *execution state* for an object as a *recurrent configuration of short-range event patterns, optionally augmented with contextual signals* (e.g., stock level, lead times, workload), that characterizes the object's situation without assuming a predetermined lifecycle. Consecutive events that share similar local patterns belong to the same execution state; when the pattern changes sufficiently, the object transitions to another state.

Anticipating these state changes is critical, especially when trajectories head toward unfavorable states such as understock or chronic overstock. Early warnings enable proactive interventions (expediting or throttling orders, reallocating inventory, or adjusting planning parameters) before costs materialize as stock-outs, excess holding, or service degradation. To explain and predict such changes, we introduce *boundary conditions*: observable combinations of events and context that significantly increase the likelihood of moving from one execution state to another within a short horizon. Boundary conditions pinpoint *why* a transition is imminent (e.g., a burst of sales orders with lagging receipts) and provide actionable triggers for monitoring and control.

Guided by these concepts, the goal of this paper is to *discover execution states directly from OCELs and to characterize the boundary conditions that precipitate transitions between them*, with an explicit focus on object-centric processes involving long-lived objects. Our running example is inventory management, where materials commonly exhibit long lifecycles and where distinguishing normal operation from emerging under- or over-stock is operationally critical.

We follow a three-step philosophy: `window`, `compress`, `map`. First, slide a short, overlapping *window* over each object's event history to capture local tem-

poral context without rigid stages. Second, *compress* each window via dimensionality reduction (using principal component analysis, PCA) to reveal recurring patterns and dampen noise. Third, *map* the points onto a two-dimensional self-organizing map (SOM) grid where neighboring cells encode similarity; trajectories reveal state evolution, and boundary-crossing windows instantiate change conditions. Transitions track neuron changes over time; boundary conditions compare intra- versus inter-state windows to highlight preceding event patterns and contextual signals. This unsupervised pipeline operates directly on OCELs and incorporates data signals like stock levels. In an inventory management case study, we compare states from activities alone versus activities plus stock level, contrasting both with optimization-defined states to show how stock information reshapes the state landscape and clarifies transitions.

The rest of the paper is organized as follows: Section 2 reviews related work on process-centric state discovery and sequence representation. Section 3 introduces notation and basic concepts. Section 4 describes the sliding-window, PCA, and SOM steps. Section 5 reports the inventory management study and findings. Section 6 concludes and outlines future directions.

2 Related Work

This section reviews prior work relevant to the discovery and analysis of execution states and boundary conditions in process mining, with a focus on object-centric perspectives. We cover three key areas: state-centric approaches tailored to inventory management, broader state-based techniques in process mining, and methods for representing and encoding event sequences.

2.1 State-centric Analyses for Inventory Management

Recent work has argued that inventory management benefits from making states first-class citizens in object-centric analyses. One line of work introduces an object-centric logging and analysis approach where states (e.g., *Understock*, *Normal*, *Overstock*) are explicitly encoded and transitions between them are logged as events. This "state-aware" view extends object-centric logging by adding state transition events and by enriching activity occurrences with the current object state. The goal is to diagnose when, why, and how inventories deviate from desirable regions and to support interventions grounded in those transitions [2]. A complementary line of work focuses on segmenting very long object lifecycles into coherent slices to reduce model complexity and to enable targeted analyses and optimization. Segmentation strategies use object relations, event types, and time intervals to isolate meaningful windows of behavior, which can then be analyzed or optimized in isolation or in sequence [1]. Together, these contributions show how an explicit notion of state, and careful temporal segmentation, can make object-centric analyses actionable for inventory settings, while remaining compatible with standard object-centric representations [2,1].

2.2 State Understanding in Process Mining

The concept of representing processes through states is well-established in process mining. Composite State Machines (CSMs) discover state-based models cap-

turing multiple perspectives and their interactions from event data, enabling interactive artifact-centric exploration of state co-evolution and performance [4]. Artifact-centric process mining formalizes processes around evolving business entities with explicit lifecycles, developing techniques for lifecycle discovery, relation, and conformance checking [5,9]. Our approach aligns with this state-centric view but derives states from short-range event patterns (not predefined stages) and characterizes transitions by contrasting intra- and inter-state windows, ensuring data-driven interpretability [4,5,9].

2.3 Event Sequence Representation

Representing event sequences numerically while preserving order and context is key in process mining and predictive analytics. Explicit encodings like index-based, last-state, and aggregation summarize prefixes or traces interpretably and have been benchmarked across tasks [11,12]. Learned representations include embeddings adapting language modeling to processes [7] and deep models like LSTMs for prediction, balancing accuracy and transparency [3]. Surveys show encoding choice depends on log traits, targets, and interpretability needs [11,6]. Orthogonally, representations can be global (whole-trace) or local (fragments). Windowed/n-gram approaches capture short-range order akin to time-series techniques [13] and recent process mining for prediction [10]. Our sliding windows with dimensionality reduction follow this local strand, maintaining simplicity and facilitating analysis of patterns and transitions.

3 Preliminaries

We introduce key concepts and techniques that enable the transformation of raw object-centric event data into interpretable state representations. We begin by formalizing object-centric event logs. We then review PCA for dimensionality reduction, and SOMs for projecting these patterns onto a discrete grid of execution states.

3.1 Object-Centric Event Logs and Event-Level Vectorization

We adopt the standard notion of an object-centric event log in [8]. Let $\mathcal{U}\sigma$ denote the universe of all strings. An *object-centric event log* (OCEL) is a tuple $(E, O, A, OT, \pi_{act}, \pi_{time}, \pi_{ot}, \pi_{omap}, \pi_{vmap}, \pi_{ovmap}, \leq)$, where E is a finite set of unique events, O is a finite set of unique objects, A is a finite set of activities, OT is a finite set of object types, $\pi_{act} : E \to A$ assigns to each event an activity, $\pi_{time} : E \to \mathbb{R}$ assigns each event a timestamp, $\pi_{ot} : O \to OT$ assigns each object to an object type, $\pi_{omap} : E \to \mathcal{P}(O)$ assigns each event to a set of related objects, $\pi_{vmap} : E \times \mathcal{U}\sigma \nrightarrow \mathcal{U}\sigma \cup \mathbb{R}$ assigns values to event-attribute pairs, $\pi_{ovmap} : O \times \mathcal{U}\sigma \times \mathbb{R} \nrightarrow \mathcal{U}\sigma \cup \mathbb{R}$ assigns values to object-attribute-timestamp triples, and $\leq \subseteq E \times E$ is a total order on events defined by $e_1 \leq e_2 \iff \pi_{time}(e_1) \leq \pi_{time}(e_2)$, with tie-breaking for equal timestamps.

3.2 Principal Component Analysis (PCA)

Principal Component Analysis (PCA) is a dimensionality reduction technique that projects high-dimensional data onto a lower-dimensional subspace while

preserving as much variance as possible. This helps in compressing event window encodings without losing key patterns.

Linear reduction and reconstruction. Let $x \in \mathbb{R}^n$ be a feature vector (e.g., the numeric encoding of an event window). PCA performs a linear reduction to dimension $d < n$ using a matrix $A \in \mathbb{R}^{d \times n}$ to obtain $z := A(x - \mu) \in \mathbb{R}^d$, where $\mu \in \mathbb{R}^n$ is the mean of the data. This centers the data to focus on variations. To reconstruct an approximation $\hat{x}$ from z, use $\hat{x} := \mu + A^+ z$, where A^+ is the Moore-Penrose pseudoinverse. When $A = W^\top$ with $W \in \mathbb{R}^{n \times d}$ having orthonormal columns, reconstruction simplifies to $\hat{x} = \mu + Wz$.

Measuring reconstruction error. The quality of the reduction is assessed by how well $\hat{x}$ approximates x. The squared reconstruction error for a single vector is $e(x) := \|x - \hat{x}\|_2^2$, using the Euclidean norm. For a dataset $\{x^{(1)}, \ldots, x^{(m)}\}$, the average error is $\frac{1}{m} \sum_{j=1}^{m} \|x^{(j)} - \hat{x}^{(j)}\|_2^2$. PCA chooses the subspace that minimizes this average error over all possible rank-d linear mappings.

Eigenvectors, eigenvalues, and the PCA directions. To find the optimal directions, center the data matrix X to get $X_c \in \mathbb{R}^{m \times n}$, then compute the sample covariance matrix $\Sigma := \frac{1}{m} X_c^\top X_c \in \mathbb{R}^{n \times n}$. Perform eigen-decomposition: $\Sigma v_i = \lambda_i v_i$ for $i = 1, \ldots, n$, with eigenvalues $\lambda_1 \geq \lambda_2 \geq \cdots \geq \lambda_n \geq 0$ and orthonormal eigenvectors v_i. These v_i point along directions of maximum variance, with λ_i quantifying the variance in that direction. PCA selects the top d eigenvectors as columns of $W = [v_1, \ldots, v_d]$. The reduced coordinates are $z = W^\top(x - \mu)$, and reconstruction is $\hat{x} = \mu + Wz = \mu + WW^\top(x - \mu)$. The variance captured by the first d components is $\sum_{i=1}^{d} \lambda_i$, and the explained variance ratio is $\mathrm{EVR}(d) := \frac{\sum_{i=1}^{d} \lambda_i}{\sum_{i=1}^{n} \lambda_i}$, often used to measure how much information is retained.

Choosing the dimension via the elbow rule. To select d automatically, examine the residual variance $E(d) := \sum_{i=d+1}^{n} \lambda_i$, which decreases as d increases. Plot $E(d)$ for $d = 1, \ldots, d_{\max}$ and identify the "elbow" where the rate of decrease slows, indicating diminishing returns. Normalize to the unit square: $\tilde{d} = \frac{d-1}{d_{\max}-1}$, $\tilde{E}(d) = \frac{E(d) - E(d_{\max})}{E(1) - E(d_{\max})}$. Consider the line L from (0,1) to (1,0), with equation $x + y - 1 = 0$. Select $\hat{d} := \arg\max_d \frac{|\tilde{d} + \tilde{E}(d) - 1|}{\sqrt{2}}$, the point farthest from L. This choice balances compactness and fidelity, ensuring the reduced representations retain essential structure for subsequent state mapping.

3.3 Self-Organizing Maps (SOM)

Self-Organizing Maps (SOMs) are unsupervised neural networks that map high-dimensional data to a low-dimensional (typically 2D) grid, preserving topological relationships so similar inputs cluster nearby. This topology-preserving property makes SOMs useful for visualizing and clustering complex data, such as compressed event patterns, into interpretable states.

From high-dimensional vectors to a 2D grid. An SOM maps inputs from $\mathbb{R}^n$ to a 2D grid of size $M_x \times M_y$, with coordinates $\mathcal{G} := \{(p, q) : p = 1, \ldots, M_x, q = 1, \ldots, M_y\}$. Each position $r_i = (p_i, q_i)$ has a prototype vector $w_i \in \mathbb{R}^n$. For input $x \in \mathbb{R}^n$, find the best-matching unit (BMU) $b(x) := \arg\min_i \|x - w_i\|_2$ (Euclidean distance), and map to $f(x) = r_{b(x)}$. Nearby inputs tend to map to adjacent cells, allowing the grid to summarize recurring patterns in contiguous regions.

How the grid is learned (training). Training initializes prototypes (e.g., randomly or via PCA) and iterates over $t = 0, \ldots, T-1$: select input x, find BMU $b = \arg\min_i \|x - w_i(t)\|_2$, then update $w_i(t+1) = w_i(t) + \eta(t) h_{b,i}(t)(x - w_i(t))$. Here, $\eta(t)$ is the decaying learning rate, $\sigma(t)$ the shrinking neighborhood radius, and $h_{b,i}(t) = \exp\left(-\frac{\|r_b - r_i\|_2^2}{2\sigma(t)^2}\right)$ a Gaussian kernel. Neighbor updates ensure topology preservation: close inputs activate nearby cells.

Using the trained map. Post-training, $f(x) = r_{b(x)}$ assigns each x deterministically to its closest prototype's cell. Prototypes w_i represent data patterns, quantization error $\|x - w_{b(x)}\|_2^2$ measures fitness, and grid regions form clusters.

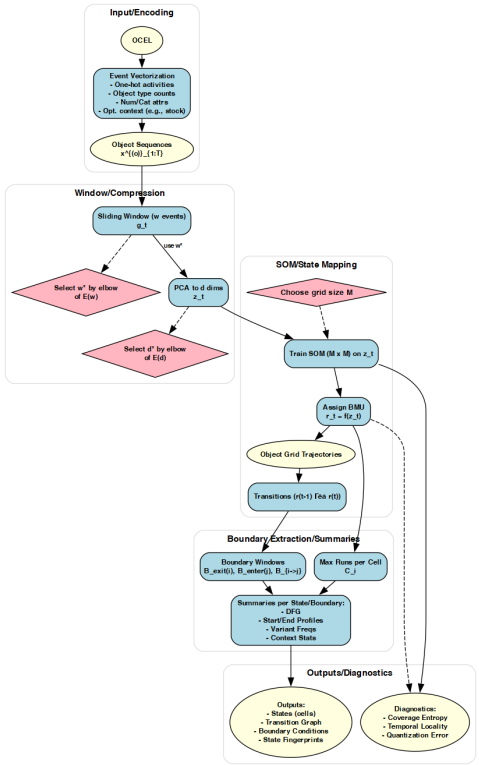

Fig. 1: Overview of the pipeline: `window` → `compress` → `map`.

In our approach, cells denote execution states; object trajectories on the grid reveal evolutions, with BMU changes signaling transitions.

4 Approach

Our unsupervised pipeline (summarized in Figure 1) discovers execution states and boundary conditions from object-centric event logs. Events are transformed into numerical vectors encoding activities, object relationships, and attributes. These vectors are aggregated into sliding windows per object and compressed via PCA to create compact, low-dimensional representations of local behavioral contexts. Projections onto a 2D self-organizing map assign grid cells as distinct states, with object trajectories tracing evolutions. Boundary conditions are characterized by comparing within-state event patterns to those precipitating transitions, enabling analyses of stability and change drivers.

4.1 From OCELs to Per-Event Vectors

Given the definition of OCEL presented in Section 3.1, events can be transformed into numerical vectors as follows. Let A be the set of activities, OT the set of object types, and let K_{num} and K_{str} denote, respectively, the sets of numeric and string (categorical) event-attribute names we decide to include from π_{vmap}. We define an encoding map $x : E \longrightarrow \mathbb{R}^n$ by concatenating four simple, interpretable blocks:

1. *Activity (one-hot).* Encode the activity $\pi_{\text{act}}(e)$ as a one-hot vector $\text{onehot}_A(\pi_{\text{act}}(e)) \in \{0,1\}^{|A|}$.
2. *Participation counts by object type.* For each $t \in OT$, count how many related objects of type t the event touches: $c_t(e) := \big|\{o \in \pi_{\text{omap}}(e) : \pi_{\text{ot}}(o) = t\}\big| \in \mathbb{R}_{\geq 0}$. The vector $\big(c_t(e)\big)_{t \in OT} \in \mathbb{R}^{|OT|}$ summarizes the event's "fan-out" across object types.
3. *Numeric event attributes (as-is).* For each $k \in K_{\text{num}}$, take the numeric value $v_k(e) \in \mathbb{R}$ if present, yielding a block in $\mathbb{R}^{|K_{\text{num}}|}$. In practice we standardize these features (zero mean, unit variance) and, when values are missing, impute a neutral value (e.g., the mean) and add a binary "is-missing" flag if desired.
4. *String event attributes (one-hot).* For each $k \in K_{\text{str}}$ with observed value set $V_k \subseteq \Sigma$, encode $\pi_{\text{vmap}}(e,k)$ via $\text{onehot}_{V_k \cup \{\text{UNK}\}}$, sending unseen/rare categories to UNK. Concatenating over $k \in K_{\text{str}}$ gives a sparse block whose dimensionality equals $\sum_{k \in K_{\text{str}}} |V_k \cup \{\text{UNK}\}|$.

Putting the blocks together,

$$x(e) := \Big[\underbrace{\text{onehot}_A(\pi_{\text{act}}(e))}_{|A|} \Big| \underbrace{(c_t(e))_{t \in OT}}_{|OT|} \Big| \underbrace{(v_k(e))_{k \in K_{\text{num}}}}_{|K_{\text{num}}|} \Big| \underbrace{\bigoplus_{k \in K_{\text{str}}} \text{onehot}_{V_k \cup \{\text{UNK}\}}(\pi_{\text{vmap}}(e,k))}_{\sum_k |V_k| + |K_{\text{str}}|}\Big] \in \mathbb{R}^n$$

where $\oplus$ denotes concatenation. The total dimension is

$$n := |A| + |OT| + |K_{\text{num}}| + \sum_{k \in K_{\text{str}}} \big(|V_k| + 1\big).$$

Optionally, time-related features such as inter-event time can be added as extra numeric attributes using π_{time}. Likewise, object attributes from π_{ovmap} may be "snapshotted" at $\pi_{\text{time}}(e)$ and aggregated across related objects (for example, averaging numeric values per type or one-hot encoding categorical values), then appended as additional blocks.

Per-object event sequences. To analyze the evolution of a specific object $o \in O$ (e.g., a material), we collect its related events $E(o) = \{e \in E \mid o \in \pi_{\text{omap}}(e)\}$ and sort them by the log's total order $\leq$. This yields a time-ordered sequence of vectors

$$x^{(o)}_{1:T} := \big(x(e_1), x(e_2), \ldots, x(e_T)\big),$$

with $e_1 \leq e_2 \leq \cdots \leq e_T$. Each $x(e_t)$ encodes what happened at e_t (activity, participation breadth, and attributes), providing a uniform numeric representation on which we later apply sliding windows, dimensionality reduction, and state mapping.

4.2 From Sequences of Events to Meaningful Representations

We start from the per-event numeric vectors introduced in Section 3.1. For a given object o, these yield an ordered sequence $(x^{(o)}_{1:T})$ with $x^{(o)}_t \in \mathbb{R}^n$. Our goal is to summarize short-range behavioral context in a way that is both compact and informative for state discovery. To do so, we consider sliding windows of consecutive events. For a candidate window size w, each position $t \geq w$ produces a window vector by simple concatenation,

$$g^{(o)}_t := \big[x^{(o)}_{t-w+1} | x^{(o)}_{t-w+2} | \cdots | x^{(o)}_t\big] \in \mathbb{R}^{wn},$$

where $|$ denotes vector concatenation. Intuitively, $g^{(o)}_t$ captures "what has just happened" before event t: if w is too small, we lose context; if w is too large, we risk mixing unrelated fragments. We build such windows for all objects and all admissible positions, obtaining a dataset of overlapping, fixed-length fragments. To identify a window size that carries meaningful information for state representation, we compress each window with PCA (Section 3.2). For every candidate w in a reasonable range, we form the corresponding window dataset in $\mathbb{R}^{wn}$, apply PCA to a small, fixed probe dimension d_{probe} (e.g., a handful of components), and compute the average reconstruction error across all windows. This yields an error curve $E(w)$ that typically decreases as w grows, then levels off. We select the window size $w^\star$ at the "elbow"—the smallest w beyond which gains become marginal. In effect, $w^\star$ balances context and parsimony: it is long enough to capture the patterns that matter for distinguishing states, yet short enough to avoid diluting them. With $w^\star$ fixed, we revisit the same windows and choose the intrinsic dimensionality d for PCA. We compute the reconstruction error as a function of the number of components $d = 1, 2, \ldots$ and again pick the elbow $d^\star$. The result is a low-dimensional representation $z^{(o)}_t \in \mathbb{R}^{d^\star}$ for each window, obtained by projecting $g^{(o)}_t$ onto the top $d^\star$ principal directions. Each event can then inherit the representation of a consistent neighboring window (e.g., the

trailing window that ends at that event), producing an event-aligned sequence of compact vectors. These sequences serve as the input to the next step, where we map them onto a two-dimensional grid of states using a self-organizing map (Section 3.3).

4.3 From Meaningful Representations to States

We take the dimensionally reduced window vectors from Section 4.2, denoted $z_t^{(o)} \in \mathbb{R}^{d^\star}$, and train a Self-Organizing Map (Section 3.3) on the pooled set $Z = \{z_t^{(o)}\}$. A SOM with a square grid of size $M \times M$ maintains one prototype $w_i \in \mathbb{R}^{d^\star}$ for each grid coordinate $r_i = (p_i, q_i)$ with $p_i, q_i \in \{1, \ldots, M\}$. After training, each window vector is assigned to its best-matching prototype $b(z) = \arg\min_i \|z - w_i\|_2$, which yields a grid coordinate $r_{b(z)}$. An object's evolution is then visible as a path of grid cells $(r_{b(z_t^{(o)})})_t$; we interpret each cell as an execution state and read state changes when the path moves to a different cell.

To choose the grid size M, we balance two desiderata: *coverage* (points should not collapse onto a few cells) and *smoothness* (neighboring events in a trace should map to the same or nearby cells). For *coverage*, we compute the cross-neuron entropy. Let $N = |Z|$ be the number of window vectors, let $n_i = \#\{z \in Z : b(z) = i\}$ be the count assigned to cell i, and let $p_i = n_i/N$ be the empirical occupancy. The entropy is $H(M) = -\sum_{i=1}^{M^2} p_i \log p_i$ (with the convention $0 \log 0 = 0$). We normalize it by the maximum $\log(M^2)$ to obtain $\tilde{H}(M) = H(M)/\log(M^2) \in [0, 1]$, where values closer to 1 indicate a more even spread across the grid.

For *smoothness*, we measure neighbor assignment along traces. For each object o and each consecutive pair $(z_{t-1}^{(o)}, z_t^{(o)})$, let $r_{t-1}^{(o)} = r_{b(z_{t-1}^{(o)})}$ and $r_t^{(o)} = r_{b(z_t^{(o)})}$. We define a grid distance $d_\infty(r, r') = \max\{|p - p'|, |q - q'|\}$, which equals 0 for the same cell and 1 for any of the eight immediate neighbors. The neighbor-preservation rate is $R(M) = \frac{1}{N'} \sum \mathbf{1}_{d_\infty(r_{t-1}^{(o)}, r_t^{(o)}) \le 1}$, where the sum runs over all consecutive pairs across objects and N' is their total count. Values of $R(M)$ near 1 indicate that the SOM preserves local temporal continuity: successive windows for the same object tend to remain in the same cell or move to an adjacent one.

We evaluate candidate sizes $M \in \{M_{\min}, \ldots, M_{\max}\}$ and pick the one that best trades off even coverage and smooth temporal locality. A simple choice is to maximize the combined score $J(M) = \alpha \tilde{H}(M) + (1 - \alpha) R(M)$ with $\alpha \in [0, 1]$ (by default $\alpha = 0.5$), breaking ties in favor of the smaller M for interpretability. The resulting grid yields a state space in which cells are well utilized and successive events move gradually, making state identification and boundary analysis straightforward.

4.4 From States to Boundary Conditions

Given the trained SOM and the event-aligned, dimensionally reduced windows $(z_t^{(o)} \in \mathbb{R}^{d^\star})$ (with $t = w^\star, \ldots, T^{(o)}$), we map each window to its best-matching neuron and obtain a sequence of grid positions per object: $r_t^{(o)} = f(z_t^{(o)}) \in \{1, \ldots, M\} \times \{1, \ldots, M\}$. We view each grid position as an execution state;

thus, for object o, the path $(r_t^{(o)})_t$ describes its state evolution over time. Let the underlying ordered events for o be $(e_1^{(o)}, \ldots, e_{T^{(o)}}^{(o)})$. For every $t \geq w^\star$, the window $g_t^{(o)} = (e_{t-w^\star+1}^{(o)}, \ldots, e_t^{(o)})$ is the contiguous list of event identifiers that produced $z_t^{(o)}$.

Boundary conditions are extracted whenever two consecutive windows land in different states. Formally, a transition occurs at (o, t) if $r_{t-1}^{(o)} \neq r_t^{(o)}$. In that case, we treat the window $g_t^{(o)} = (e_{t-w^\star+1}^{(o)}, \ldots, e_t^{(o)})$ as the *sequence leading to the change*. We record it on both sides of the boundary: as an *exiting sequence* for the origin state $r_{t-1}^{(o)}$ and as an *entering sequence* for the destination state $r_t^{(o)}$. Denoting by $\mathcal{B}_i^{\text{exit}}$ and $\mathcal{B}_j^{\text{enter}}$ the multisets of exiting/entering sequences maintained per grid cell i and j, the update rule is:

$$\text{if } r_{t-1}^{(o)} = i \text{ and } r_t^{(o)} = j \neq i, \quad \mathcal{B}_i^{\text{exit}} \leftarrow \mathcal{B}_i^{\text{exit}} \cup \{g_t^{(o)}\}, \qquad \mathcal{B}_j^{\text{enter}} \leftarrow \mathcal{B}_j^{\text{enter}} \cup \{g_t^{(o)}\}.$$

When it is useful to keep the "before" and "after" contexts distinct, we additionally store $g_{t-1}^{(o)} = (e_{t-w^\star}^{(o)}, \ldots, e_{t-1}^{(o)})$ specifically as the exiting sequence for $r_{t-1}^{(o)}$ and $g_t^{(o)}$ as the entering sequence for $r_t^{(o)}$. For pairwise analysis of adjacent states, we also maintain $\mathcal{B}_{i \to j} = \{g_t^{(o)} \mid r_{t-1}^{(o)} = i, r_t^{(o)} = j\}$, which collects all boundary windows observed at the $i \to j$ interface.

In parallel, we summarize *within-state* behavior by collecting all subsequences of original events that are completely contained in a given state. For a fixed cell i, define the set of maximal index intervals for object o,

$$\mathcal{I}_i^{(o)} = \big\{[a, b] : w^\star \leq a \leq b \leq T^{(o)},\ r_t^{(o)} = i \text{ for all } t \in [a, b], \\ \text{and } r_{a-1}^{(o)} \neq i,\ r_{b+1}^{(o)} \neq i \qquad \text{when defined}\big\}.$$

The sequences $\mathcal{C}_i = \bigcup_o \{(e_a^{(o)}, \ldots, e_b^{(o)}) : [a, b] \in \mathcal{I}_i^{(o)}\}$ are the event-identifier lists that lie wholly within state i. Together, the state-conditioned collections $(\mathcal{C}_i)$ (in-state evidence) and the boundary-conditioned collections $(\mathcal{B}_i^{\text{exit}}, \mathcal{B}_i^{\text{enter}})$ (change evidence) provide a precise, data-driven basis to compare states and to characterize what patterns tend to precede exits and facilitate entries.

Summarizing in-state and boundary behavior as activity patterns. Each of the multisets of event-identifier sequences introduced above—namely, the in-state collection $\mathcal{C}_i$ and the boundary collections $\mathcal{B}_i^{\text{exit}}$ and $\mathcal{B}_i^{\text{enter}}$—can be projected to activity sequences using the activity map π_{act}. For any sequence of events $s = (e_1, \ldots, e_L)$, let $\alpha(s) = (a_1, \ldots, a_L)$ with $a_t = \pi_{\text{act}}(e_t)$. Given a multiset $\mathcal{S} \subseteq E^\star$ of such sequences (we will instantiate $\mathcal{S}$ with $\mathcal{C}_i$, $\mathcal{B}_i^{\text{exit}}$, or $\mathcal{B}_i^{\text{enter}}$), we derive four standard summaries that make comparisons across states and boundaries straightforward.

First, the *directly-follows* counters define a weighted directed graph over activities. For activities $a, b \in A$, the count is $C_{a \to b}^{\mathcal{S}} := \sum_{s \in \mathcal{S}} \sum_{t=1}^{|\alpha(s)|-1} \mathbf{1}\{\alpha(s)_t = a,\ \alpha(s)_{t+1} = b\}$, which records how often a is immediately followed by b within the sequences of $\mathcal{S}$. This yields a directly-follows graph (DFG) with edge weights

$C^{\mathcal{S}}_{a\to b}$. Second, start-activity usage is summarized by $S^{\mathcal{S}}_a := \sum_{s\in\mathcal{S}} \mathbf{1}\{\alpha(s)_1 = a\}$, counting how many sequences in $\mathcal{S}$ begin with activity a. Third, end-activity usage is captured by $E^{\mathcal{S}}_a := \sum_{s\in\mathcal{S}} \mathbf{1}\{\alpha(s)_{|\alpha(s)|} = a\}$, counting how many sequences end with a. Fourth, *variants* aggregate identical activity strings. Let $A^\star$ denote all finite sequences over A. For a variant $v \in A^\star$, its frequency in $\mathcal{S}$ is $F^{\mathcal{S}}(v) := \#\{s \in \mathcal{S} : \alpha(s) = v\}$. In practice we also report the support $F^{\mathcal{S}}(v)/|\mathcal{S}|$ to compare sets of different size.

Applying these summaries to $\mathcal{C}_i$ yields an "in-state" DFG $C^{\mathcal{C}_i}_{\cdot\to\cdot}$, along with start and end profiles $(S^{\mathcal{C}_i}_a)_a$ and $(E^{\mathcal{C}_i}_a)_a$, and a variant spectrum $F^{\mathcal{C}_i}(\cdot)$ that characterizes what typically happens while the process remains in state i. Doing the same for $\mathcal{B}^{\text{exit}}_i$ and $\mathcal{B}^{\text{enter}}_i$ provides complementary "near-boundary" views: $C^{\mathcal{B}^{\text{exit}}_i}_{\cdot\to\cdot}$, $S^{\mathcal{B}^{\text{exit}}_i}_a$, $E^{\mathcal{B}^{\text{exit}}_i}_a$, $F^{\mathcal{B}^{\text{exit}}_i}(\cdot)$ summarize activity flows immediately before leaving i; $C^{\mathcal{B}^{\text{enter}}_i}_{\cdot\to\cdot}$, $S^{\mathcal{B}^{\text{enter}}_i}_a$, $E^{\mathcal{B}^{\text{enter}}_i}_a$, $F^{\mathcal{B}^{\text{enter}}_i}(\cdot)$ summarize flows just before and upon entering i. Because these statistics are computed on contiguous subsequences (windows or maximal in-state stretches), they reflect localized behavior; normalizing by totals or by sequence length allows fair comparisons across states with different support. Together, the in-state and boundary summaries provide concise, interpretable fingerprints of each state and of the conditions under which transitions occur.

5 Case Study

To demonstrate the practical utility of our framework, we apply it to a real-world object-centric event log from inventory management at a major European pet retailer[3].

Inventory management treats each material as a long-lived object whose lifecycle consists of recurring inbound, outbound, and planning activities, such as *goods receipt*, *goods issue*, *create purchase suggestion*, and *create sales order item*. On this backdrop, our starting point is the stock-based triad of *understock*, *normal*, and *overstock*, obtained from mathematical models that determine whether the on-hand position lies below, within, or above a target region. A natural managerial hypothesis is that these states drive day-to-day operations—for instance, avoiding receipts when in overstock and expediting them when in understock.

However, prior analyses on the same inventory object-centric event log[4] show that this coupling is weak in practice: receipts and issues regularly occur across all three stock states and do not consistently follow the expected policy [2]. This leads to the central question of the case study: Can richer, data-driven execution states, learned from short-range activity patterns and context, rather than from stock attributes alone, provide a more faithful explanation of how inventory is actually managed?

[3] Key traits: observation window *December 2022–April 2024*; *9,648 materials* tracked.

[4] State time shares *38.5% Normal*, *37.2% Overstock*, *24.3% Understock*. Activity scale: *Goods Issue (Overstock)* (= 24,289,441), *Goods Issue (Normal)* (= 8,428,451), *Goods Issue (Understock)* (= 210,264), *Create Sales Order Item (Overstock)* (= 1,684,667), *Create Sales Order Item (Normal)* (= 680,844), *Create Purchase Suggestion Item (Overstock)* (= 198,908).

State	Stock	Entering (boundary sequences)	Exiting (boundary sequences)	In-state (maximal runs)
(0,0)	O	Entry marked by long *Goods Issue* chains; frequent *Create Sales Order Item* bursts; visible *Purchase Suggestion→Purchase Order* cycles; few receipts.	Still dominated by *Goods Issue*; more *Create Sales Order Item* runs than on entry; occasional planning links before moving away.	Overwhelming *Goods Issue→Goods Issue*; sparse receipts; small but steady planning (*Purchase Suggestion→Purchase Order*).
(0,1)	O	Entry via *Goods Issue→Goods Issue*, with supportive *Goods Receipt→Goods Issue* and *Purchase Order→Goods Issue*; minimal planning.	Transitions occur while *Goods Issue* chains persist; very low activity diversity near exits.	Nearly pure *Goods Issue* self-loops; "steady outbound execution" regime.
(0,2)	O	Entry mixes outbound and planning: strong *Goods Issue→Purchase Suggestion*, *Purchase Suggestion→Purchase Order*, and *Goods Receipt→Goods Issue*.	Before leaving, *Create Sales Order Item→Create Sales Order Item* becomes prominent; receipts small but present.	Blend of *Goods Issue* with stable planning loops and sales-order runs; "replenishment-aware outbound."
(1,0)	N	Entry with long *Goods Issue* chains plus *Goods Receipt→Goods Issue*; moderate planning and sales-order loops.	Exit keeps *Goods Issue* high; more sales-order starts than on entry; modest planning persists.	Mixed regime: *Goods Issue* strong, but sales-order and planning edges are consistently visible.
(1,1)	O	Entry dominated by *Goods Issue→Goods Issue*; symmetric hand-offs *Goods Receipt↔Goods Issue*; *Create Sales Order Item→Goods Issue* strong.	Exits happen under continued *Goods Issue* activity; other flows negligible.	Canonical "pure issuing" state: almost exclusively *Goods Issue→Goods Issue*.
(1,2)	O	Balanced entry: *Create Sales Order Item↔Goods Issue* plus planning *Purchase Suggestion→Purchase Order* and *Purchase Order→Goods Issue*.	Exits follow long *Goods Issue* runs; limited diversification at the boundary.	*Goods Issue*-centric but with clear demand/planning couplings; "supply–demand balancing."
(2,0)	U	Entry characterized by intense *Create Sales Order Item→Create Sales Order Item* (order build-up), with *Goods Issue* following; light receipts.	Exits keep sales-order sequences active; planning edges (*Purchase Suggestion→Purchase Order*) remain visible.	Order-entry/backlog processing: sales-order runs dominate; *Goods Issue* and planning present but secondary.
(2,1)	O	High-activity hub on entry: *Goods Issue* heavy, strong *Create Sales Order Item↔Goods Issue*, robust *Purchase Suggestion→Purchase Order*, and many receipts.	Exiting remains diverse (sales-order sequences, receipts, planning); transitions look like redistribution rather than mode change.	Busy, mixed pipeline: outbound, inbound, and planning all active in comparable magnitudes.
(2,2)	N	Hybrid entry: *Goods Issue* strong with *Goods Issue→Purchase Suggestion*, *Purchase Suggestion→Purchase Order*, and moderate receipts; *Create Sales Order Item↔Goods Issue* visible.	Exits proceed under similar hybrid flows; no single driver dominates.	Hybrid operations: *Goods Issue* plus meaningful sales-order and planning loops; moderate receipts sustained.

Table 1: Nine SOM states (3×3 grid). Stock level figures show how each cell aligns with stock-based states. Text summarizes boundary conditions (enter/exit) and typical in-state behavior.

Our case study examines the stability and interpretability of states discovered by our unsupervised pipeline. The implementation of the analysis is available as an open-source repository[5]. It comprises a pipeline of Python scripts that process object-centric event logs to extract numerical vectors from event sequences, apply sliding windows followed by principal component analysis to obtain compact representations, train a self-organizing map to infer execution states, compute quality metrics to assess alignment with ground-truth labels, and produce directly-follows graphs along with activity summaries to characterize in-state behaviors and boundary conditions.

Parameter selection. All hyperparameters were optimized on *stock-aware* embeddings (activities + stock). For the sliding window, we swept $w \in [2, 20]$ and, at a fixed probe dimension, minimized the weighted mean PCA reconstruction error across traces; the elbow of the $E(w)$ curve occurs at $w^\star = 8$ (e.g., $E(8) \approx 1.15 \times 10^{-3}$), which balances local context and fidelity. With $w^\star$ fixed, we selected the PCA dimension by the elbow of $E(k)$ versus components: the steep error drop flattens around $k = 10$, yielding $d^\star = 10$. For the SOM grid, we evaluated square sizes and maximized $J(M) = \alpha\text{Adjacency} + (1-\alpha)\text{Entropy}_{\text{norm}}$ with $\alpha = 0.7$, using quantization error as a tie-breaker. The best grid was 3×3 in

[5] `https://github.com/fit-alessandro-berti/states-detection-understanding`

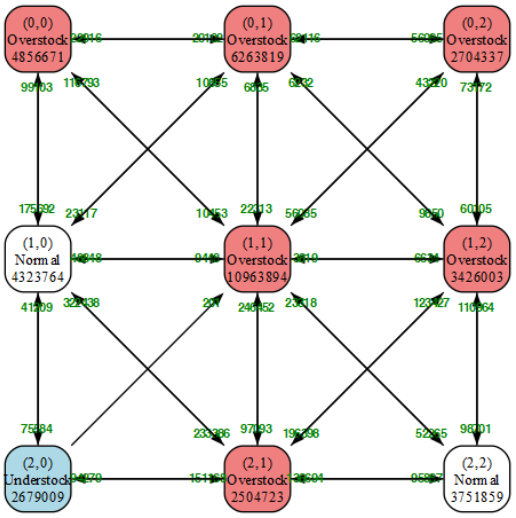

Fig. 2: State-transition view of the 3×3 SOM in Table 1. Each node corresponds to a SOM cell and is annotated by the dominant stock state and the number of assigned windows. The green label at the tail of each edge reports the frequency of transitions from the source node to the target node.

the 70% of the materials (vs. 21% for 4×4 and 9% for 5×5), consistently offering high temporal locality and good neuron utilization without over-fragmentation. We therefore fixed $M = 3$ globally (nine states), and used $(w^\star, d^\star, M) = (8, 10, 3)$ for all subsequent analyses.

Why activities + stock? We compare *activities only* vs. *activities + stock* on a 3×3 SOM, evaluating micro/balanced accuracy for stock-state alignment, Adjacency (neighbor-preservation), and η^2 (stock variance explained). Activities alone yield coherent clusters (Adjacency=0.91) but poor alignment: micro accuracy 0.66, balanced 0.33 (chance), $\eta^2 = 0.03$, with scattered Understock. Adding stock preserves smoothness (Adjacency=0.92) and boosts alignment: micro 0.74, balanced 0.44, $\eta^2 = 0.43$, concentrating Understock distinctly. Thus, stock enhances state interpretability without losing coherence, confirming its value as a contextual signal.

Identified states. Table 1 shows that the SOM discovers *distinctive, interpretable regimes*. Two cells, **(0,1)** and **(1,1)**, are "pure issuing" states dominated by *Goods Issue→Goods Issue*, offering a clean baseline. Others mix outbound with planning and/or inbound: **(0,2)** and **(1,2)** consistently exhibit *Purchase Suggestion→Purchase Order* alongside sales-order couplings; **(2,1)** is a high-volume hub where outbound, inbound, and planning co-exist at scale. The cell **(2,0)** stands out for *Create Sales Order Item* runs (order build-up), which also explains why it concentrates the largest share of *Understock* among all cells. Finally, Normal-majority cells **(1,0)** and **(2,2)** differ in kind; **(1,0)** is more throughput-oriented, while **(2,2)** is a hybrid with persistent planning. Overall, these patterns are interpretable, while also revealing finer structure (e.g., order build-up vs.

State	Favorable (N)?	Issues in current state						Stay suggestions (if favorable)					Leave suggestions (if not favorable)							
		Overstock (accumulated)	Understock or backlog	GI monoculture	Cancelled receipts	GI lag versus SO	Weak PS→PO	Maintain PO cadence and lot size	Cross-dock or fast receiving	Balance slots or storage	Drift alerts (GI-only)	Expedite POs or higher reorder (U→N)	Reserve stock or throttle GI (U→N)	Substitutes or transfer-in (U→N)	Reduce receipts or lot size (O→N)	Promote demand or transfer-out (O→N)	Suppress new suggestions (GI-only O)		
(0,0)		×			×										×	×				
(0,1)		×		×			×								×	×	×			
(0,2)		×			×										×	×				
(1,0)	×				×			×	×	×	×									
(1,1)		×		×			×								×	×	×			
(1,2)		×			×		×								×	×				
(2,0)			×		×	×						×	×	×						
(2,1)		×			×										×	×				
(2,2)	×				×			×	×	×	×									

Table 2: Policy matrix per SOM cell (3×3 grid). A cross (×) means the issue is observed for that state or the recommendation applies.

replenishment-aware outbound) that stock alone does not distinguish. Figure 2 visualizes the same 3×3 SOM as Table 1, with nodes indicating the dominant stock state in each cell and edge tails annotated with the observed frequencies of transitions from the source to the respective target cell.

Operational Suggestions Based on our analysis, we recommend using the policy matrix in Table 2 as a practical guide: it highlights key issues across the nine inventory states (in a 3x3 grid) and suggests targeted actions to sustain favorable normal-stock states ("N") or exit unfavorable overstock ("O") or understock ("U") ones. Checkmarks pinpoint specific problems, such as overstock buildup, backlogs, repetitive goods issues (GI), canceled receipts, delays between sales orders (SO) and GI, or poor conversion from purchase suggestions (PS) to purchase orders (PO), enabling quick diagnostics. For maintaining normal states like **(1,0)** and **(2,2)**, we suggest prioritizing consistent PO scheduling and sizing, streamlining receiving with faster processes, optimizing warehouse space allocation, and implementing alerts for early detection of GI-heavy drifts or receipt disruptions. To resolve understock in **(2,0)**, often driven by SO accumulation and GI slowdowns, we advise expediting POs or increasing reorder points, reserving available stock while curbing GI, and exploring substitutes or inter-location transfers. When tackling overstock, we recommend customizing interventions by state: for GI-centric traps in **(0,1)** and **(1,1)**, reduce lot sizes, stimulate demand via promotions or outbound transfers, and halt new PS; for planning-intensive states like **(0,0)**, **(0,2)**, **(1,2)**, and **(2,1)**, slash receipt volumes and lots, amplify demand or transfers, and refine planning precision. Universally, we propose bolstering receiving through standardized ASNs, stricter vendor accountability, and fast-tracking critical deliveries to cut cancellations. To activate these, we suggest monitoring the latest events against matrix indicators (for instance, escalating SOs warrant PO acceleration, whereas GI sequences prompt receipt reductions).

6 Conclusion and Future Work

In this paper, we introduced an unsupervised framework to discover execution states and boundary conditions from object-centric event logs, providing in-

terpretable summaries of object evolution via recurrent patterns. Using sliding windows for local contexts, PCA for compression, and SOM for projection, our approach reveals data-driven states aligned with operations and novel subregimes. The inventory case study showed that activities alone produce coherent but poorly aligned clusters with stock states, while adding stock levels boosts alignment and interpretability. The nine-state map uncovered regimes like *pure issuing*, *replenishment-aware outbound*, and *order build-up*, with distinct patterns and transitions enabling interventions such as early warnings and rebalancing. Future work includes enriching representations with temporal features, positional encodings, and object attributes; exploring embeddings like contrastive learning or autoencoders; and integrating with applications like real-time warnings, counterfactuals, and online learning to support process monitoring and optimization.

References

1. Berti, A., Kretzschmann, D., Aalst, W.M.P.V.D.: Segmentation for optimizing long-lifecycle processes in object-centric process mining. TechRxiv (Jun 2025), preprint
2. Berti, A., Kretzschmann, D., Aalst, W.M.P.V.D.: State-aware object-centric process mining: Enhancing ocel 2.0 with explicit state transitions. TechRxiv (Jun 2025), preprint
3. Camargo, M., Dumas, M., Rojas, O.G.: Learning accurate LSTM models of business processes. In: BPM. Lecture Notes in Computer Science, vol. 11675, pp. 286–302. Springer (2019)
4. van Eck, M.L., Sidorova, N., van der Aalst, W.M.P.: Guided interaction exploration and performance analysis in artifact-centric process models. Bus. Inf. Syst. Eng. **61**(6), 649–663 (2019)
5. Fahland, D.: Artifact-centric process mining. In: Encyclopedia of Big Data Technologies. Springer (2019)
6. Junior, S.B., Ceravolo, P., Damiani, E., Tavares, G.M.: Evaluating trace encoding methods in process mining. In: DataMod@CIKM. Lecture Notes in Computer Science, vol. 12611, pp. 174–189. Springer (2020)
7. Koninck, P.D., vanden Broucke, S., Weerdt, J.D.: act2vec, trace2vec, log2vec, and model2vec: Representation learning for business processes. In: BPM. Lecture Notes in Computer Science, vol. 11080, pp. 305–321. Springer (2018)
8. Koren, I., Adams, J.N., Berti, A., van der Aalst, W.M.P.: OCEL 2.0 resources - www.ocel-standard.org. In: ICPM Doctoral Consortium / Demo. CEUR Workshop Proceedings, vol. 3648. CEUR-WS.org (2023)
9. Li, G., de Carvalho, R.M.: Dealing with artifact-centric systems: a process mining approach. EMISA Forum **38**(1), 61–65 (2018)
10. Pellicani, A., Ceci, M.: Positional trace encoding for next activity prediction in event logs. Knowl. Based Syst. **319**, 113544 (2025)
11. Tavares, G.M., Oyamada, R.S., Barbon, S., Ceravolo, P.: Trace encoding in process mining: A survey and benchmarking. Eng. Appl. Artif. Intell. **126**(Part D), 107028 (2023)
12. Teinemaa, I., Dumas, M., Rosa, M.L., Maggi, F.M.: Outcome-oriented predictive process monitoring: Review and benchmark. ACM Trans. Knowl. Discov. Data **13**(2), 17:1–17:57 (2019)
13. Ye, L., Keogh, E.J.: Time series shapelets: a new primitive for data mining. In: KDD. pp. 947–956. ACM (2009)