Visualizing Object-Centric Petri Nets

Tobias Brachmann[1], István Koren[2,3][0000−0003−1350−6732], Lukas Liss[2][0000−0002−4719−7993], and Wil M. P. van der Aalst[2][0000−0002−0955−6940]

[1] RWTH Aachen University, Aachen, Germany `tobias.brachmann@rwth-aachen.de`
[2] Chair of Process and Data Science, RWTH Aachen University, Aachen, Germany
[3] Department of Data Science and Engineering, ELTE University, Budapest, Hungary

Abstract. Object-centric process mining (OCPM) is gaining traction in both academia and industry due to its ability to model real-world processes more accurately than traditional case-centric approaches. By considering multiple interacting objects, object-centric Petri nets (OCPNs) offer a richer process representation, but this also introduces unique challenges for visualization. The presence of multiple object types, variable arcs, and complex interactions complicates the creation of clear and interpretable layouts. In this paper, we address these challenges by presenting a dedicated layout algorithm tailored to the structural characteristics of OCPNs. Inspired by the Sugiyama framework, the algorithm balances aesthetic and functional criteria, guided by a set of domain-specific quality metrics. We implemented our approach in an open-source web-based tool, OCPN Visualizer, and a reusable JavaScript library for integration into third-party applications. A user study confirms the practical relevance of our approach and highlights its effectiveness in improving the interpretability of object-centric process visualizations.

Keywords: Object-Centric Petri Net · Process Visualization · Layout Algorithm · Sugiyama.

1 Introduction

The rise of object-centric process mining (OCPM) reflects a shift in how organizations analyze and understand processes. Traditional process mining approaches typically rely on a case-centric perspective, modeling a process in terms of a single entity, such as an order. Real-world processes involve interactions among entities (e.g., orders, customers, products) that case-centric models cannot adequately capture. This disconnect has motivated the development of OCPM, which provides a more realistic and comprehensive representation of complex business processes [33].

Object-centric Petri nets, an extension of classical Petri nets with typed places representing different object types, have emerged as a key modeling formalism for OCPM. OCPNs support richer modeling but pose new visualization challenges. The structural complexity arising from multiple object types, variable arcs, and inter-object dependencies makes it difficult to generate intuitive

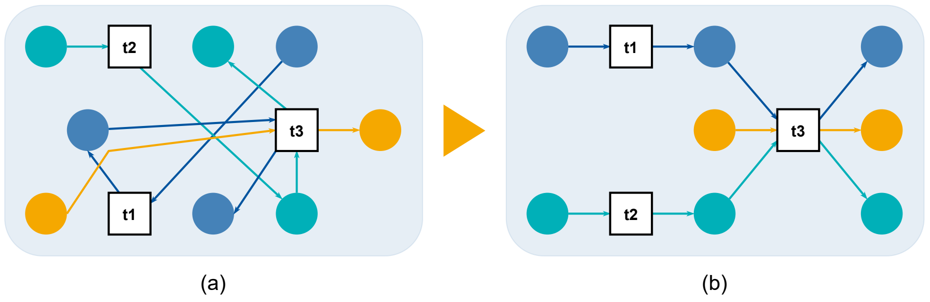

Fig. 1: Two visualizations of the same OCPN, where (a) represents a poorly organized layout and (b) illustrates an improved, more structured layout

layouts. Traditional visualization approaches for Petri nets are often insufficient, as they do not account for the semantics introduced by object-centricity.

Larger OCPNs, especially those mined from real data, make clear visualization increasingly difficult. Poorly structured layouts can lead to visual clutter and obscure the underlying relationships between entities, complicating interpretation. Figure 1 illustrates the impact of layout quality: while the left-hand side displays a disorganized structure with overlapping edges and weak object separation (objects distinguished by colors), the right-hand side demonstrates how a layout tailored to object-centric properties improves readability and interpretability by grouping related elements and emphasizing structural flow.

Despite the growing interest in OCPM, existing tools offer limited support for visualizing OCPNs. We address this gap by proposing a layout algorithm tailored to OCPNs, extending the Sugiyama framework [29] to incorporate object-type grouping, flow direction, and edge clarity. Our contributions are threefold: (1) quality metrics that combine graph aesthetics with object-centric criteria, (2) the layout algorithm itself, and (3) an open-source implementation available both as a web-based visualizer and a JavaScript library. We evaluate our approach through runtime analysis and a user study.

The remainder of this paper is structured as follows. Section 2 reviews related work in process mining and graph drawing. Section 3 introduces key concepts and definitions. Section 5 presents our layout algorithm and quality metrics. Section 6 describes the implementation. Section 7 reports the evaluation results, including a comparison with existing tools and a user study. Finally, Section 8 concludes the paper and outlines directions for future work.

2 Related Work

This section reviews prior work on process mining techniques and graph drawing methods relevant to the visualization of object-centric process models.

2.1 Object-Centric Process Mining

Traditional process mining focuses on analyzing event logs under a single case notion [32]. However, many real-life processes involve interactions between multiple entities, which led to the development of *object-centric process mining*. This paradigm shift addresses the limitations of single-case models by enabling the discovery of OCPNs from event logs with multiple interconnected case notions [34]. The method was implemented in the *pm4py* library [8], which continues to evolve as a key tool in the field.

Research has addressed layout challenges specifically in the context of business process visualization. Bernstein and Soffer [7] investigated which layout features users perceive as meaningful, providing empirical foundations for layout design. Another work proposes a stable layout algorithm based on the Sugiyama framework, tailored to process graphs and aimed at preserving the user's mental map during interactive filtering [23]. Gschwind et al. [21] introduced a linear-time layout algorithm for business process models that emphasizes readability through structural simplification. Sonke et al. [27] focused on optimal algorithms for compact linear layouts, which are particularly relevant for minimizing space while retaining clarity in flow-based visualizations.

Despite these advances, tools for visualizing OCPNs remain limited. *Celonis* is a commercial tool supporting object-centric visualizations. $OC\pi$ [1] offers sequence-based visualizations and filtering capabilities for OCPNs, relying on the *Graphviz* library [18]. Meanwhile, *pm4py* supports object-centric discovery and visualization within a Python environment. General-purpose graph layout libraries (e.g., Graphviz) do not consider domain semantics like object grouping, making their output hard to interpret in OCPN contexts. This motivates the need for specialized visualization approaches.

2.2 Graph Drawing

Graph drawing aims to represent relational data visually using algorithms that produce readable, structured layouts [13,5]. One prominent approach is the already mentioned Sugiyama method [29], a framework for drawing directed graphs by organizing vertices into hierarchical layers. The method is composed of multiple steps (i.a., cycle breaking, layering, vertex ordering), and has inspired extensive research into optimizing each step (e.g., [30,16].

Force-directed algorithms such as Fruchterman-Reingold [19] use the physical analogies of vertices repelling each other and edges acting as springs to achieve visually balanced layouts. These methods are intuitive and flexible but computationally intensive for larger graphs. Magnetic-field-based techniques [28] introduce directional forces to emphasize hierarchies or flows, especially useful in layered or directed graphs. Hybrid approaches aim to combine the strengths of these techniques [12].

Aesthetic principles play a vital role in enhancing graph readability. Studies emphasize minimizing edge crossings, maintaining consistent edge lengths, and maximizing symmetry [25]. These criteria improve user comprehension and reduce cognitive load when interpreting complex graphs [6].

3 Preliminaries

In this section, we provide an overview of key concepts in process mining and graph drawing.

3.1 Process Mining

OCPM extends classical process mining by modeling interactions between multiple object types, such as orders, products, and customers. While traditional Petri nets rely on a single-case notion, OCPNs support multiple interacting lifecycles. For example, the transition "ship product" may involve both an order and a product. This complexity complicates visualization and motivates our graph-based layout approach. We assume universes of activity names ($\mathbb{U}_{act}$) and object types ($\mathbb{U}_{ot}$) as given.

Definition 1 (Labeled Petri Net [34]). *A labeled Petri net is a tuple $N = (P, T, F, l)$ where:*

- *P: Set of places*
- *T: Set of transitions, with $P \cap T = \varnothing$*
- *$F \subseteq (P \times T) \cup (T \times P)$: Flow relation between places and transitions*
- *$l \in T \nrightarrow \mathbb{U}_{act}$: Labeling function that maps transitions to activity names*

While the underlying graph structure of Petri nets is bipartite, we formally define relevant graph-theoretic concepts in Section 3.2.

Definition 2 (Object-Centric Petri Net [34]). *An object-centric Petri net is a tuple $ON = (N, pt, F_{var})$, where:*

- *$N = (P, T, F, l)$: Labeled Petri net*
- *$pt \in P \rightarrow \mathbb{U}_{ot}$: Mapping function that assigns object types to places*
- *$F_{var} \subseteq F$: Subset of variable arcs*

In an OCPN, places represent typed objects and transitions represent activities involving them. Arcs define token flow (i.e., object references) between places and transitions. *Variable arcs*, rendered as double lines, indicate that the number of involved objects is not fixed (e.g., "assemble order" may consume multiple products).

3.2 Graphs

To support our layout algorithm, we now formalize the underlying graph structures derived from OCPNs.

Definition 3 (Directed Graph [3]). *A directed graph $G = (V, E)$ consists of a set of vertices V and a set of ordered pairs $E \subseteq V \times V$ called edges.*

For an edge (u, v), often denoted $u \to v$, u is the *head*, v the *tail*. Vertices u, v are *adjacent* if $(u, v) \in E$, and each is a *neighbor* of the other. A vertex is *incident* to an edge if it is either head or tail. A sequence $\langle u_1, \dots, u_k \rangle$ is a *path* if $(u_i, u_{i+1}) \in E$ for all $i < k$; it is a *cycle* if also $(u_k, u_1) \in E$. A digraph is *acyclic* if it contains no cycles.

Definition 4 (DAG [31]). *A DAG is a directed acyclic graph.*

The *outdegree* and *indegree* of a vertex u are denoted $d^+_G(u)$ and $d^-_G(u)$, respectively. A vertex with zero outdegree is a *sink*; with zero indegree, a *source* [14]. A graph $G = (V, E)$ is *bipartite* if V can be partitioned into $V_1 \cup V_2$ such that no two vertices within the same set are adjacent [2]. OCPNs can be modeled as bipartite graphs with places P and transitions T as vertex sets, and flow relation F as edges.

A key step in layered layout is converting cyclic graphs into DAGs. Since OCPNs derived from real-world data may contain cycles, these must be removed as a preprocessing step for layout computation.

Definition 5 (Feedback Arc Set (FAS) [11]). *A feedback arc set in a digraph $G = (V, E)$ is a subset $F \subseteq E$ such that $(V, E \setminus F)$ is acyclic.*

Finding a minimal FAS is NP-hard [9,26] and underpins the preprocessing for layer assignment.

Layer Assignment Given a DAG $G = (V, E)$, a *layering* is a partition $\mathcal{L} = \{L_0, \dots, L_h\}$ of V, where $(u, v) \in E$ implies $u \in L_j, v \in L_i, i < j$. We define:

Definition 6 (Layer Assignment Problem [17,22]). *Given a DAG, find a layering $\mathcal{L}$ such that all edges point from lower to higher layers.*

The *rank* of a vertex u is the index i such that $u \in L_i$, denoted $\text{rank}(u, \mathcal{L})$. The *span* of edge (u, v) is $\text{rank}(u) - \text{rank}(v) \geq 1$. If span $= 1$, the edge is *tight*; otherwise, *long*. We convert long edges to tight ones by inserting *dummy vertices* at intermediate layers.

The *height* of a layering is $h + 1$, its *width* is the maximum layer size, and its *area* is the product of both [22,14,31].

Barycenter Heuristic Vertex ordering within layers plays a crucial role in reducing edge crossings and preserving structural clarity. A common technique for this is the barycenter heuristic, which reorders vertices based on the average positions of their neighbors in adjacent layers.

Definition 7 (Barycenter [14,29]). *For a vertex u in a layer, let $N(u)$ denote the adjacent vertices of u in the layer above (or below). The barycenter $b(u)$ is defined as:*

$$b(u) = \frac{1}{\mid N(u) \mid} \sum_{v \in N(u)} pos(v)$$

where $pos(v)$ is the position of the vertex v in the adjacent layer.

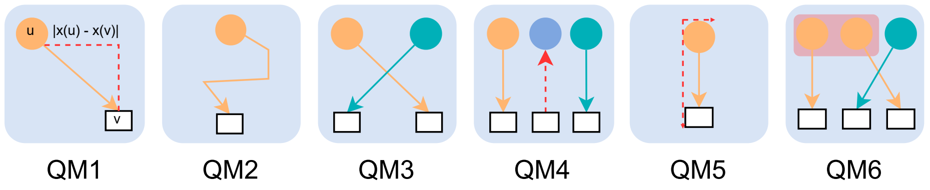

Fig. 2: The six quality metrics used in the layout algorithm

Having established the key graph-theoretic foundations, the next section defines the quality metrics that guide and evaluate our layout algorithm. These metrics combine classical aesthetic criteria from graph drawing with domain-specific considerations relevant to object-centric process models.

4 Quality Metrics

Our layout algorithm is driven by the following six quality metrics, which can be categorized into two groups. The first five metrics (**QM1–QM5**) are derived from principles in the graph drawing literature [25] and aim to promote general readability. The final metric (**QM6**) is specific to the object-centric nature of OCPNs, capturing structural aspects that are unique to this modeling formalism.

QM1 *Total edge length.* Measures the sum of edge lengths across the layout. While longer edges are not inherently problematic, reducing total length often leads to more compact visualizations and helps avoid excessive whitespace.

QM2 *Edge bends.* Counts the number of bends along edges. Layouts with fewer bends tend to produce simpler and more coherent visual connections.

QM3 *Edge crossings.* Counts the number of edge intersections. Minimizing crossings is a well-known heuristic to support visual clarity and reduce ambiguity in complex graphs.

QM4 *Flow consistency.* Identifies edges that run counter to the dominant flow direction (e.g., from bottom to top in a top-down layout). While not all reversed edges hinder interpretation, a consistent flow direction can support the mental map of users and improve navigation through the process structure.

QM5 *Aspect ratio balance.* Assesses the ratio between the width and height of the layout. Extremely wide or tall layouts can lead to inefficient use of space and fragmented views when visualized in scrolling interfaces.

QM6 *Object-type grouping.* Measures the spatial variance of places with the same object type across the layout. A lower variance indicates stronger grouping, which can help users identify object-level structure and interpret inter-object dependencies.

Figure 2 illustrates the six quality metrics using synthetic examples. These metrics are used throughout the layouting pipeline and also inform the evaluation described in Section 7.

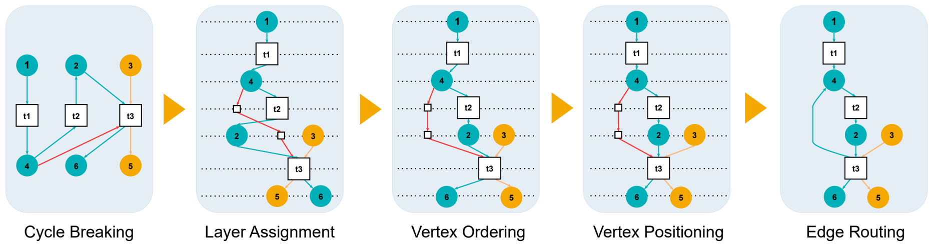

Fig. 3: Substeps of our algorithm based on the Sugiyama framework

5 Layout Algorithm

In this section, we present the design of our layout algorithm, based on the Sugiyama framework [29]. The general pipeline of our algorithm consists of five steps, shown in Figure 3. In the first step, **Cycle Breaking**, the possibly cyclic graph is transformed into an acyclic graph by reversing the minimum number of edges whose reversal makes the graph acyclic. This is necessary for the following step, **Layer Assignment**, where each vertex of the graph is assigned a rank so that for each edge of the graph, the tail has a lower rank than the head. For edges where the head and tail vertices are not on adjacent layers, so-called dummy vertices are inserted in every layer between the layers of the head and tail. After each vertex has been assigned a rank, the **Vertex Ordering** step minimizes the number of edge crossings and groups related vertices together by determining a relative order within the layers. Now, every vertex has a relative position determined by its rank and index in its layer. The next step, **Vertex Positioning**, computes the actual X and Y coordinates for the vertices based on their rank and index in their layer. In the last step, **Edge Routing**, the edges between adjacent vertices are drawn with their original direction. For edges where dummy vertices had been introduced in the Layer Assignment step, the positions of the dummy vertices are used as path points to draw the edge.

Input. The inputs of our layout algorithm are the OCPN $ON = (N, pt, F_{var})$ where $N = (P, T, F, l)$, and the user settings *Config*, including various parameters and preferences that influence the layout, such as flow direction, included object types, distances, and vertex sizes.

Cycle Breaking. To build a layered visualization using the Sugiyama framework, the input graph must be acyclic. However, the input graph, $G = (V, E)$, derived from the OCPN may contain cycles. To address this issue, our algorithm includes a cycle-breaking step, which reverses a set of edges to make the graph acyclic. Following QM4 (flow direction), we aim to find the smallest possible set that satisfies the condition of making the graph acyclic upon reversal. In our implementation, we adapt and modify an algorithm, known as Greedy Cycle Removal, proposed by Eades et al. [15]. This algorithm provides a good, though not necessarily optimal, solution to the minimum FAS problem and runs in linear time. The greedy FAS algorithm computes a linear ordering of vertices and

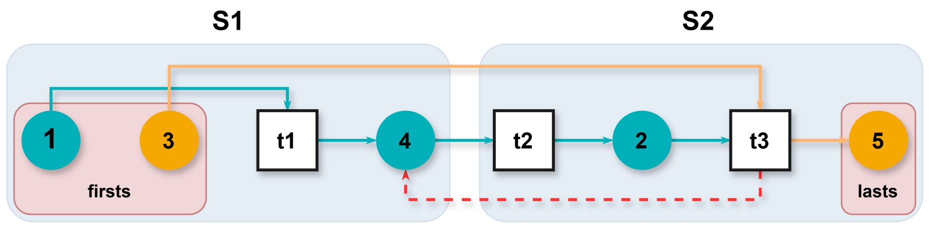

Fig. 4: The modified Greedy Cycle Removal algorithm. For this example, the user has selected $firsts = [1, 3]$ and $lasts = [5]$. The edge $e = (t3, 4)$ will be reversed because $ind_s(t3) = 6 > 3 = ind_s(4)$. Note that reversing the edge $e = (t3, 4)$ results in a DAG.

solves the FAS problem by taking the set of edges with direction against the ordering as a solution.

We modified the algorithm by incorporating two sets of vertices: *firsts* and *lasts*. The vertices in *firsts* are placed at the beginning of the ordering and the vertices in *lasts* are placed at the end. This modification enables users to control the assignment of vertices to the horizontal layers, assigned during the Layer Assignment.

Layer Assignment. The objective for the layer assignment step is to obtain a proper layered DAG $G^{'}$ by solving the layer assignment problem for the DAG G^*. That means we assign each vertex of the DAG to a specific layer, ensuring that every edge flows from a higher layer (head) to a lower layer (tail) while inserting dummy vertices for edges spanning multiple layers. This process provides a clear directional flow for all edges, addressing QM4. Furthermore, a compact layering is crucial, as it directly impacts the quality of the layout. Fewer layers result in fewer dummy vertices, which can shorten edge lengths (QM1), reduce edge bends (QM2), and positively influence the number of edge crossings (QM3). Additionally, a good layer assignment improves the proportion between the width and height of the visualization (QM5).

To achieve an optimal layering, $\mathcal{L} = \{L_0, L_1, \ldots, L_h\}$, of G^*, we employ an ILP approach, first introduced by Gansner and Emden [20]. This method minimizes h, the number of layers required, and thus the amount of dummy vertices, while respecting the graph's structural constraints. The ILP formulation is as follows:

$$\text{minimize} \sum_{e \in E^*} span(e, \mathcal{L}) \qquad \text{(Objective Function)}$$

Subject to:

$$rank(u, \mathcal{L}) - rank(v, \mathcal{L}) \geq 1, \forall (u, v) \in E^* \qquad \text{(Edge Constraint)}$$
$$rank(u, \mathcal{L}) \geq 0, \forall u \in V \qquad \text{(Positive Constraint)}$$
$$rank(u, \mathcal{L}) \in \mathbb{N}, \forall u \in V \qquad \text{(Integer Constraint)}$$

To solve the above ILP formulation, we utilize a library that implements a simplex method [24] that returns a layering $\mathcal{L}$ of G^*. That is, for every vertex u we obtain a $rank(u, \mathcal{L})$ so that the objective function is minimized.

Since a proper layering is essential for the Vertex Ordering step, we insert dummy vertices for edges that are not tight. This turns the potentially not proper layering $\mathcal{L}$ into a proper one. Formally, let $e = (u, v)$ be an edge with $rank(u, \mathcal{L}) = j$ and $rank(v, \mathcal{L}) = i$ and $span(e, \mathcal{L}) = j - i > 1$. Then we add dummy vertices $d_e^{i+1},\ d_e^{i+2}, \ldots, d_e^{j-1}$ to the layers $L_{i+1}, L_{i+2}, \ldots, L_{j-1}$ and replace edge e by the path $(u,\ d_e^{j-1}, \ldots,\ d_e^{i+1},\ v)$ [22].

Vertex Ordering. The Vertex Ordering step aims to minimize edge crossings and promote the clustering of places with the same object type, addressing layout quality metrics such as edge crossings (QM3), total edge length (QM1), edge bends (QM2), and object type grouping (QM6). The output is a reordered layering $\mathcal{L}$ of the graph $G^{'}$ with the lowest observed layout score.

Layer-by-Layer Sweep Our algorithm applies a layer-by-layer sweep based on a modified barycenter heuristic, alternating between two directions: a *downward sweep*, where layers L_{h-1} to L_0 are reordered based on the positions of vertices in the layer below (L_{i+1}); and an *upward sweep*, where layers L_1 to L_h are reordered based on the positions of vertices in the layer above (L_{i-1}).

An optional presorting step may precede the sweeps, where object types are ordered according to a user-defined mapping and places are sorted accordingly to promote initial clustering.

Barycenter Computation for Places Each place vertex $p \in L_i$ receives a barycenter value $b_p(p)$ combining neighbor-alignment and object type grouping. The value is computed as:

$$b_p(p) = (1 - \alpha) \cdot b(p) + \alpha \cdot b_{\text{object}}(p),$$

where $\alpha \in [0, 1]$ is a user-defined weight, $b(p)$ is the average position of p's neighbors in $L_{i\pm1}$ (depending on sweep direction), and

$$b_{\text{object}}(p) = \frac{1}{\mid N_{\text{object}}(p) \mid} \sum_{u \in N_{\text{object}}(p)} pos(u),$$

with $N_{\text{object}}(p)$ being the set of places with the same object type as p in a user-defined range of layers above or below the current layer.

Barycenter Computation for Other Vertex Types For *transitions*, the barycenter $b(t)$ is computed using only adjacent vertices in the neighboring layer; clustering is not applied. For *dummy vertices*, which connect to exactly one neighbor in the adjacent layer, the barycenter equals the position of that neighbor.

Reordering Based on Barycenters Once barycenter values are computed for all vertices in layer L_i, the layer is reordered from left to right by sorting vertices in ascending order of these values. In case of ties, the previous order is preserved to maintain stability.

Termination Conditions The sweep process alternates downward and upward passes across the layered graph. After each *complete sweep* (i.e., one downward and one upward pass), the current vertex layering $\mathcal{L}$ is evaluated using the layout score:

$$score(\mathcal{L}, G') = |\text{Edge Crossings}| + \alpha \cdot \text{Object Attraction Quality},$$

where the first term counts edge crossings and the second quantifies how well places of the same type are grouped. The process continues until either (i) no score improvement is observed after k consecutive sweeps (with k from the configuration), or (ii) the vertex ordering matches a previous sweep. The final output is the layering $\mathcal{L}$ with the lowest score, minimizing edge crossings and promoting object-type clustering.

Vertex Positioning. Given the reordered layering $\mathcal{L}$ from the vertex ordering step, the graph G', and user configurations $Config$, the goal is to assign X and Y coordinates to all vertices, while respecting the layer structure.

Our algorithm supports both top-down (vertical) and left-right (horizontal) layouts, aligning with the flow semantics in QM4. We describe the top-down case: X coordinates are computed based on the within-layer order, while Y coordinates are uniform across each layer. Since Y positioning is straightforward, we focus on the more complex task of X coordinate computation. In the horizontal layout, this logic is mirrored.

To compute horizontal positions while preserving edge straightness and vertex order, we adapt the heuristic by Brandes and Köpf [10], which addresses layout quality via QM1 and QM2. The algorithm consists of three phases: **Vertical Alignment**, **Horizontal Compaction**, and **Balancing**. The first two are repeated across four alignment variants (upper/lower, leftmost/rightmost medians), and the final layout is obtained by averaging the results to ensure a well-balanced visual structure.

Edge Routing. With the X and Y coordinates for all vertices in the graph $G^{'}$ determined, the next step is to finalize the layout by routing the edges. Edge routing ensures that edges are drawn clearly and do not overlap with other elements of the graph, such as places or transitions, while preserving the structure and flow of the graph. Any edges that were reversed during the cycle-breaking step are restored to their original direction during this phase.

We begin by adjusting the positions of the outer dummy vertices, ensuring that they are aligned at the top or bottom of their respective layers. For an edge e, which had been replaced by the path $(u, d_e^{j-1}, \ldots, d_e^{i+1}, v)$, the outer dummies are d_e^{j-1} and d_e^{i+1}. After adjusting the positions of the outer dummies the edges are routed to the center of the respective vertices. Finally, the outer

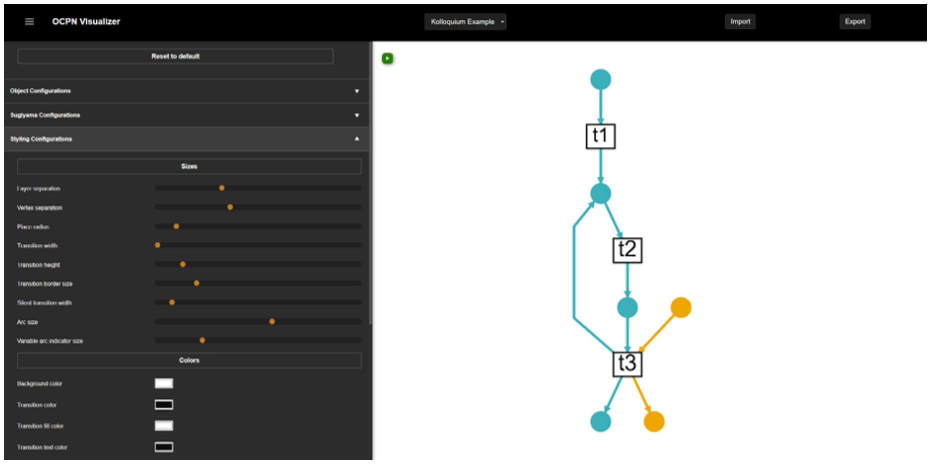

Fig. 5: GUI of the OCPN Visualizer with layout configuration options on the left (not intended to be readable here)

dummy vertices are used as path points to guide the edges. The inner dummy vertices, if any, are ignored for routing, as the horizontal alignment guarantees that long edges will follow a straight path along the X-axis. This simple approach results in a clear, well-organized layout where edges are routed to the appropriate connection points of vertices and avoid overlap.

6 Implementation

The OCPN Visualizer is a web-based tool that integrates our layout algorithm[4]. It is developed using a state-of-the-art technology stack, consisting of a React and Next.js frontend and a TypeScript backend. Our implementation utilizes several libraries: `GLPK.js` is employed to solve the layer assignment problem, and `D3.js` supports the creation of interactive and visually expressive SVG-based graphics. The core layouting logic is available as the reusable npm package `ocpn-viz`.

Figure 5 shows the GUI of the OCPN Visualizer with the configuration options on the left. To support flexible exploration, our tool includes a configuration panel that allows users to adjust visual parameters, such as node width, arc curvature, and label padding, and see the layout update in real time. This helps users adapt the visualization to specific model characteristics or preferences (e.g., focusing on arc routing or text readability). Additionally, hovering and clicking on transitions or places reveals further details, aiding the interpretation of densely connected models. While we do not yet support filtering or collapsing parts of the net, these interactive adjustments already improve navigability and comprehension, especially for large models.

[4] Source code and app: `https://github.com/rwth-pads/ocpn-visualizer`

7 Evaluation

To evaluate the performance of our approach, we used a set of OCPNs of varying sizes and structural complexities. These included both small, synthetic models and larger, more complex instances designed to reflect real-world characteristics. All benchmarks were run on a modern consumer-grade laptop using Chrome. For each OCPN, the metrics recorded included the number of places ($|P|$), transitions ($|T|$), edges ($|F|$), variable arcs ($|F_{var}|$), dummies introduced during layer assignment ($|D|$), and object types ($|OT|$). *Layout time* refers to the time required to compute the coordinates for all vertices in the OCPN, while *visualization time* is the time taken to draw the output SVG. Table 1 summarizes the performance results across OCPNs of varying complexities.

Using the profiler within Chrome DevTools, we measured the memory usage of the *OCPN Visualizer* across OCPNs of varying complexity. The memory usage grew proportionally with the size of the OCPNs, starting at 746 kB for a small OCPN with two vertices and reaching 1.5 MB for a larger OCPN with 310 vertices. This trend indicates good scalability, as the memory usage remains manageable even for more complex OCPNs.

7.1 PM4Py vs OCPN Visualizer

We compared visualizations generated by the *OCPN Visualizer* and *pm4py* [8] using three OCPNs of varying sizes to highlight their respective strengths.

For small and medium models, the *OCPN Visualizer* offers faster run times, interactive features, and clearer layouts, thanks to object-type grouping, variable arc support, and styling options. In contrast, *pm4py* produces aesthetically polished, compact layouts with curved edges and clear source/sink labeling, but lacks interactivity and customization. For larger models, *pm4py* outperforms in runtime and visual metrics such as edge length, bends, and crossings, though its layouts vary across executions. The *OCPN Visualizer*, while slower on large inputs, delivers consistent, interpretable results. Overall, *pm4py* suits large-scale visualization, while the *OCPN Visualizer* excels in usability and clarity for smaller models. Further improving its layout algorithm could extend its competitiveness to larger OCPNs.

Table 1: Run times in ms for OCPNs of increasing sizes

$\mid P \mid$	$\mid T \mid$	$\mid F \mid$	$\mid F_{var} \mid$	$\mid D \mid$	$\mid OT \mid$	Layout (ms)	Visualization (ms)
1	1	2	0	0	1	25.81	18.24
17	9	28	9	0	3	28.61	163.41
25	9	40	20	2	5	37.14	173.81
23	17	46	10	4	3	28.27	318.13
17	22	50	4	34	3	44.18	449.44
37	23	74	6	48	7	66.7	440.39
48	44	116	10	86	7	85.87	828.29
38	62	140	4	210	2	149.2	1222.7

7.2 User Study

To evaluate the user experience of our tool *OCPN Visualizer* we conducted a user study employing the System Usability Scale (SUS) [4]. The 20 participants that had varying levels of expertise were given two tasks: First, they had to follow a set of guided instructions to visualize a provided OCPN. Then, they were asked to achieve a target visualization by utilizing the available user configurations. After completing both tasks, participants submitted the SUS questionnaire, consisting of ten standardized questions, and open feedback. The SUS score averaged 78.6, indicating good usability. While most users found the tool highly usable, a few experienced moderate difficulties. Informal feedback indicated that the live configuration options and responsive layout adjustments improved the comprehensibility and usability of the tool. To further assess the practical value of our approach, we plan to conduct additional user studies using models discovered from real event logs to evaluate effectiveness in realistic process analytics scenarios, and to directly compare our visualizations with those produced by commercial tools such as Celonis.

8 Conclusion

This paper addressed the challenge of visualizing OCPNs, a task complicated by the lack of specialized tools and evaluation methods. To bridge this gap, we developed the *OCPN Visualizer*, a web-based tool featuring a dedicated layout algorithm and user configuration options, along with *ocpn-viz*, a reusable NPM package. Our work was guided by three objectives focusing on layout strategies, user interaction features, and quality metrics for visualization evaluation.

We defined six quality metrics tailored to OCPNs, combining established graph aesthetics with domain-specific needs such as object-type clustering. These informed the development of our layout algorithm. A two-part evaluation comparing our tool with *pm4py* and conducting a user study demonstrated strong performance in usability, interpretability, and scalability. The algorithm produced clear layouts for small and medium models, while offering improvements in clarity and interactivity over existing tools.

The findings validate our approach while highlighting areas for future improvement, including layout refinements, more advanced edge routing, and empirical validation of the proposed metrics. The defined quality metrics may also serve as a foundation for replicable comparisons of alternative layout algorithms in future work. Overall, this research provides a solid foundation for advancing OCPN visualization in both academic and practical contexts.

References

1. Adams, J.N., van der Aalst, W.M.P.: OCπ: Object-centric process insights. In: Bernardinello, L., Petrucci, L. (eds.) Application and Theory of Petri Nets and Concurrency, PETRI NETS 2022, Bergen, Norway, June 19-24, 2022. LNCS, vol. 13288, pp. 139–150. Springer (2022)

2. Asratian, A.S., Denley, T.M.J., Häggkvist, R.: Bipartite Graphs and Their Applications. Cambridge University Press, 1 edn. (1998). https://doi.org/10.1017/CBO9780511984068
3. Bang-Jensen, J., Gutin, G.Z.: Digraphs - Theory, Algorithms and Applications, Second Edition. Springer Monographs in Mathematics, Springer (2009)
4. Bangor, A., Kortum, P.T., Miller, J.T.: An empirical evaluation of the system usability scale. Int. J. Hum. Comput. Interact. **24**(6), 574–594 (2008). https://doi.org/10.1080/10447310802205776
5. Battista, G.D., Eades, P., Tamassia, R., Tollis, I.G.: Algorithms for drawing graphs: An annotated bibliography. Computational Geometry **4**(5), 235–282 (1994). https://doi.org/10.1016/0925-7721(94)00014-X
6. Bennett, C., Ryall, J., Spalteholz, L., Gooch, A.: The aesthetics of graph visualization. In: Cunningham, D.W., et al. (eds.) 3rd International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging, Banff, AB, Canada, June 20-22, 2007. pp. 57–64. Eurographics Association (2007). https://doi.org/10.2312/COMPAESTH/COMPAESTH07/057-064
7. Bernstein, V., Soffer, P.: How Does It Look? Exploring Meaningful Layout Features of Process Models. In: Persson, A., Stirna, J. (eds.) Advanced Information Systems Engineering Workshops, vol. 215, pp. 81–86. Springer International Publishing, Cham (2015). https://doi.org/10.1007/978-3-319-19243-7_7
8. Berti, A., van Zelst, S., Schuster, D.: PM4Py: A process mining library for Python. Software Impacts **17**, 100–556 (2023). https://doi.org/10.1016/j.simpa.2023.100556
9. Brandenburg, F.J., Hanauer, K.: Sorting heuristics for the feedback arc set problem (2011)
10. Brandes, U., Köpf, B.: Fast and simple horizontal coordinate assignment. In: Mutzel, P., Jünger, M., Leipert, S. (eds.) Graph Drawing, 9th International Symposium, GD 2001 Vienna, Austria, September 23-26, 2001, Revised Papers. LNCS, vol. 2265, pp. 31–44. Springer (2001)
11. Charbit, P., Thomassé, S., Yeo, A.: The minimum feedback arc set problem is NP-hard for tournaments. Comb. Probab. Comput. **16**(1), 1–4 (2007). https://doi.org/10.1017/S0963548306007887
12. Chimani, M., Gutwenger, C., Jünger, M., Klau, G.W., Klein, K., Mutzel, P.: The open graph drawing framework (OGDF). In: Tamassia, R. (ed.) Handbook on Graph Drawing and Visualization, pp. 543–569. Chapman and Hall/CRC (2013)
13. Eades, P., Hong, S.H.: Symmetric graph drawing. In: Tamassia, R. (ed.) Handbook on Graph Drawing and Visualization, pp. 87–113. Chapman and Hall/CRC (2013)
14. Eades, P., Lin, X.: How to draw a directed graph. In: IEEE Workshop on Visual Languages, VL 1989, Rome, Italy, October 4-6, 1989. pp. 13–17. IEEE Computer Society (1989). https://doi.org/10.1109/WVL.1989.77035
15. Eades, P., Lin, X., Smyth, W.F.: A fast and effective heuristic for the feedback arc set problem. Inf. Process. Lett. **47**(6), 319–323 (1993). https://doi.org/10.1016/0020-0190(93)90079-O
16. Eades, P., Wormald, N.C.: Edge crossings in drawings of bipartite graphs. Algorithmica **11**(4), 379–403 (1994). https://doi.org/10.1007/BF01187020
17. Eiglsperger, M., Siebenhaller, M., Kaufmann, M.: An efficient implementation of sugiyama's algorithm for layered graph drawing. J. Graph Algorithms Appl. **9**(3), 305–325 (2005). https://doi.org/10.7155/JGAA.00111
18. Ellson, J., Gansner, E., Koutsofios, L., North, S.C., Woodhull, G.: Graphviz—open source graph drawing tools. In: Graph Drawing 2001 Vienna, Austria, September 23–26, 2001 Revised Papers 9. pp. 483–484. Springer (2002)

19. Fruchterman, T.M.J., Reingold, E.M.: Graph drawing by force-directed placement. Softw. Pract. Exp. **21**(11), 1129–1164 (1991). https://doi.org/10.1002/SPE.4380211102
20. Gansner, E.R., Koutsofios, E., North, S.C., Vo, K.P.: A technique for drawing directed graphs. IEEE Trans. Software Eng. **19**(3), 214–230 (1993). https://doi.org/10.1109/32.221135
21. Gschwind, T., Pinggera, J., Zugal, S., Reijers, H.A., Weber, B.: A linear time layout algorithm for business process models. Journal of Visual Languages & Computing **25**(2), 117–132 (2014). https://doi.org/10.1016/j.jvlc.2013.11.002
22. Healy, P., Nikolov, N.S.: Hierarchical drawing algorithms. In: Tamassia, R. (ed.) Handbook on Graph Drawing and Visualization, pp. 409–453. Chapman and Hall/CRC (2013)
23. Mennens, R.J., Scheepens, R., Westenberg, M.A.: A stable graph layout algorithm for processes. Computer Graphics Forum **38**(3), 725–737 (2019). https://doi.org/10.1111/cgf.13723
24. Nelder, J.A., Mead, R.: A simplex method for function minimization. Comput. J. **7**(4), 308–313 (1965). https://doi.org/10.1093/COMJNL/7.4.308
25. Purchase, H.C.: Metrics for graph drawing aesthetics. J. Vis. Lang. Comput. **13**(5), 501–516 (2002). https://doi.org/10.1006/JVLC.2002.0232
26. Simpson, M., Srinivasan, V., Thomo, A.: Efficient computation of feedback arc set at web-scale. Proc. VLDB Endow. **10**(3), 133–144 (2016). https://doi.org/10.14778/3021924.3021930
27. Sonke, W., Verbeek, K., Meulemans, W., Verbeek, E., Speckmann, B.: Optimal Algorithms for Compact Linear Layouts. In: 2018 IEEE Pacific Visualization Symposium (PacificVis). pp. 1–10. IEEE, Kobe (2018). https://doi.org/10.1109/PacificVis.2018.00010
28. Sugiyama, K., Misue, K.: Graph drawing by the magnetic spring model. J. Vis. Lang. Comput. **6**(3), 217–231 (1995). https://doi.org/10.1006/JVLC.1995.1013
29. Sugiyama, K., Tagawa, S., Toda, M.: Methods for visual understanding of hierarchical system structures. IEEE Trans. Syst. Man Cybern. **11**(2), 109–125 (1981). https://doi.org/10.1109/TSMC.1981.4308636
30. Tamassia, R.: On embedding a graph in the grid with the minimum number of bends. SIAM J. Comput. **16**(3), 421–444 (1987). https://doi.org/10.1137/0216030
31. Tang, H., Hu, Z.: Network Simplex Algorithm for DAG Layering. In: 2013 International Conference on Computational and Information Sciences. pp. 1525–1528. IEEE, Shiyang, China (2013). https://doi.org/10.1109/ICCIS.2013.401
32. van der Aalst, W.M.P.: Process Mining - Data Science in Action, Second Edition. Springer (2016)
33. van der Aalst, W.M.P.: Object-centric process mining: Dealing with divergence and convergence in event data. In: Ölveczky, P.C., Salaün, G. (eds.) Software Engineering and Formal Methods - 17th International Conference, SEFM 2019, Oslo, Norway, September 18-20, 2019, Proceedings. LNCS, vol. 11724, pp. 3–25. Springer (2019). https://doi.org/10.1007/978-3-030-30446-1_1
34. van der Aalst, W.M.P., Berti, A.: Discovering object-centric petri nets. Fundam. Informaticae **175**(1-4), 1–40 (2020). https://doi.org/10.3233/FI-2020-1946