A Linear Time Layout Algorithm for Business Process Models

Thomas Gschwind[a,*], Jakob Pinggera[b], Stefan Zugal[b], Hajo A. Reijers[c,d], Barbara Weber[b]

[a] *IBM Research, Zurich, Switzerland*
[b] *University of Innsbruck, Austria*
[c] *Eindhoven University of Technology, The Netherlands*
[d] *Perceptive Software, The Netherlands*

Abstract

The layout of a business process model influences how easily it can be understood. Existing layout features in process modeling tools often rely on graph representations, but do not take the specific properties of business process models into account. In this paper, we propose an algorithm that is based on a set of constraints which are specifically identified toward establishing a readable layout of a process model. Our algorithm exploits the structure of the process model and allows the computation of the final layout in linear time. We explain the algorithm, show its detailed run-time complexity, compare it to existing algorithms, and demonstrate in an empirical evaluation the acceptance of the layout generated by the algorithm. The data suggests that the proposed algorithm is well perceived by moderately experienced process modelers, both in terms of its usefulness as well as its ease of use.

Keywords: layout, graph, workflow, workflow languages, business process model

1. Introduction

Business process models serve a wide variety of purposes where a distinction can be made between models that are to be read by *humans* versus those to be read by *machines* [6]. In the latter case, one may think of *workflow specifications*, as enacted by process-aware information systems, or *simulation models*, which are used to estimate a certain measure's effect. Our concern in this paper, however, is with the former category, i.e., those models that are studied by humans to make sense of how organizational operations are related to one another. It is crucial that such models are understandable to end users from a variety of backgrounds [6].

[*] Corresponding author

The *graphical layout* of a process model has been named as affecting the ease with which a human reader can access the information in such a model [29, 26]. When one is concerned with simplifying the task of reading a process model, layout seems a highly attractive angle. After all, in many situations, the graphical positioning of model elements is at the discretion of the modeler, who is creating the model, or even the user, who wants to read the model. Also, layout, as part of what is known as a model's *secondary notation* [22], does not affect the formal meaning of the model. In this way, focusing on a model's layout allows for a separation of concerns with respect to other quality aspects (e.g., a model's semantic quality).

A considerable body of knowledge exists on how to layout graphical models in order to improve their readability [24, 25]. What is currently missing is a clear understanding of how the specific characteristics of business process models can be taken into account when applying these insights. This hampers the transfer of such knowledge to its practical application. In that respect, it is telling that despite a huge availability of advanced process modeling tools (e.g., IBM Blueworks, Oryx, BPMOne), none of these provide automated layout features beyond elementary alignment operations.

To pick up on this demand, this paper presents an efficient algorithm for clearly laying out business process models. Our approach has been to identify a set of favorable layout constraints from literature, which we subsequently used as the basis for the development of an algorithm that enforces these constraints. The process structure tree as presented in [32] plays an important role in this algorithm, as it provides the hierarchical abstraction of the semantics of the underlying business process. Furthermore, we have validated our theoretical results both in terms of measuring actual performance, the extent to which the proposed algorithm is perceived as useful, and we conducted a comparison between the proposed algorithm with existing ones.

The contribution of the proposed algorithm is to ensure the application of state-of-the-art knowledge to display business process models in a clear and concise way. The approach has been developed for BPMN models that are modeled from left-to-right, but the presented insights can be easily transferred to other flow-oriented languages as well as other model orientations. Our vision is that the algorithm is picked up to be implemented in various electronic modeling environments, in this way improving the quality of the models being created *without* putting an extra load on the modelers themselves. In fact, the provided support may even alleviate a modeler's task, particularly in situations when her modeling experience is limited. When process models are being used within an electronic environment, which is an increasingly realistic option [20], the proposed algorithm directly supports the reader herself in arranging the model in a highly readable form (regardless of the original layout).

The remainder of this paper is structured as follows. Section 2 introduces basic concepts used throughout this paper. Section 3 then elaborates on the layout constraints that seem sensible to follow when laying out business process models. Section 4 introduces the algorithm that takes these constraints take into account to provide automated layout support and discusses the algorithm's

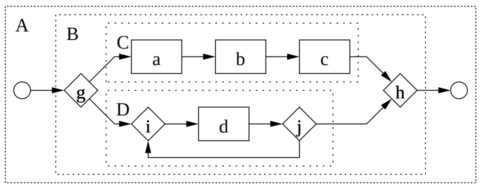


Figure 1: A Process Model and Its Major Sub-Fragments

complexity. Subsequently, Section 5 reports our evaluation of the proposed algorithm in terms of actual run-time performance, its perceived ease of use and usefulness, and gives a comparison of the proposed algorithm with existing ones. Section 6 examines related work. Finally, Section 7 concludes the paper with a summary and outlook.

2. Background

Process models typically describe in a graphical way the activities, events, states, and control flow logic that constitute a business process [3]. Additionally, process models may also include information regarding the involved data, organizational and IT resources, and even other artifacts such as external stakeholders and performance metrics, see e.g. [28].

The process model depicted in Figure 1, for example, consists of a *start event*, and an *end event*, four *activities* (i.e., a, b, c, and d), four *gateways*, and a set of *control-flow edges* connecting these elements. Gateways can be either splitting nodes (i.e., nodes with one incoming and multiple outgoing arcs, like gateways g and j) or joining gateways (i.e., nodes with multiple incoming and one outgoing arc, like gateways h and i).

Similar to other forms of conceptual modeling, process models are often required to be intuitive and easily understandable, especially in the phases of information systems projects that are concerned with requirements documentation and communication [6, 10].

The graph layouting algorithm proposed in this paper is based on the idea of decomposing a process model into single-entry single-exit (SESE) fragments. Figure 1 shows an exemplary SESE decomposition, which we will use throughout this paper. The decomposition of this model results in SESE fragments A, B, C and D. As the name suggests, each SESE fragment has *exactly one* incoming and *exactly one* outgoing edge, irrespective of the internal structure of the fragment. For instance, the internal structure of fragment C is a sequence of activities, whereas fragment D consists of a branched construct. Furthermore, SESE fragments can be embedded in other SESE fragments: note how fragment B aggregates fragment C and fragment D to a SESE fragment.

SESE fragments can be characterized as structured or unstructured. Structured fragments are composed of *blocks*, which may be nested, but must not overlap; i.e., their nesting must be regular [32]. Thereby, a *block* refers to a SESE fragment. In general, structured fragments can be classified as *sequences*, *branching fragments*, *atomic*, and *structured loops*. *Sequences* consist of a sequence of activities or fragments (cf. fragments A and C in Figure 1). *Branching fragments* consist of a diverging (i.e., splitting) gateway as entry and a converging (i.e., joining) gateway as exit. To illustrate a branching fragment, we refer to fragment B in Figure 1. *Atomic* fragments cannot be further subdivided into fragments and represent gateways and individual nodes. *Structured loops* consist of a converging gateway followed by an optional fragment, in turn, followed by a diverging gateway. The latter has an exit branch and one or more branches that loop back (cf. fragment D in Figure 1).

Within unstructured fragments, on the other hand, not all blocks are regularly nested. Depending on whether or not they can be laid out without edge-crossings, they are denoted as either *planar* or *non-planar*.

3. Constraints for Laying Out Process Models

An important consideration when laying out business process models is the set of constraints that should be followed. After a review of the literature, we have identified constraints for laying out business process models [19, 24, 29], as well as trees [30]. Based on this surveys, we have extended and refined these constraints and verified their validity in [12]. In the following, we summarize and motivate these constraints.

C1. Edges should be drawn in the direction of the process's flow. Edges running in the opposite direction of the flow of the process make it hard for humans to understand the process, since it is no longer sufficient to scan the process in one direction. One exception to that rule are edges that are part of a cycle. In such a case, a single edge of that cycle, the back edge, has to flow backwards. This constraint is implicitly present in [24, 29].

C2. Incoming and outgoing edges must be separated. Incoming and outgoing edges can be grasped more easily if they are separated, especially if the diagram is large and has been zoomed out to such a degree that the arrowheads can no longer be identified clearly. Commonly, incoming edges are drawn on the left side of a node, whereas the outgoing edges are drawn on its right side. This constraint has been implemented in commercial modeling tools such as [16] and serves to support the flow of the business process.

C3. Edge crossings should be minimized. The number of edge crossings should be minimized because edge crossings are known to have a significant impact on the understandability of business process models [29, 24]. Specifically, upward planar graphs should be drawn in a planar way. Upward planar graphs are similar to planar graphs (i.e., graphs that can be drawn without crossing edges), but allow edge crossings if these edges would have to encircle the entire process to be drawn in a planar way (cf. Figure 2(a)).

C4. Bendpoints of edges should be minimized. Edges should be drawn with as little bendpoints as possible [29, 24]. Each bendpoint changes the flow of an edge and makes it harder to trace the source and destination of an edge, especially, if edges have to cross other edges.

C5. A Manhattan layout of edges must be used. Edges should preferably use an orthogonal layout that consists of horizontal and vertical lines. This makes it easier to follow edge crossings. However, strictly following the Manhattan layout can lead to edges with an unnecessary high number of bendpoints (cf. Figure 2(b)), contradicting constraint C4 (i.e., bendpoints of edges should be minimized). Hence, we slightly relax this constraint and allow the first or the last segment of an arrow to be drawn diagonally (cf. Figure 2(c)). This constraint is commonly present in business process modeling layout algorithms such as [19, 16].

C6. Minimality should be applied. The business process should consume as little space as necessary [19]. In most cases, this constraint additionally achieves symmetry—another desired property—by putting branches together as close as possible from all sides. However, symmetry may be violated while minimizing bendpoints, which also is considered more important in [29]. Hence, we do not list symmetry explicitly as a constraint.

There is a small number of prominent constraints that we do not consider explicitly. *Clustering* is generally accomplished through SESE decomposition (cf. Section 4). However, the actual position of a modeling element frequently depends on its semantics. Consequently, we did not include *clustering* in the set of constraints underlying the layout algorithm. *Edge length minimization* is intentionally not considered by our algorithm, because of its undesirable effects on business process models. In particular, adherence to it would cause decisions to be pushed away from related elements. For example, assume that elements a, b, and c in Figure 3 belong together. To minimize edge length, however, decision c and the following elements would have to be pushed towards the right, in this way separating elements a and b from c.

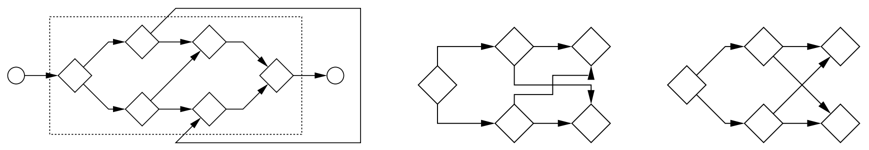

(a) Avoid to Route Edges Around the Process (b) Manhattan Layout (c) Relaxed Manhattan Layout

Figure 2: Constraints

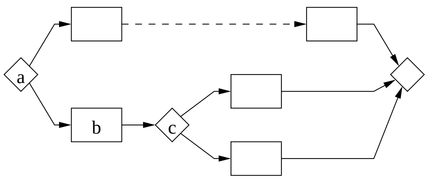


Figure 3: Edge Length Minimization

4. Layout Algorithm

Up to this point, we have introduced desirable constraints for the layout of business process models. In this section, we describe how these constraints are taken into account by our algorithm. We will first provide a general overview of the algorithm in Section 4.1. Next, we will go into more detail for the algorithm's more intricate parts in Sections 4.2, 4.3, and 4.4. Finally, we will reflect on the intended use of the algorithm in Section 4.5.

4.1. Overall approach

Overall, the proposed algorithm follows three phases, as shown in Algorithm 1. First, the process model to be laid out is pre-processed. Second, the resulting process model is decomposed into Single-Entry Single-Exit (SESE) fragments [32]. Third, based upon the SESE fragment's structure, the layout is computed. In the following, we will explain each phase.

Algorithm 1 LAYOUT(g)

```
1: PREPROCESS($g$)
2: $p \leftarrow$ DECOMPOSEPST($g$)
3: LAYOUTFRAGMENT($p$)
```

The PREPROCESS function of the algorithm pre-processes gateways with multiple incoming and outgoing edges to facilitate the model's decomposition into SESE fragments. Respective gateways are split into two gateways: one having all the originally incoming edges and the other covering all the originally outgoing edges. Similarly, multiple incoming and multiple outgoing edges of non-gateways (e.g., activities) are separated into an additional gateway. To illustrate the pre-processing step, Figure 4 shows two exemplary pre-processing transformations. In Figure 4 (a) an additional gateway is introduced to have *either* multiple incoming *or* multiple outgoing edges. Similarly, in Figure 4 (b), two gateways are introduced to make the join gateways and split gateways explicit.

In order to preprocess the process (or graph), we have to visit every node and check the number of incoming and outgoing edges which yields a performance complexity of $O(n)$.

By applying these transformations, the business process model remains semantically equivalent to the original model but ensures constraint C2 (i.e., incoming and outgoing edges must be separated). The gateways introduced during

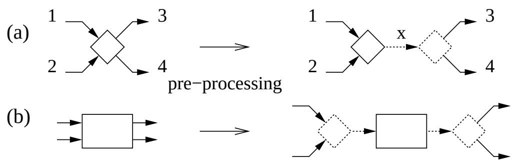


Figure 4: Pre-Processing Steps Applied to the Business Process Model

this pre-processing step are used for the computation of the edge ordering only; they can be removed from the final layout of the business process model, i.e., the algorithm does not change the structure of the model.

Due to the pre-processing, each node has either multiple incoming edges or multiple outgoing edges. The only correct edge orderings for the left gateway are all permutations of $\{1, 2, x\}$ and for the right $\{x, 3, 4\}$. This constrains the permutations allowed for edges 1 to 4 to the permutations of $\{1, 2, \{3, 4\}\}$ which makes sure that edges 1 and 2 and edges 3 and 4 stay together. Since edges are drawn in a clock-wise order, orderings $\{1, 2, 3, 4\}$ and $\{2, 3, 4, 1\}$ are the same.

The function DECOMPOSEPST uses the Process Structure Tree (PST) algorithm to decompose the business process into SESE fragments 2 and runs with a complexity of $O(n)$ (for a detailed description we refer to [32]).

Layout Computation. After the algorithm has pre-processed and analyzed the process model's structure, the actual computation of the layout is carried out as shown in Algorithm 2.

The FRAGMENTS function returns the fragments of a given process or fragment as well as the nodes that are not part of another fragment. For the process model depicted in Figure 1, the function would return:

$$fragments(A) = \{B\}$$

$$fragments(B) = \{C, D, g, h\}$$

$$fragments(d) = fragments(g) = \emptyset$$

In particular, the algorithm makes use of the SESE's tree-structuredness to compute the layout recursively and bottom-up (lines 1–3). This part of the algorithm runs in performance complexity of $\sum c(f_i)$ that is the sum of the complexity used for laying out each individual fragment. Overall, the following steps are performed:

1. Categorize SESE fragment as *structured* or *unstructured*
2. According to this classification, compute the layout and size for the SESE fragment (as detailed in Sections 4.2, 4.3 and 4.4)
3. Use the SESE fragment's size to position it within the tree of SESE fragments

As discussed above, the SESE decomposition partitions the process model into fragments A, B, C and D. According to the nesting of SESE fragments, the

Algorithm 2 LayoutFragment(f)

```
for all $f' \in$ fragments($f$) do
    LayoutFragment($f'$)
end for
if isAtomic($f$) then
    LayoutAtomic($f$)
else if isSequence($f$) then
    LayoutSequence($f$)
else if isBranching($f$) then
    LayoutBranching($f$)
else if isLoop($f$) then
    LayoutLoop($f$)
else
    LayoutUnstructured($f$)
end if
```

algorithm starts by classifying and laying out fragments C and D. Since the size of fragments C and D is now known, it can be used for positioning the fragments within fragment B. Finally, fragment A is laid out—in its turn using the spatial information about fragment B.

This concludes the description of the algorithm's overall structure. In the following we detail how the internal structure of SESE fragments is laid out.

4.2. Structured Fragments

As described in Section 2 structured fragments can be classified as *atomic*, *sequences*, *branching fragments* and *structured loops*. *Atomic* fragments are individual nodes that cannot be split up any further. *Sequences* are laid out as straight lines, ensuring that the exit of one fragment is on the same height as the entry of the next fragment. *Branching fragments*, in turn, are laid out by first arranging the individual branches vertically. Then, the diverging node is put to the left of the branches and the converging node is put to the right, as shown in Figure 5. When laying out the branches, they can be optimized by using the actual shape of the branches and pushing them vertically together as shown in Figure 5, which minimizes the fragment's area.

Since *Structured loops* and sequences are almost identical from a structural perspective, we use a similar strategy for computing the layout. In particular, we lay out the converging gateway, the optional body and the diverging gateway like a sequence. In addition, the loop-back branches are laid out like the branches of a branching fragment. The complexity of each algorithm is $O(|f|)$ where $|f| =$ NODES$(f)| + |$EDGES$(f)|$ which is the size of the fragment at hand. The layout computation of structured fragments is straight-forward and is not discussed any further.

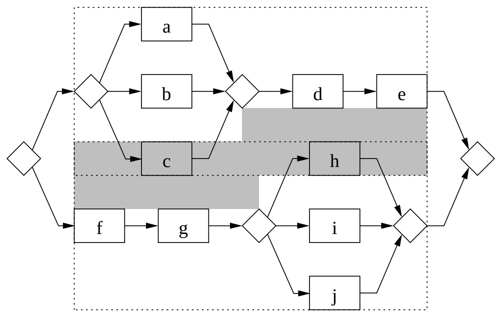


Figure 5: Layout of Branching Fragments

4.3. Unstructured Fragments

In this section, we assume that unstructured fragments are planar. That is they can be laid out without any edges crossing each other. In the following section, we will expand this algorithm to cover non-planar fragments as well. Figure 6 explains the functions and global variables used by this and the following algorithms. The samples used in the figure are based on the process shown in Figure 1.

To layout planar structured fragments, we use Algorithm 3 which consists of three stages. The first identifies back edges and computes the order of the nodes. Not taking back edges into account, nodes have to be ordered such that all prerequistes of a given node n are drawn farther to the left than node n itself. The second stage computes the order in which the edges have to be drawn such that edges do not cross each other. The third stage computes the layout based on the information garthered in the first two stages.

S1. Back Edges and Node Order. This stage computes the order of the nodes to ensure constraint C1 (edges should be drawn in the direction of the process's flow). It consists of three phases.

Back edges are identified using Algorithm 4, which is based on a depth-first search algorithm (DFS). The algorithm uses three arrays, one to store the edge types (edges are numbered consecutively) and two to identify when a node in the DFS algorithm has been discovered and finished (nodes are also numbered consecutively). When the IdentifyEdgeTypes function encounters a node that has not yet been discovered, the edge is part of the graph's spanning tree; if it has not yet been finished, the edge is a back edge; otherwise it is either a forward or a cross edge.

The depth first search algorithm is well known to run with a performance complexity of $O(|f|)$. Also by looking at the algorithm we identify that the recursion is only invoked for nodes which have not yet been visited, hence, each node and each edge is visited exactly once.

In the second phase, the topology order of the graph (without considering back edges) is computed to ensure that nodes depending on other nodes are drawn to the right of their prerequisite nodes [17]. This ensures that nodes

- ENTRYNODE returns the entry node of a given SESE fragment. That is the node to which the single input of the SESE fragment is connected to.

$$\textsc{entryNode}(B) = g$$

- NODES returns the nodes within a given fragment.

$$\textsc{nodes}(B) = \{g, C, D, h\}$$

- EDGES returns the edges within a given fragment.

$$\textsc{edges}(B) = \{gC, Ch, gD, Dh\}$$

- OUTEDGES returns the outgoing edges of a given node.

$$\textsc{outEdges}(g) = \{gC, gD\}, \textsc{outEdges}(h) = \emptyset$$

- REVERSE reverses a given edge, i.e., the source and target of the edge are exchanged. Internally, the edge is still marked as a back edge so that in the final layout, the edge can be drawn correctly.
- SOURCE returns the source node of an edge.
- TARGET returns the target node of an edge.
- NODEX and NODEY are two global arrays that at the end of the algorithm contain the x and y coordinates of each node.
- EDGEY is a global array that contains the y coordinate of each edge at the end of the algorithm. Each edge consists of a diagonal edge from its source node to the y coordinate to a diagonal edge to the target node.

Figure 6: Function Overview

Algorithm 3 LayoutUnstructured(f)

```
 1: {1. Identify back edges and compute the node order}
 2: $type \leftarrow newArray$
 3: $n \leftarrow$ entryNode($f$)
 4: IdentifyEdgeTypes($n, type, newArray, newArray, 0$)
 5: $topology \leftarrow$ TopologySort($f, n$)
 6: $parent \leftarrow newArray$
 7: $length \leftarrow newArray$
 8: LPSTree($topology, type, parent, length$)
 9: {2. Order edges to reduce number of crossings}
10: $f' \leftarrow$ OrderEdges($f, n$)
11: {3. Internally reverse back edges and compute layout}
12: for all $e \in$ edges($f$) do
13:    if $type[e] = BACKEDGE$ then
14:       reverse($e$)
15:    end if
16: end for
17: ComputeBranchDimensions($n, parent$)
18: PreliminaryLayout($n, parent, 0, 0$)
19: CompactLayout($f$)
```

Algorithm 4 IdentifyEdgeTypes($n, type, discovered, finished, t$)

```
 1: $t \leftarrow t + 1$
 2: $discovered[n] \leftarrow t$
 3: for all $e \in$ outEdges($n$) do
 4:    $n' \leftarrow$ target($e$)
 5:    if $discovered[n'] = 0$ then
 6:       $type[e] = TREEEDGE$
 7:       $t \leftarrow$ IdentifyEdgeTypes($n', type, discovered, finished, t$)
 8:    else if $finished[n'] = 0$ then
 9:       $type[e] = BACKEDGE$
10:    else if $discovered[n'] > discovered[n]$ then
11:       $type[e] = FORWARDEDGE$
12:    else
13:       $type[e] = CROSSEDGE$
14:    end if
15: end for
16: return $finished[n] \leftarrow t + 1$
```

are ordered based on their dependencies. This is a prerequisite for fulfilling constraint C1 (i.e., edges should be drawn in the direction of the process's flow).

Algorithm 5 TopologySort(f, n)

```
1: {1. Setup}
2: $inedges \leftarrow newArray$
3: for all $n' \in$ NODES($f$) do
4:    for all $e \in$ OUTEDGES($n'$) do
5:       if $type[e] \neq BACKEDGE$ then
6:          $inedges[n'] = inedges[n'] + 1$
7:       end if
8:    end for
9: end for
10: {2. Topology sort}
11: $topology \leftarrow newArray$
12: $i \leftarrow 0$
13: $topology[i] \leftarrow n$
14: while $i \neq |$NODES($f$)$|$ do
15:    $n' \leftarrow topology[i]$
16:    $i \leftarrow i + 1$
17:    for all $e \in$ OUTEDGES($n'$) do
18:       if $type[e] \neq BACKEDGE$ then
19:          $t \leftarrow$ TARGET($n'$)
20:          $inedges[t] \leftarrow inedges[t] - 1$
21:          if $inedges[t] = 0$ then
22:             $topology[|topology|] \leftarrow t$
23:          end if
24:       end if
25:    end for
26: end while
27: return $topology$
```

Algorithm 5 shows the topology sort algorithm. It first computes the number of incoming edges of each node not counting back edges (Setup). Otherwise the following topology sort algorithm would not be able to correctly compute a topology. The setup iterates over all out edges of all nodes and hence runs with a complexity of $O(|f|)$.

Next, it computes the topology (Topology sort). The first node in the topology is the fragment's start node. The next node in the topology is computed by iterating over the nodes in the topology and by reducing the number of in edges of the nodes reachable from the current node ($n' \leftarrow topology[i]$). Once a node has no more in edges, it must follow the nodes already identified and is appended to the topology array. Since loop edges are ignored, every time the outer loop executes at least one new node is being added to the topology order until all nodes are processed. Hence this part of the algorithm also executes in $O(|f|)$ time.

In the third phase, a spanning tree, representing the longest path to each individual node is computed. This is necessary for identifying the parts of the graph that might be drawn in parallel to each other and for determining the leftmost position a node might occupy. Using the topology sort alone would be insufficient because it would simply return the ordering a, b, c, d, e without indicating which nodes may be drawn in parallel of each other (cf. Figure 7).

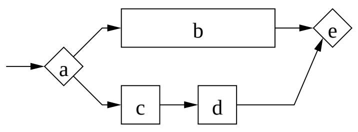


Figure 7: Longest Path Computation

The third phase identifies that node e has to be placed after node b and not after node d, despite the fact that node d is more nodes away from the entry node a. This is accomplished by computing the longest path to each node. The longest path is measured by taking the width of a node and the gap inserted between the node and its successor node. This step ensures that constraint C1 (i.e., edges should be drawn in the direction of the process's flow) is satisfied.

Algorithm 6 shows the longest path spanning tree algorithm. It starts out with the topology order and takes the width of edge nodes into account as well as some extra *space* to allow for drawing the edges between the nodes. For each node in the topology order, it checks all children and checks if the current path is longer than the currently existing path to that node. If yes, it updates the length to that child and the child's parent.

The algorithm iterates over all out edges over all nodes in the topology order which gives $O(|\mathit{topology}| + |\text{EDGES}(f)|)$. Since $|\mathit{topology}| = |\text{NODES}(f)|$, this algorithm has a runtime complexity of $O(|f|)$.

S2. Order Edges. In the second step, our layout algorithm computes the order in which the edges have to be drawn to fulfill constraint C3 (i.e., edge crossings should be minimized). A planar order for the edges of the graph is computed using a left-right planarity checker [14]. This checker computes a spanning tree of the business process's underlying undirected graph and partitions the remaining edges into left and right partitions such that they do not generate conflicts (i.e., edge-crossings).

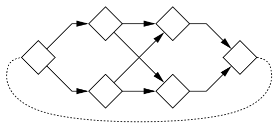


Figure 8: Artificial Edge Added to Each Fragment

Algorithm 6 LPSTree($topology, type, parent, length$)

```
for all $n \in topology$ do
   $w \leftarrow width[n] + space$
   for all $e \in$ outEdges($n$) do
      if $type[e] \neq BACKEDGE$ then
         $t \leftarrow$ target($e$)
         if $length[t] < length[n] + w$ then
            $length[t] \leftarrow length[n] + w$
            $parent[t] \leftarrow n$
         end if
      end if
   end for
end for
```

Before passing a fragment to the planarity checker, it is pre-processed such that edges are no longer permitted to be drawn around the entire process (cf. Figure 2(a)). We accomplish this by adding an artificial edge from a fragment's entry node to the fragment's exit node (cf. Figure 8) which prohibits an edge on either the upper or the lower side of a fragment to be drawn around the entry or exit nodes since it would need to cross the artificial edge.

The planarity checker returns a planar order for edges indicating how to draw them in order to avoid edge crossing. In this stage all edges (i.e., including back edges that were omitted previously) are considered. The planar order will be used in subsequent stages. If a graph is non-planar we perform some extra processing to find an optimal edge order. This modification will be discussed in Section 4.4.

For the details of the planarity checking algorithm, we refer the interested reader to [14] where Hopcroft et al. show that the algorithm runs with a complexity of $O(|f|)$.

S3. Compute Layout. The third and last step computes the layout of the graph. It consists of four phases. In the first phase, all back edges are reversed, so that they are treated like cross and forward edges. These edges need to be marked in some table such that in the final drawing of the process, they can again be drawn in their correct direction.

Phases two and three compute a preliminary tree layout on the basis of the longest path spanning tree. Each branch is put into its own horizontal section such that the branches are not overlapping each other (cf. Figure 9).

All non-tree edges are considered to be branches of zero height that run up to the point where they merge with another branch. For instance, the branch with e and f in Figure 9, extends up to the converging gateway d due to the forward edge from f to d. This ensures that edges can be drawn in a straight manner up to the branch with which they are to be connected to. This is also the reason why it is sufficient to store only the y position of the edges in edgeY.

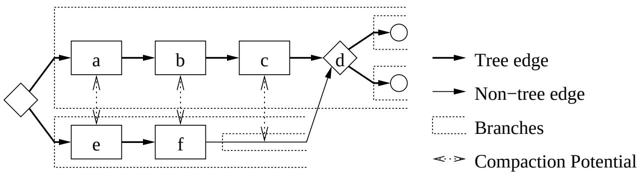


Figure 9: Layout of the Branches of the Spanning Tree

Algorithm 7 ComputeBranchDimensions($n, parent$)

```
1:  $(h, w) \leftarrow (0, 0)$
2:  $k \leftarrow 0$
3:  **for all** $e \in$ outEdges($n$) **do**
4:      **if** parent[target(e)] = source(e) **then**
5:          ComputeBranchDimensions($parent$, target($e$))
6:          $(h', w') \leftarrow (branchheight[\text{target}(e)], branchwidth[\text{target}(e)])$
7:      **else**
8:          $(h', w') \leftarrow (0, 0)$
9:      **end if**
10:     $(h, w) \leftarrow (h + k * space + h', \text{max}(w, w'))$
11:     $k \leftarrow 1$
12: **end for**
13: $(h, w) \leftarrow (\text{max}(height[n], h), width[n] + k * space + w)$
14: $branchheight[n] \leftarrow h$
15: $branchwidth[n] \leftarrow w$
```

In phase two, the height of each branch in the tree is computed (Algorithm 7). The algorithm first assumes the branch to be of size 0, goes over all outedges of the current node, and adds the height of each subbranch plus some extra space for aesthetic reasons. Line 4 covers tree edges which point to paths in the process having other nodes. Line 7 covers other edges which connect in the tree to other branches. At the end we check whether the height of the current node is higher than all the branches and expand the height of the branch (*branchheight*) if necessary ($\textsc{max}(height[n], h)$). We also compute the width of the branches which is not used in this presentation of the algorithm but could be used for future optimization purposes.

Algorithm 8 $\textsc{PreliminaryLayout}(n, parent, x, y)$

```
1: $\textsc{nodex}[n] \leftarrow x$
2: $\textsc{nodey}[n] \leftarrow y + (branchheight[n] - height[n])/2$
3: $h \leftarrow 0$
4: $k \leftarrow 0$
5: for all $e \in \textsc{outEdges}(n)$ do
6:    if parent[$\textsc{target}$(e)] = $\textsc{source}$(e) then
7:       $h \leftarrow h + k * space + branchheight[\textsc{target}(e)]$
8:    else
9:       $h \leftarrow h + k * space$
10:   end if
11:   $k \leftarrow 1$
12: end for
13: $x \leftarrow x + width[n] + space$
14: $y \leftarrow y + \textsc{max}(0, (branchheight[n] - h)/2)$
15: for all $e \in \textsc{outEdges}(n)$ do
16:    if parent[$\textsc{target}$(e)] = $\textsc{source}$(e) then
17:       $\textsc{PreliminaryLayout}(\textsc{target}[e], parent, x, y)$
18:       $y \leftarrow y + branchheight[\textsc{target}[e]] + space$
19:    else
20:       $\textsc{edgey}[e] \leftarrow y$
21:       $y \leftarrow y + space$
22:    end if
23: end for
```

Once we have computed the dimensions of the branches, we can compute a preliminary layout of the process (phase three). Preliminary, because the layout is not yet compact. Algorithm 8 shows this part of the algorithm. After it has completed, the global arrays NODEX and NODEY contain the coordinates where to draw the nodes and the global array EDGEY contains the y coordinate where to draw the corresponding cross, forward, or back edges. The edges' start and end x coordinates are derived from the source and target nodes.

This is very similar to Algorithm 7. We first compute the location of the current node and then the height of the subsequent branches (not taking the current node into account). If memory is not of concern this intermediary result could have been maintained by Algorithm 7 in an extra table.

The height of the subsequent branches is necessary because the current node could be higher then the height of all the subbranches combined. In this case we need to offset the subsequent branches (line 14). This, together with the fact that in the current version of the algorithm, the coordinates are absolute within a given fragment, is the reason that these two algorithms are separate. If locations are relative to their branches and branch information is stored until all fragments have been computed, these two algorithms could be combined.

Both algorithms (7 and 8) iterate over every out edge in the spanning tree of the process. The latter actually twice, which yields a combined runtime complexity of $O(3|f|)$ for phases two and three.

Edges are laid out with the relaxed Manhattan layout (i.e., constraint C5). Tree edges are laid out with at most one bendpoint. All other edges have a maximum of two bendpoints ensuring C4 (i.e., bendpoints of edges should be minimized). Depending on the modeler's personal aesthetic preferences, back edges may be laid out with more than two bendpoints.

In the fourth phase, we compute how much closer a given branch might be moved to the branch drawn directly above it. Second, the tree is compacted to ensure constraint C6 (i.e., minimality should be applied). In Figure 9, this stage would shift nodes e and f closer to nodes a to c since the lower branch ends before the upper branch requires more space to fit in the final nodes.

The detailed algorithm to accomplish this part would be a lengthy discussion of all the different combinations of when an element can be moved closer to another. Hence, we limit the discussion to presenting the overall approach and to deriving the runtime complexity from this discussion.

A spanning tree and the spaces between the elements of neighboring branches are computed. For computing the runtime complexity of the compaction algorithm, let us represent the space between neighboring elements as a graph such as the double-headed arrows in Figure 9. These arrows form a planar graph and a planar graph is know to have a maximum of $3v - 6$ edges, where v is the number of vertices for graphs with three or more vertices [14] ($v < |f|$). During the compaction phase we iterate over this graph twice, assessing how much each element may be moved (once to compute the minimum for moving the entire branch, and once for the actual compaction). Hence, this part of the algorithm runs in $O(6 * |f|) = O(|f|)$ time.

So far, we have presented the runtime complexity of each individual phase for laying out unstructured fragments. Each phase has a linear runtime complexity and as shown in Algorithm 3, the phases are executed sequentially giving us a runtime complexity of $O(|f|) + O(2|f|) + O(|f|) + O(|f|) + O(|f|) + O(3|f|) + O(6|f|) = O(15|f|) = O(|f|)$.

4.4. Non-Planar Unstructured Fragments

In order to adapt the algorithm presented for planar fragments to non-planar ones, stage S2 (i.e, order edges) has to be modified to find an edge ordering producing a minimal number of edge crossings. The general problem of minimizing edge crossings is known to be NP complete [9]. Hence, we use a heuristic that is based on the edge ordering algorithm itself. When the planarity checker identifies a conflict, we ignore it and continue to sort edges eliminating crossings of the remaining edges. This approach does not change the algorithmic behavior of the planarity tester and hence has the same runtime complexity of $O(|f|)$.

To verify that this heuristic has a positive effect on the number of edge crossings, we compared it to a random order of the edges using a collection of 850 insurance processes. After a visual inspection, we observed that using our heuristic the non-planar fragments looked better than with random edge-ordering since our heuristic managed to remove many of the unnecessary edge crossings present when using a random edge order.

Surprisingly, however, even the random edge ordering produced fragments that looked clean and are easy to understand. The reason for this is that unstructured fragments in business processes are relatively rare and contain a small number of nodes limiting the number of potential crossings. This finding has also been observed in [33].

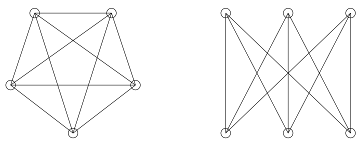


Figure 10: K_5 and $K_{3,3}$ Graphs

Due to the positive results we have received so far with our simple heuristic, we have not invested more efforts in further improving it. Possible extensions would be to identify the $K_{3,3}$ and K_5 subgraphs (cf. Fig. 10), which are responsible for the non-planarity. The planarity tester can be extended to compute these in linear time and to minimize edge crossings introduced by these subgraphs. Another approach would be to use the heuristics presented in [34], which also produce attractive results [8].

4.5. Complexity Analysis

The proposed layout algorithm is intended to be used during business process modeling, not only as a post-processing step. The modeler of the process is envisioned to be empowered to automatically layout the process model at any time, for example by calling a layout service through the push of a button of the modeling tool's GUI. In other words, modeling actions and layout invocations can be interwoven. In this way, the modeler herself can benefit from the increased clarity *during* the modeling session itself.

To ensure the feasibility of this intended use it is paramount that the invocation of the automated layout is not perceived as a nuisance: the layout's computation *must* be performed efficiently, not wasting the modeler's time to an unacceptable degree. In order to validate that our algorithm fulfills this requirement, we have performed a complexity analysis and described the algorithm's runtime behavior using the *O-Notation* [5].

Table 1 summarizes the results of the individual parts of the algorithm which have been presented in the previous sections. In the table, n is the sum of number of vertices of all nodes and edges and f_i the number of vertices and edges of a given fragment.

Phase	**Runtime**
Phase 1: Pre-Processing	$O(n)$
Phase 2: SESE Decomposition	$O(n)$
Phase 3: Computation of Layout	$\sum c(f_i)$
Structured Fragments	$O(\|f_i\|)$
Unstructured Fragments	$O(\|f_i\|)$
Non-planar Unstructured Fragments	$O(\|f_i\|)$
Total	$O(n)$

Table 1: Complexity Analysis

So far, we considered complexities for each phase individually. To obtain the overall performance complexity of the layout algorithm these numbers have to be combined. For this, we will now expand the runtime complexity of phase 3, which is the sum of the complexities of the layout computations of the different fragments. Since the complexity of all the different fragments is the same, hence we get $\sum c(f_i) = \sum O(|f_i|)$. Since $\sum |f_i| = n$, we get an overall complexity of $O(n)$ for phase 3. Since each individual phase has a complexity of $O(n)$ and the phases are executed sequentially, the total runtime complexity is $O(3n) = O(n)$.

With a resulting overall complexity of $O(n)$ we can confidently conclude that the proposed layout feature is suitable to be used by implementors of process models at any time during process modeling.

5. Evaluation

So far we have described the mechanics of the proposed layout algorithm and conducted a theoretical performance examination. To further substantiate the evaluation, we have also applied the layout algorithm to an extensive collection of real-world process models, assessed its perceived usefulness and ease of use as well as compared it with other layout algorithms.

5.1. Performance Evaluation

To empirically verify that our implementation of the algorithm indeed runs in linear time, we have also applied it to a collection of approximately ∼700 insurance process models. These insurance process models range from small to large models with about thousand modeling elements (branches and edges). The results of this evaluations are shown in Figure 11 and Figure 12.

Each dot in Figure 11 represents the size of a process model (x-axis) and how long it took to lay out the process model in ms (y-axis). Looking at the

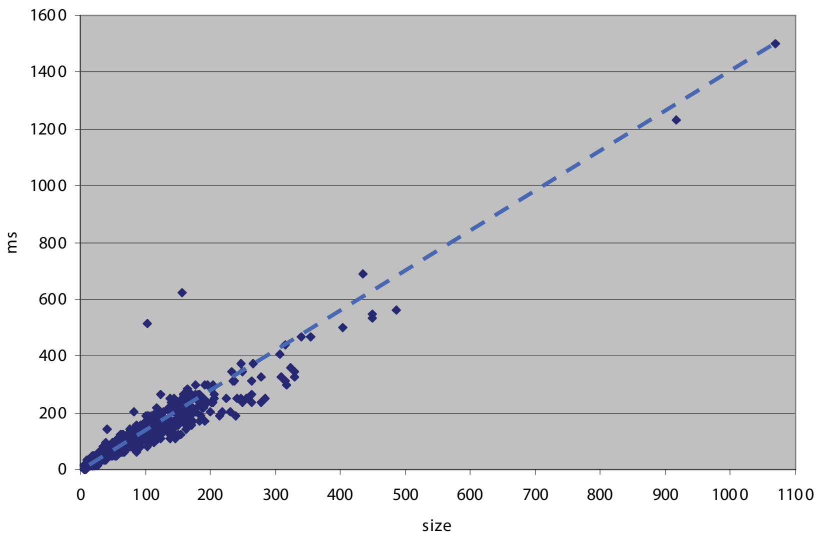


Figure 11: Performance of the Layout Algorithm

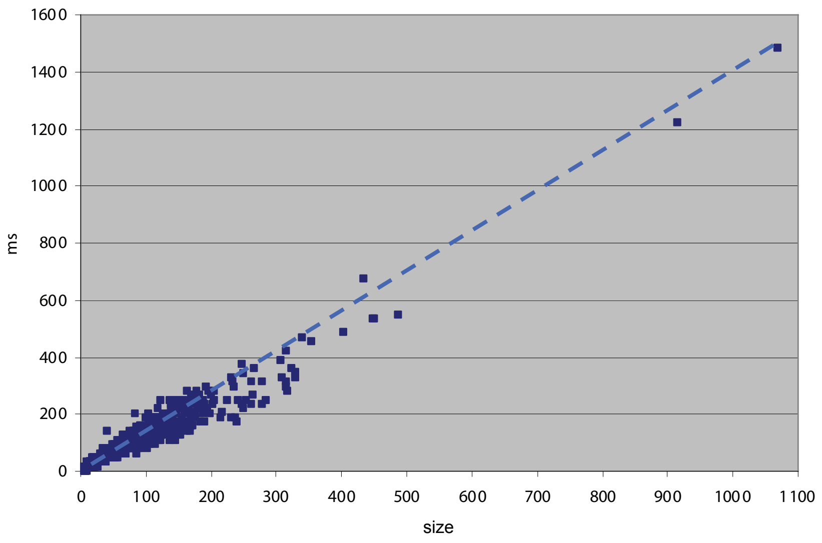


Figure 12: Performance of the Layout Algorithm (excluding PST processing)

graph we can see that all the dots are scattered around a line showing that our implementation in fact has a linear time performance as we have proven in Section 4.

In the figure, we can identify two dots that are somewhat off the linear distribution. After further investigation we have identified that this anomaly is generated by the PST computation as can be seen in Figure 12 which shows the same timings but excluding the time it takes to compute the PST. Unfortunately, the original developer of the PST library is no longer available and hence we cannot explore this anomaly any further.

We can summarize that typical process models with sizes of up to 100 modeling elements can be laid out in under 200ms or 600ms including the PST anomaly. Even process models with up to 500 elements can be laid out in about half a second, a time that is well suited for using the algorithm interactively.

5.2. Empirical Evaluation

This section discusses the empirical evaluation of the layout algorithm in terms of perceived ease of use and perceived usefulness. For this purpose, a modeling session with students following a graduate course on Business Process Management was conducted. The evaluation builds on classic notions of perceived usefulness and perceived ease of use measures, as described in [4], to evaluate the acceptance of the layout algorithm. In essence, the underlying theory in [4] postulates that actual usage of an information technology artifact—an automated layout feature in our context—is mainly influenced by the perceptions of potential users regarding usefulness and ease of use. Accordingly, a potential user who perceives the feature to be useful and easy to use would be likely to actually adopt it when modeling business processes. Similar set-ups have been applied in the evaluation of other artifacts in the field of business process modeling, see e.g. [27].

5.2.1. Subjects

For our evaluation, we were targeting subjects who would be at least moderately familiar with imperative process modeling languages, preferably with BPMN. Without such experience, the effects would be blurred by subjects struggling with the modeling notation itself. In a similar vein, the subjects needed to represent a rather homogeneous group in terms of prior domain knowledge as this might affect their modeling behavior [18].

5.2.2. Objects

The object of our study is a process modeling task, starting from an empty canvas using a subset of BPMN. Subjects received a textual description of the process model[1] to be created, designed to reach a medium level of complexity going well beyond a "toy-example". To ensure that the process model is of appropriate complexity and not misleading, the textual descriptions were refined

[1]Material available at http://bpm.q-e.at/experiment/AutomatedLayoutSupport

in several iterations. Additionally, we performed a pre-test involving ten students with a similar background. The gathered feedback allowed us to further refine the modeling task, e.g., remaining ambiguous parts of the descriptions were clarified. In the end, the process model to be created consisted of 25 activities covering several basic control-flow patterns, i.e., sequence, structured loop, parallel split, synchronization, exclusive choice and simple merge [31].

5.2.3. Response Variables

In order to asses the usability of the layout algorithm we assess perceived usefulness and perceived ease of use. For the assessment, we followed the approach proposed in [4]. Perceived usefulness is defined as "the degree to which a person believes that using a particular system would enhance his or her job performance". Perceived ease of use, on the contrary, focuses on "the degree to which a person believes that using a particular system would be free of effort". We strove for keeping changes to the questions proposed in [4] to a minimum to avoid introducing any bias. Therefore, we only changed the name of the tool under consideration. For example, *"I would find it easy to get CHART-MASTER to do what I want it to do"*, proposed in [4], was changed to *"I would find it easy to get the automated layout feature to do what I want it to do"* for the current evaluation.

5.2.4. Instrumentation and Data Collection.

We utilized the Cheetah Experimental Platform (CEP) [23] to collect the data for the assessment of the usability of our layout algorithm. The utilization of CEP minimizes the danger of misunderstandings and students accidentally disobeying the setup by guiding them through the evaluation [23]. Additionally, all data was automatically transferred and stored on a central database server preventing potential data loss.

In particular, we employed CEP's BPMN modeling editor to provide subjects with a modeling environment that allowed them to focus on the task of modeling without being distracted by an abundance of software features. The built-in modeling tutorial informed the subjects on how the modeling editor and the provided layout feature could be used.

Figure 13 illustrates how the layout algorithm was implemented in Cheetah Experimental Platform for the empirical evaluation.

5.2.5. Execution

The evaluation was conducted in October 2011 at Eindhoven University of Technology as part of a graduate course on Business Process Management. 64 students participated in the evaluation, which resulted in the creation of 64 business process models. The evaluation was guided by CEP's experimental workflow engine [23], subsequently leading students through an initial questionnaire collecting demographic data, the modeling task, a survey containing the perceived usefulness and perceived ease of use questions, and a feedback questionnaire.

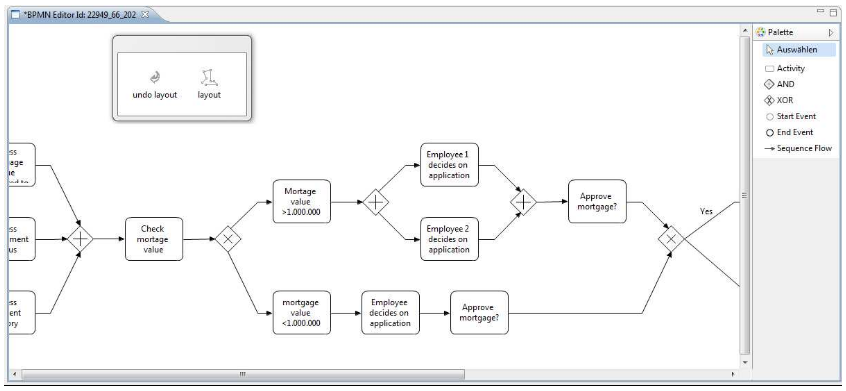


Figure 13: Layout Feature in Cheetah Experimental Platform

5.2.6. Data Validation

To ensure that the subjects match the targeted profile, i.e., modelers who are at least moderately familiar with BPMN, we used a questionnaire to assess familiarity with BPMN, confidence in understanding BPMN, and competence in using BPMN. To quantify these factors, we used a seven-point Likert-Scale, ranging from *strongly disagree* (1) over *neutral* (4) to *strongly agree* (7). The subjects were found to be moderately familiar with BPMN (mean value: 4.13), moderately confident in understanding BPMN (mean value: 4.64) and consider themselves moderately competent in using BPMN (mean value: 4.17). Thus, we conclude that the subjects are reasonable proxies for average process modelers, i.e., the targeted profile. As domain knowledge is known to influence problem solving tasks [18], we controlled the students' familiarity with the problem domain by asking them whether they are familiar with mortgage processes. The same Likert-Scale as before was utilized resulting in an average familiarity of 3.11. The standard deviation was relatively low (1.48). Therefore, we conclude that the students represented a rather homogeneous group in terms of prior domain knowledge. Only one student reported very high familiarity with mortgage processes (*strongly agree*) and another one reported high familiarity (*agree*). Additionally, we controlled for the actual usage of the layout feature since feedback on perceived usefulness and perceived ease of use of students who did not use the layout feature cannot be considered meaningful.

5.2.7. Data Analysis and Results

In line with the data validation, we dropped the two students who reported high familiarity with the problem domain. Similarly, we did not consider one student for further analysis who did not utilize the layout feature at all. The remaining 61 students, who comprise our core data set, invoked the layout feature between 1 and 10 times during the entire session, with an average of 3.52. Taking into account that the average modeling session lasted 35 minutes, this relates to an average student invoking the layout feature every 10 minutes.

These figures suggest that a reasonable basis is present for the further evaluation of the data.

To assess the perceived usefulness and perceived ease of use of the layout feature, we averaged for each subject the scores for the six questions that were used for each of these variables. Recall that all questions were measured on a Likert scale that ranged from 1 to 7 (with 4 as the neutral value). This computation led to average values for perceived usefulness of 4.89 and for perceived ease of use of 5.24. The median values are higher, respectively 5.67 and 5.33. In other words, 50% of the respondents agree that the layout feature is useful, and the same amount of respondents find 'somewhat agree' that the layout feature is easy to use. Since the median values are higher than the average values, the distributions seem skewed. To analyze the distribution of the evaluation of the layout feature in more detail, we we will be using the histogram as shown in Figure 14.

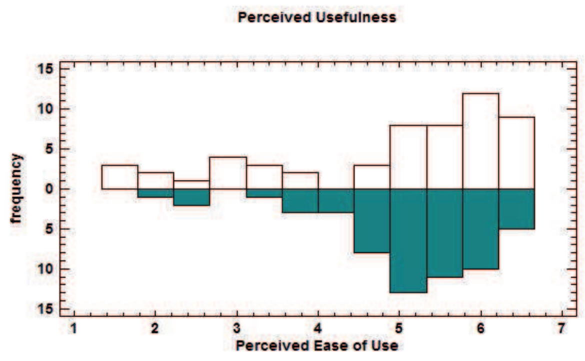


Figure 14: Distribution of Perceived Usefulness and Perceived Ease of Use

Depicted in the histogram are the frequencies of the various values for perceived usefulness and perceived ease of use, in 12 equally distributed bands. What can be observed clearly is that the bulk of respondents deliver a value of 4.5 or higher, both for perceived usefulness and ease of use. These insights corroborate with the high median values we noted before. The appreciation of usefulness seems even higher than that of ease of use, but it is also characterized by more variation across the subjects. Overall, the modelers can be said to be quite positive about the usefulness as well the ease of use of the layout feature.

5.2.8. Limitations

Our empirical findings are presented with the explicit acknowledgement of a number of limitations. First of all, our subjects represented a rather homogeneous group. More experienced process modelers may have laid out the process models differently, following different constraints than the constraints underlying the layout algorithm presented in this paper.

Secondly, all students were working on the same process model that might favor the satisfaction of some constraints. We tried to compensate for this problem by including a variety of model constructs when designing the process modeling task. Specifically we referred to the *workflow patterns* [31] to include sequence, parallel split, synchronization, exclusive choice, simple merge, and structured loop constructs.

5.3. Comparison with Existing Layout Algorithms

Up till this point we have elaborated on runtime efficiency (cf. Section 5.1) as well as perceived usefulness and ease of use of the proposed algorithm (cf. Section 5.2). In the following, we compare the proposed layout algorithm with existing approaches. Due to the large number of existing modeling environments and the diversity of layout functionality provided in these, we have chosen two representatives for layout algorithms. In particular, we selected the dot algorithm [8] implemented in the Graphviz [11] framework, which provides a *general-purpose layout*, i.e., an algorithm that works for graphs in general. In addition, we have selected yWorks [35], which provides a layout algorithm *specifically designed for business process models*. Generally speaking, it can be said that our proposed algorithm as well as the dot algorithm are available freely, while yWorks is a proprietary implementation. We have selected yWorks and Graphviz because we consider them high-quality frameworks. In particular, yWorks claims to be used by several renowned institutions, such as *Northrop Grumman IT* and the *Massachusetts Institute of Technology*[2]. Graphviz, in turn, has been under development since 1988 and claims to have *"important applications in networking, bioinformatics, software engineering, database and web design"*[3].

To evaluate the selected algorithms, we have used the process model collection described in Section 5.1. Obviously, it is not possible to list all of the ∼700 process models here, so we decided to take a representative set of process models and laid them out using three approaches: according to the proposed layout algorithm, by using yWorks, and by applying the dot algorithm. To representatively select process models, we have taken parameters that are central for the layout of a process model (cf. Section 2): *structured* (yes/no), *cyclic* (yes/no) and *planar* (yes/no). Of the expected 2^3 categories, however, only 6 are valid because there are no structured models that are also non-planar. An overview of all models is provided in Table 2. In the following we discuss the table on and show a subset of the models used for the comparison. The complete set of models we used for the comparison is available online.[4]

First, The most prominent differences can be observed with respect to the layout of structured process fragments. As our proposed algorithm decomposes the process model into SESE-fragments, the layout can take advantage from

[2]http://www.yworks.com/de/company_references.html
[3]http://www.graphviz.org/
[4]http://bpm.q-e.at/misc/LayoutComparison

Model 1 Cyclic: Yes, Structured: Yes, Planar: Yes

	Structure	Crossings	Compactness	Docking[a]	B. Edges[b]
Ours	+	0	+	+	+
yWorks	∼	0	+	+	−
Dot	∼	0	+	−	−

Model 2 Cyclic: Yes, Structured: No, Planar: Yes

	Structure	Crossings	Compactness	Docking[a]	B. Edges[b]
Ours	+	0	+	+	+
yWorks	∼	0	+	+	−
Dot	∼	0	+	−	−

Model 3 Cyclic: Yes, Structured: No, Planar: No

	Structure	Crossings	Compactness	Docking[a]	B. Edges[b]
Ours	+	27 (24)	−	+	+
yWorks	∼	10	∼	+	−
Dot	−	23	+	−	−

Model 4 Cyclic: No, Structured: Yes, Planar: Yes

	Structure	Crossings	Compactness	Docking[a]	B. Edges[b]
Ours	+	0	∼	+	n/a[c]
yWorks	∼	0	∼	+	n/a[c]
Dot	−	4	+	−	n/a[c]

Model 5 Cyclic: No, Structured: No, Planar: Yes

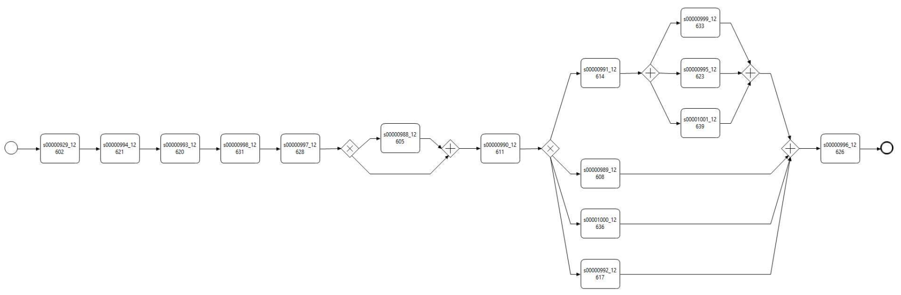
	Structure	Crossings	Compactness	Docking[a]	B. Edges[b]
Ours	+	0	−	+	n/a[c]
yWorks	∼	0	∼	+	n/a[c]
Dot	∼	2	+	−	n/a[c]

Model 6 Cyclic: No, Structured: No, Planar: No

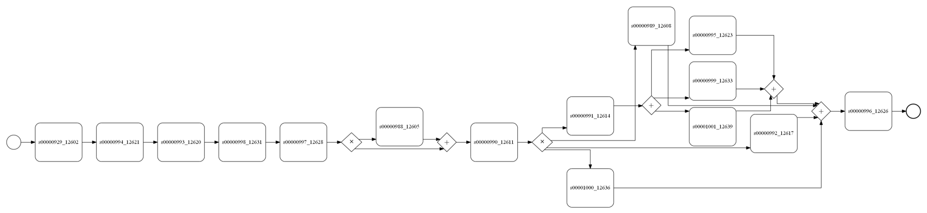
	Structure	Crossings	Compactness	Docking[a]	B. Edges[b]
Ours	∼	3 (2)	+	+	n/a[c]
yWorks	∼	1	+	+	n/a[c]
Dot	∼	1	+	−	n/a[c]

Grades: (+) fulfilled; (∼) somewhat fulfilled; (-) not fulfilled
[a] Docking points for edges from/to activities are centered
[b] Back edges are indicated explicitly
[c] not applicable, model is non-cyclic

Table 2: Comparison of Algorithms

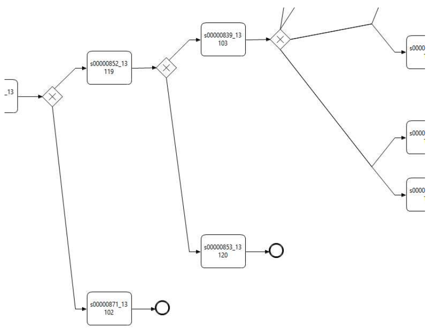


(a) Our Algorithm

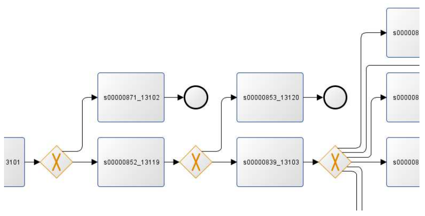


(b) The dot Algorithm

Figure 15: Compactness and Structure Comparison

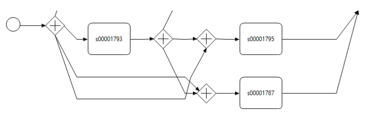


(a) Our Algorithm

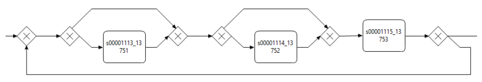


(b) yWorks

Figure 16: Compactness Comparison



(a) Edge Crossings



(b) Back Edges

Figure 17: Our Algorithm

this information. Likewise, the structure of the process models can presumably be easiest seen when the model is laid out using our proposed algorithm. Also, in yWorks the structure of the process model can still be perceived. The dot algorithm, however, has apparently problems to display the structure of the process model—probably due to the fact that it is a general purpose algorithm for laying out graphs. This is also shown in Figure 15 that shows model laid out with our algorithm versus the dot algorithm.

With respect to bendpoints, even if not explicitly listed in Table 2, yWorks and our algorithm introduce additional bendpoints (cf. Figure 15) to lay out edges—the dot algorithm, in turn, appears to minimize bendpoints. Even though bendpoints should be minimized in general (cf. Section 3), we would argue that additional bendpoints improve the readability in this case.

Second, we counted the number of edge crossings. We found that the smallest number of edge crossings were achieved by yWorks. Our proposed algorithm and the dot algorithm both produced the same number of edge crossings. Apparently the usage of structural information, as used in the proposed algorithm, helps to reduce the number of edge crossings for *structured* models (cf. Figure 15). We also realized that the edge-routing in our algorithm sometimes introduced unnecessary edge crossings as shown by model 6 in Figure 17(a) by drawing the down arrow of the cross edge (middle edge) too late. Without that bug this model would only have 2 edge crossings (number given in parenthesis in Table 2. Still, looking at yWorks, it becomes clear that our algorithm for non-planar unstructured fragments could be improved further.

Third, we looked at the compactness of the layout algorithms. The dot algorithm produces the most compact layout, but as we have discussed evidently in a way that is detrimental to readability. On the other hand our proposed algorithm, highlights the structure of a process model, in certain cases, which in certain cases is detrimental to its compactness (Model 3). We also have to admit that for structured fragments we have not yet implemented the compaction phase which we have discussed in Section 4.3, stage 3. This affects Model 5 as shown by the excerpt in Figure 16. yWorks on the other hand lays out this model to compact to our taste.

Fourth, the proposed algorithm and yWorks consistently docked edges centrally to activities, while the dot algorithm varied docking points in order to improve the compactness of the layout (cf. Figures 15 and 16).

Finally, back edges are laid out so that they can be easily identified as such in our algorithm (cf. Figure 17(b)). Due to our partitioning between out and in edges, the loop can be identified easily even on small print when arrow heads can no longer be identified easily. As it has been shown that back edges potentially cause problems in understanding process models [21], it seem desirable to explicitly emphasize back edges. yWorks and the dot algorithm did not separate input from output edges or otherwise did not lay out back edges in any particular way that allows them to be identified easily.

Summarizing, it can be said that each algorithm shows strengths and weaknesses. The proposed algorithm appears to be specifically well suited for visualizing the structure of a process model and the layout of back edges. We consider both of these aspects as highly supportive for the task of sense-making of process models. The yWorks algorithm, in turn, appears to be suitable for laying out unstructured process models, i.e., it minimizes edge crossings. Still, the structure of the process model is not that clearly visualized, which is an important drawback. Finally, the dot algorithm produces the most compact layout. On the downside, it also shows deficiencies with respect to the ability to visualize the structure of the process model and the number of edge crossings.

6. Related Work

The importance of laying out business process models has been identified in [29]. While this work confirms the importance of a good layout of business process models for their understandability, we presented an algorithm for automatically laying out business process models and verify that the underlying layout characteristics improve over the modeling behavior of process modelers, hence complementing that work.

yFiles which is at the heart of yWorks is an extensive Java class library that provides algorithms and components enabling the analysis, visualization, and the automatic layout of graphs, diagrams, and networks [35]. yFiles is also used by WebSphere Business Modeler for laying out business process models, even though it does not provide a specific algorithm for business process models. The underlying layout algorithm is proprietary and its runtime behavior is unknown.

An important and widely used graph layout tool, although not directly related to business process models, is Graphviz which utilizes the dot algorithm [8], The dot algorithm uses a four stage approach for laying out graphs. First, ranks are assigned to the nodes and the vertex order is determined for minimizing edge crossings. Afterwards, positions are assigned to the nodes before computing splines for laying out the graph's edges. The dot algorithm, however, tries to minimize the length of edges, a characteristic not always desirable for laying out business process models as shown in Section 2 and 5.3.

A simple algorithm to automatically lay out business process models has been presented in [19]. This is done by arranging elements in a topological order and using a grid that allows to insert new elements by inserting rows and hence pushing other elements aside to make room for new elements. This approach by itself may create space-consuming layouts, hence heuristics are used to improve on this situation. The runtime complexity of this approach has not been assessed.

In [1], Di Battista presents a set of algorithms for plane representations of acyclic digraphs, supporting visibility graphs, grid drawings, and straight line drawings ranging from $O(n)$ to $O(n \log n)$. The grid drawings are similar to our approach, but do not support cyclic or non-planar graphs. Similar approaches can be found in [15, 2].

A problem related to laying out business process models is the layout of trees. In [13], Hasan presents an algorithm to draw a tree in a compact form. We use this algorithm for laying out the spanning tree of unstructured fragments and extended it to support the layout of non-tree edges (cf. Section 4.3).

The ILOG JViews Component Suite is a set of components for Web-based user interfaces, including a component specifically designed for building graphical user interfaces for workflow applications [7]. It is a general-purpose graph drawing tool based on a constraint system allowing, among others, the definition of relations between node positions and the placement of incoming and outgoing edges. In contrast, the layout algorithm presented in this paper relies on a set of pre-defined constraints specifically tailored for the needs of business process models (cf. Section 2).

7. Conclusions

In this work we presented an automated layout algorithm that is specifically tailored to the characteristics of business process models. In particular, we improved over state-of-the-art generic layout algorithms by following a three-phase approach to pre-process, structure and layout business process models. The proposed algorithm partitions the model into SESE fragments that can be laid out using a bottom-up approach. Additionally, we have combined ideas used for laying out graphs with those for laying out trees, essentially, computing a spanning tree and treat cross and forward edges like branches in the tree. In a run-time complexity analysis, we have shown that our algorithm runs in $O(n)$, i.e., linear time.

Furthermore, we performed an empirical evaluation of the proposed algorithm. We measured the actual execution times of the implementation to confirm the linear execution time of our implementation. We showed that models with up to 500 modeling elements can be laid out in under 800ms, the reaction time of a human, which ensures that its on-demand application is also feasible in real-world modeling settings. We established that the automatic layout algorithm is perceived both useful and easy to use by process modelers with moderate modeling experience. Based on these findings, we argue that our algorithm is both effective and attractive to create process models that are more easily read and understood.

Moreover, we compared the proposed algorithm with existing algorithms, i.e., yWorks and the dot algorithm. This comparison showed that the proposed algorithm is particularly well suited for visualizing the structure of a process model and the layout of back edges. It is less suited, in turn, when compactness of the layout is more important than the process model's readability.

Additional potential benefits that have not been discussed so far are that an automatic layout can be utilized for computer-generated models when no human modeler is available at all. Furthermore, the use of the algorithm will allow for a consistent layout across models, i.e., all constraints are fully supported. This may be particularly beneficial in the context of large process modeling initiatives where several process modelers create process models, which are to be read by many different stakeholders. Establishing whether this benefit materializes and, if so, to what extent will be a topic for further research.

Acknowledgements. Research on Cheetah Experimental Platform, which was used for the empirical evaluation, was funded by the Austrian Science Fund (FWF): P23699-N23. The authors thank all the students who participated in the evaluation and are particularly grateful for the assistance of Heidi Romero and Ronny Mans in its execution.

References

[1] Giuseppe Di Battista and Roberto Tamassia. Algorithms for plane representations of acyclic digraphs. *Theoretical Computer Science*, 61(2–3):175–198, 1988.

[2] Paola Bertolazzi, Giuseppe Di Battista, Carlo Mannino, and Roberto Tamassia. Optimal upward planarity testing of single-source digraphs. *SIAM Journal on Computing*, 27(1):132–169, 1998.

[3] Bill Curtis, Marc I. Kellner, and Jim Over. Process modeling. *Communications of the ACM*, 35(9):75–90, 1992.

[4] Fred D. Davis. Perceived usefulness, perceived ease of use, and user acceptance of information technology. *MIS Quarterly*, 13(3):319–340, 1989.

[5] Nicolaas Govert de Bruijn. *Asymptotic Methods in Analysis*. Dover Publications, 1958.

[6] Juliane Dehnert and Wil M. P. van der Aalst. Bridging the gap between business models and workflow specifications. *International Journal of Cooperative Information Systems*, 13(3):289–332, 2004.

[7] Gilles Diguglielmo, Eric Durocher, Philippe Kaplan, Georg Sander, and Adrian Vasiliu. Graph layout for workflow applications with ILOG JViews. In Stephen G. Kobourov and Michael T. Goodrich, editors, *Revised Papers from the 10th International Symposium on Graph Drawing*, pages 362–363. Springer-Verlag, 2002.

[8] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *IEEE Transactions on Software Engineering*, 19(3):214–230, 1993.

[9] Michael R. Garey and David S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983.

[10] George M. Giaglis. A taxonomy of business process modeling and information systems modeling techniques. *International Journal of Flexible Manufacturing Systems*, 13(2):209–228, 2001.

[11] GraphViz, accessed 04/08/2013. `http://www.graphviz.org/`.

[12] Thomas Gschwind, Jakob Pinggera, Stefan Zugal, Hajo A. Reijers, and Barbara Weber. Edges, structures, and constraints: The layout of business process models. Technical Report RZ 3825, IBM Research, Zurich, 2011.

[13] Masud Hasan, Md. Saidur Rahman, and Takao Nishizeki. A linear algorithm for compact box-drawings of trees. *Networks*, 42(3):160–164, October 2003.

[14] John Hopcroft and Robert Tarjan. Efficient planarity testing. *Journal of the ACM*, 21(4):549–568, 1974.

[15] Michael D. Hutton and Anna Lubiw. Upward planar drawing of single source acyclic digraphs. In Allok Aggarwal, editor, *Proceedings of the second annual ACM-SIAM symposium on Discrete algorithms*, pages 203–211, 1991.

[16] IBM. WebSphere Business Modeler, accessed 03/14/2011. `http://www.ibm.com/software/integration/wbimodeler`.

[17] A. B. Kahn. Topological sorting of large networks. *Communications of the ACM*, 5(11):558–562, 1962.

[18] Vijay Khatri, Iris Vessey, Paul Clay V. Ramesh, and Sung-Jin Park. Understanding conceptual schemas: Exploring the role of application and IS domain knowledge. *Information Systems Research*, 17(1):81–99, March 2006.

[19] Ingo Kitzmann, Christoph König, Daniel Lübke, and Leif Singer. A simple algorithm for automatic layout of bpmn processes. In Birgit Hofreiter and Hannes Werthner, editors, *Proceedings of the 2009 IEEE Conference on Commerce and Enterprise Computing*, pages 391–398, 2009.

[20] Agnes Koschmider, Minseok Song, and Hajo A. Reijers. Social software for business process modeling. *Journal on Information Technology*, 25(3):308–322, 2010.

[21] J. Mendling, H. A. Reijers, and J. Cardoso. What makes process models understandable? In *Proceedings of the 5th international conference on Business process management*, pages 48–63. Springer, 2007.

[22] Marian Petre. Why looking isn't always seeing: readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, 1995.

[23] Jakob Pinggera, Stefan Zugal, and Barbara Weber. Investigating the process of process modeling with cheetah experimental platform. In *Proceedings of Empirical Research in Process-Oriented Information Systems (ER-POIS 2010)*, pages 13–18, 2010.

[24] Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In Giuseppe Di Battista, editor, *Proceedings of the 5th International Symposium on Graph Drawing*, pages 248–261, 1997.

[25] Helen C. Purchase, David A. Carrington, and Jo-Anne Allder. Empirical evaluation of aesthetics-based graph layout. *Empirical Software Engineering*, 7(3):233–255, 2002.

[26] Hajo A. Reijers and Jan Mendling. A study into the factors that influence the understandability of business process models. *IEEE Transactions on Systems, Man and Cybernetics, Part A*, 41(3):449–462, 2010.

[27] Marcello La Rosa, Arthur H. M. ter Hofstede, Petia Wohed, Hajo A. Reijers, Jan Mendling, and Wil M. P. van der Aalst. Managing process model complexity via concrete syntax modifications. *IEEE Transactions on Industrial Informatics*, 7(2):255–265, 2011.

[28] August-Wilhelm Scheer. *ARIS—Business Process Modeling*. Springer-Verlag, 2000.

[29] Matthias Schrepfer, Johannes Wolf, Jan Mendling, and Hajo A. Reijers. The impact of secondary notation on process model understanding. In Anne Persson and Janis Stirna, editors, *Proceedings of 2nd Working Conference on The Practice of Enterprise Modeling*, pages 161–175, 2009.

[30] Benno Stein and Frank Benteler. On the Generalized Box-Drawing of Trees: Survey and New Technology. In *Proc. I-KNOW '07*, pages 408–415, September 2007.

[31] Wil M. P. van der Aalst, Arthur H. M. ter Hofstede, Bartek Kiepuszewski, and Alistair P. Barros. Workflow patterns. *Distributed and Parallel Databases*, 14(1):5–51, 2003.

[32] Jussi Vanhatalo, Hagen Völzer, and Jana Koehler. The Refined Process Structure Tree. *DKE*, 68(9):793–818, 2009.

[33] Jussi Vanhatalo, Hagen Völzer, and Frank Leymann. Faster and More Focused Control-Flow Analysis for Business Process Models Through SESE Decomposition. In *Proc. ICSOC '07*, pages 43–55, 2007.

[34] John Warfield. Crossing theory and hierarchy mapping. *IEEE Transactions on Systems, Man, and Cybernetics*, 7(7):505–523, 1977.

[35] yWorks. yFiles for Java, accessed 03/14/2011.
`http://www.yworks.com/en/products\_yfiles\_about.html`.