Drawing Layered Graphs with Port Constraints

Christoph Daniel Schulze*, Miro Spönemann, Reinhard von Hanxleden

Department of Computer Science, Christian-Albrechts-Universität zu Kiel, Olshausenstr. 40, 24098 Kiel, Germany

Abstract

Complex software systems are often modeled using *data flow diagrams*, in which nodes are connected to each other through dedicated connection points called *ports*. The influence a layout algorithm has on the placement of ports is determined by *port constraints* defined on the corresponding node.

In this paper we present approaches for integrating port constraints into the *layer-based approach* to graph drawing pioneered by Sugiyama et al. We show how our layout algorithm, called KLay Layered, progresses from relaxed to more restricted port constraint levels as it executes, and how established algorithms for crossing minimization and edge routing can be extended to support port constraints. Compared to the previous layout algorithms supporting ports, our algorithm produces fewer edge crossings and bends and yields pleasing results.

We also explain and evaluate how layout algorithms can be kept simple by using the concept of *intermediate processors* to structure them in a modular way. A case study integrating our layout algorithm into UC Berkeley's Ptolemy tool illustrates how KLay Layered can be integrated into Java-based applications.

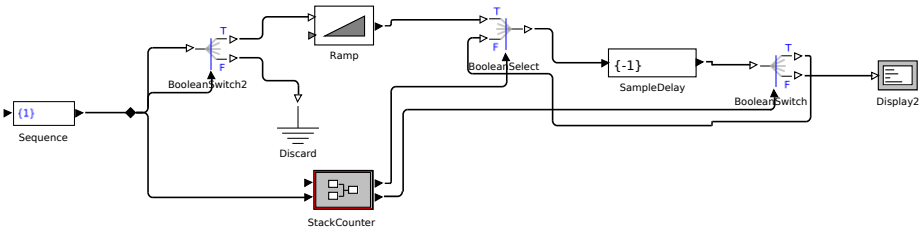
Keywords: graph drawing, crossing minimization, port constraints, layered graphs, data flow diagrams

1. Introduction

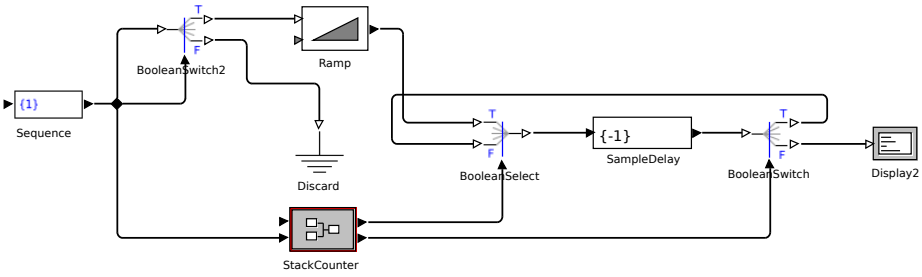
With up to ten million lines of code, software-based functions account for 50–70% of the effort in the development of automotive electronic control units [3]. To keep up with the growing complexity and tightening time-to-market requirements, embedded software domains such as the automotive, rail, or aerospace industries increasingly take advantage of graphical model-based development tools that follow the *actor-oriented* approach [19] such as Simulink (The MathWorks), SCADE (Esterel Technologies), ASCET (ETAS), or Ptolemy (UC Berkeley). Herein, graphical diagrams are used as input representations for simulators,

*Corresponding author

Email addresses: cds@informatik.uni-kiel.de (Christoph Daniel Schulze), msp@informatik.uni-kiel.de (Miro Spönemann), rvh@informatik.uni-kiel.de (Reinhard von Hanxleden)



(a) Layout using a previous approach [25] (3 edge crossings, 30 edge bends)



(b) Layout using the method presented here (1 edge crossing, 14 edge bends)

Figure 1: A Ptolemy model representing a stack (example by Edward A. Lee)

rapid prototyping systems, and code generators. Figure 1 shows a typical data flow diagram from Ptolemy and reveals the basic components of such a diagram, namely *actors* (also called *blocks* or *operators*), *connections* between the actors, and *ports* that define the interface of actors and the kind of data that is transported by connections.

While it is generally assumed that graphical diagrams are more readable than textual programs, the readability of a diagram strongly depends on its layout. Therefore, when creating or changing a model, an estimated 25% of a user's time is spent on manual layout adjustments according to Klauske [16]. Using graph layout algorithms to automate layout adjustments, this time can be better spent improving the model itself instead of its graphical representation.

Since in data flow diagrams actors are connected through ports, graph layout algorithms have to provide support for them. The position of a port relative to its actor need not even be fixed; instead, the layout algorithm can be made responsible for finding a good position. The degree of freedom it has in doing so is given by *port constraints* placed on the actor. Traditional layout algorithms support these requirements only in very limited ways, if at all.

Contributions. In this paper, we address the automatic layout of graphs with port constraints, building on the layer-based approach that has already proven successful in many domains but so far has received little attention with respect to port constraints. We introduce a structured way for transitioning from relaxed to restricted port constraint levels during the algorithm. We describe how the

popular barycenter-based crossing minimization heuristic can be extended to support ports and compare different methods for doing so. We also introduce methods for sorting ports where their order is not fixed. We describe how inverted ports can be handled that require connected edges to be routed around a node. For handling north/south-side ports, we introduce layout units and a new flexible method for dummy node creation. We also present the concept of intermediate processors to structure layout algorithms in a modular way, including an account of our practical experience with this concept.

As explained above, the development of the methods presented here was primarily motivated by the desire for automatic layout of data flow diagrams as used in embedded software development. However, our results should be applicable to other application areas as well, such as electronic circuit diagrams or any other graph representation with port constraints.

Outline. The rest of this section defines the necessary notation, introduces the layer-based approach to graph drawing, and gives an overview of related work in this area. Section 2 describes a clean approach for handling port constraint levels in a layout algorithm. Since port constraints have large implications for crossing minimization and edge routing, the necessary modifications are described in depth in Sections 3 and 4. Shifting towards a more practical perspective, we give details on the algorithm's modular architecture and a case study describing its integration into an existing software product in Section 5. We finish with a detailed evaluation of the algorithm in Section 6 and conclude with Section 7.

1.1. Definitions

Let (V, E) be a directed graph and $v \in V$. We denote the set of incoming edges of v by $E_\mathrm{i}(v)$ and the set of outgoing edges of v by $E_\mathrm{o}(v)$. The source node of an edge $e = (u, u')$ is $v_\mathrm{s}(e) = u$, and its target node is $v_\mathrm{t}(e) = u'$. In a *port-based graph* each edge is connected to its source and target over specific ports. The set of ports attached to v is denoted by $P(v)$. An edge e has a source port $p_\mathrm{s}(e) \in P(v_\mathrm{s}(e))$ and a target port $p_\mathrm{t}(e) \in P(v_\mathrm{t}(e))$. For a port $p \in P(v)$ we write $E_\mathrm{i}(p)$ for the subset of incoming edges of v that have p as their target port, and $E_\mathrm{o}(p)$ for the subset of outgoing edges of v that have p as their source port. Furthermore, we define $E(p) = E_\mathrm{i}(p) \cup E_\mathrm{o}(p)$.

The *drawing* of a port-based graph maps each node $v \in V$ to an area bounded by a rectangle with its upper left corner at $\mathrm{pos}(v) = (x, y) \in \mathbb{R}^2$ and its bottom right corner at $(x + w, y + h)$ with $w, h > 0$. Likewise, each port $p \in P(v)$ is mapped to a position $\mathrm{pos}(p) \in \mathbb{R}^2$, but that position is constrained to the boundary of the bounding rectangle of v. As a consequence, p can be assigned to a side $s \in \{\mathsf{north}, \mathsf{east}, \mathsf{south}, \mathsf{west}\}$ depending on which side of the bounding rectangle of v is chosen. The vertical axis is assumed to point downwards.

Edges are drawn with series of line segments that are each determined by a start point, an end point, and a sequence of bend points. Usually *orthogonal routing* is applied to port-based graphs: all line segments are aligned either horizontally or vertically.

During its execution, an algorithm may add and remove additional nodes. We distinguish such additional nodes from the nodes originally present in the graph and call them *dummy nodes* and *regular nodes*, respectively.

Many modeling applications that use ports do not allow graph drawing algorithms to modify the positions of ports relative to the node they are attached to, i. e. $\text{pos}(p) - \text{pos}(v)$ must be constant for each $p \in P(v)$. Other applications allow repositioning ports within certain constraints. The port constraints model used here assigns a constraint level $\text{PC}(v)$ to each node v, which can be chosen from five different levels:

FREE Ports can be placed at arbitrary positions on the boundary of node v.

FIXEDSIDE A side of v, denoted by $\text{side}(p) \in \{\mathsf{north}, \mathsf{east}, \mathsf{south}, \mathsf{west}\}$, is assigned to each port $p \in P(v)$.

FIXEDORDER The side of each port is fixed, and the ports of each side have to be placed in a specific order.

FIXEDRATIO Each port is assigned a position, relative to the containing node v; if v is resized, the position is scaled accordingly.

FIXEDPOS Each port is assigned a relative position that must not be modified by the layout algorithm.

1.2. Drawing Graphs with the Layer-Based Approach

The *layer-based* graph layout approach was introduced by Sugiyama et al. [26]. Given a directed acyclic graph, this approach arranges all edges in the same direction by organizing the nodes in subsequent *layers*, which are also called *hierarchies* or *levels* in graph drawing literature. All algorithms are described with the assumption that the main orientation of edges is from left to right, hence the nodes of each layer are arranged vertically. While this is inconsistent with most of the graph drawing literature, which assumes top-to-bottom layout [10, 26], the left-to-right layout is commonly used for data flow diagrams.

The layer-based approach solves the graph layout problem by dividing it into five consecutive phases.

1. *Elimination of Cycles.* Directed cycles can be eliminated by reversing a subset of the edges. The aim is to minimize this subset in order to have as many edges as possible pointing in the same direction in the final drawing. A popular algorithm for this problem is the heuristic of Eades et al. [4].
2. *Layer Assignment.* Nodes are assigned to layers $L_1, \ldots, L_k$ such that all edges point from layers of lower index to layers of higher index. Edges that span more than one layer (*long edges*) are split with dummy nodes in order to obtain a *proper layering*, where all edges connect nodes placed in consecutive layers. The number of required dummy nodes can be minimized efficiently with the algorithm of Gansner et al. [10].

3. *Crossing Minimization.* The nodes of each layer L_i are ordered with the goal of minimizing the number of edge crossings that can occur between pairs of layers. The *barycenter* heuristic, proposed by Sugiyama et al. [26], is very simple, fast, and gives reasonably good results [15].
4. *Node Placement.* The nodes of each layer L_i are assigned vertical positions according to the order determined in the previous step. Positions should be chosen such that the edges can be drawn as straight as possible, e. g. with the method of Sander [21] or the method of Brandes and Köpf [2].
5. *Edge Routing.* This final phase adds bend points to edges, depending on the desired routing style. Methods for orthogonal routing that also support hyperedges (edges connecting multiple ports) have been proposed by Eschbach et al. [7] and Sander [22].

The barycenter heuristic for crossing minimization is an important basis for the extensions proposed in this paper. It reduces the ordering problem to consider only two layers L_a and L_b at a time, where L_a is kept fixed and L_b is reordered. In the following, we assume that L_a is left of L_b in the ordering of layers; thus we consider only the set E_i of incoming edges of the nodes in L_b, but the symmetric case can be computed in the same way by considering the set E_o of outgoing edges of the nodes in L_a. For each node $v \in L_a$, let $r(v)$ be its position in the fixed order of L_a. The *barycenter value* for each node $w \in L_b$ is defined by

$$b(w) = \frac{1}{|E_\mathrm{i}(w)|} \sum_{(v,w) \in E_\mathrm{i}(w)} r(v) \ , \tag{1}$$

that is the average position of the connected nodes in L_a. The order of L_b is obtained by sorting the contained nodes by their barycenter values. The barycenter method can be used to order all layers using the *layer sweep* algorithm:

1. Determine a random order for the nodes of L_1.
2. Repeat for $i = 2, \ldots, k$: Reorder the nodes of L_i to reduce the number of crossings of edges with their source in L_{i-1} and their target in L_i.
3. Repeat for $i = k-1, \ldots, 1$: Reorder the nodes of L_i to reduce the number of crossings of edges with their source in L_i and their target in L_{i+1}.
4. Repeat steps 2 and 3 until the total number of crossings is not further reduced.

Generally the result of the layer sweep algorithm can be improved by repeating it with different random initial orderings, and then selecting the result that produced the least number of edge crossings. This selection process as well as step 4 of the layer sweep algorithm require a method for counting the number of crossings that result from a given layering. There are simple and efficient algorithms for counting crossings, e. g. the algorithm of Barth et al. [1].

More details on layer-based layout are reported by Healy and Nikolov [14].

1.3. Related Work

The first contributions to the problem of integrating port constraints in the layer-based approach were motivated by the layout of data structures, where

certain fields of a structure may contain pointers to other structures. Gansner et al. showed how node positioning can be extended for including offsets derived from port positions [10]. Sander introduced the idea of handling side ports by adding dummy nodes in order to route the respective edges [21]. The problem of crossing minimization with port constraints was first discussed by Waddle, who adapted the barycenter heuristic to consider port positions [28]. These contributions employ FixedPos constraints with spline curve edge routing, but they do not support inverted ports or other port constraints and are not sufficient for the layout of data flow diagrams. Basic approaches for handling multiple levels of port constraints were proposed by Spönemann et al. [25].

Schreiber proposed different solutions in the context of drawing bio-chemical networks [23]. The crossing minimization phase is adapted by inserting dummy nodes for each port and adding constraints to respect the order of ports. Side ports are handled by routing the incident edges locally for each node, which is done through transformation into a two-layer crossing minimization problem. This suffices for treating FixedPos constraints, but can lead to unpleasant layouts since the number of resulting bend points is possibly higher than necessary (see Section 4). The approach of Siebenhaller suffers from the same problem because it also routes edges of side ports locally [24]. However, it supports more flexible port constraints by associating them with individual edges. The consequence is that a node may have some edges that are constrained to ports, and some that are not. This flexibility can be useful for the layout of UML diagrams, where it is possible that only a subset of the edges is connected to fixed points of a node. The crossing minimization problem that results from this additional degree of freedom can be partly solved by reducing it to a network flow problem. The extension of our approaches to consider such mixed constraints is not required for most data flow languages and is left for future research.

Klauske and Dziobek introduced a specialized node placement for FixedRatio constraints [16, 17] in the context of the Simulink modeling language. Their approach is an extension of the linear program for node placement proposed by Gansner et al. [10]. This extension does not only determine vertical positions for the nodes, but also modifies their height in order to find an optimal placement of ports.

Gutwenger et al. introduced the concept of *embedding constraints* for modeling the port constraints of a node [12], but that model captures only the order of ports, and not their concrete positions. Such constraints are used in the context of *planarization* based layout, e. g. the *topology-shape-metrics* approach [27]. Harrigan and Healy applied embedding constraints to *level planarization* [13], which consists in finding a planar subgraph respecting a given layer assignment.

Klauske et al. proposed a new approach in which edges connected to north/south-side ports are routed through dummy nodes that are allowed to be interleaved with dummy nodes of long edges, giving more freedom to the edge routing [18]. However, the order of the dummy nodes is fixed and minimizes crossings only locally. This approach is compared with a new approach that does not fix the order of the dummy nodes in this paper. For handling inverted ports, Klauske et al. also introduced the concept of *in-layer edges* that connect

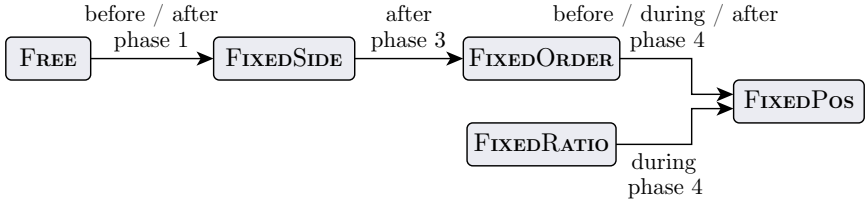


Figure 2: Overview of port constraint level transitions in the five phases of the layer-based layout approach. Nodes can start at all constraint levels depending on user preferences or requirements imposed by the diagram language, but will always be transitioned to FixedPos once the layout algorithm has finished.

two nodes in the same layer, and described the algorithmic modifications necessary to support them. Finally, Klause et al. introduced a way for structuring layout algorithms in a modular way, allowing them to dynamically adapt to different layout tasks. We follow this up in this paper with an account of our practical experience with this concept after having made extensive use of it.

2. Handling Port Constraint Levels

Let $v \in V$ be a node and $\mathrm{PC}(v)$ be the port constraint level of v. The basic principle for handling this constraint level is to lift it to stricter values as we progress through of the five phases of the layer-based approach. One important goal is to modify the algorithms employed in the five phases as little as possible, and to realize most of the extensions in additional preprocessing or postprocessing algorithms (see Section 5.1). This allows a modular implementation with a clear separation of concerns and helps to tame the complexity of the problems related to port constraints. In this section, we examine the transitions of constraint levels, of which an overview is given in Figure 2.

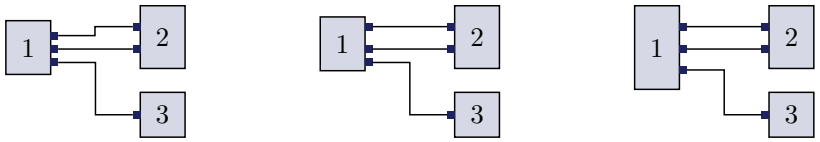
Free → Fixed Side. If $\mathrm{PC}(v) = \textsc{Free}$, all ports can be aligned to the main layout direction of the edges, which we assume to be left-to-right throughout this paper. This means that ports with incoming edges can be placed on the west side of v, while ports with outgoing edges can be placed on the east side. However, in the rare case that a port p has both incoming and outgoing edges, a compromise has to be found by comparing the number of edges:

- If $|E_\mathrm{i}(p)| - |E_\mathrm{o}(p)| > 0$, assign the side $s(p) = \mathsf{west}$.
- If $|E_\mathrm{i}(p)| - |E_\mathrm{o}(p)| < 0$, assign the side $s(p) = \mathsf{east}$.
- If $|E_\mathrm{i}(p)| - |E_\mathrm{o}(p)| = 0$, choose an arbitrary side.

The transition to fixed sides must be done before the crossing minimization phase since the decision which side to assign to each port has an impact on the number of edge crossings. If the transition is done before the cycle elimination phase, the edges incident to each node are consistently attached west or

(a) Before cycle elimination (b) After cycle elimination

Figure 3: Transition from FREE port constraints to FIXEDSIDE before or after the cycle elimination phase.



(a) Even distribution (b) Local adjustment (c) Scaling of node size

Figure 4: Different methods for port placement: (a) even distribution on the node sides, (b) local adjustments to eliminate edge bends, and (c) scaling of node height with FIXEDRATIO port constraints.

east depending on whether they are incoming or outgoing (see Figure 3a). In contrast, if the transition is done after cycle elimination, some edges may be reversed, hence they are attached to the opposite sides (see Figure 3b). The variant shown in Figure 3a emphasizes more clearly the flow represented by the edges, but the variant in Figure 3b is more compact. Therefore both options are valid and should be available in an implementation.

Fixed Side → Fixed Order. If $\mathrm{PC}(v) = \textsc{FixedSide}$ after the nodes of each layer have been ordered, it is necessary to order the ports on each side of v such that the number of crossings is minimized. Since v can contain arbitrarily many ports and each port can have arbitrarily many incident edges, the port ordering problem is equivalent to the two-layer crossing minimization problem where one layer is free and the other is fixed. As a consequence, ordering the ports of a node optimally is NP-hard [11], but reasonably good solutions can be found with an adapted version of the barycenter heuristic for two-layer crossing minimization. The adapted heuristic is described in Section 3. If dummy nodes are used to route edges connected to certain node sides, the order of these dummy nodes must be considered when sorting ports, which is described in Section 4.

Fixed Order → Fixed Position. If $\mathrm{PC}(v) = \textsc{FixedOrder}$, the final constraint level transition consists in setting concrete coordinates for each port. A straightforward method for this is to distribute the ports evenly on the boundary of v before the node placement phase (phase 4 of the layer-based approach). Then the node placement algorithm is responsible for considering these relative port coordinates in order to straighten the edges as much as possible.

Fixed Ratio → Fixed Position. The case $\mathrm{PC}(v) = \textsc{FixedRatio}$ only makes sense if the layout algorithm is allowed to modify the size of v, which is an extension to the layout problem that is outside the scope of this paper. It is particularly helpful for modeling languages/tools where the port position is derived from node sizes, as is, for example, the case in Simulink. Let p be a port with initial position (x, y) relative to v. If the height of v is scaled by a factor λ_h and p is on the east or west side, the vertical position of p is scaled accordingly to $y' = \lambda_\mathrm{h} y$. This behavior can be exploited for minimizing the number of edge bends: in Figure 4c, the height of node 1 is increased such that the distance between the two topmost ports equals the port distance of node 2, eliminating the bends of the connecting edges that are seen in Figure 4a. Node sizes can be optimized for edge straightening using the node placement method of Klauske and Dziobek [16, 17].

3. Crossing Minimization with Port Constraints

Extending the crossing minimization phase to consider port constraints is crucial since the number of crossings does not depend only on the order of nodes, but also on the order of ports for each node. We show how to modify the barycenter heuristic used in the layer sweep algorithm [26] such that it includes the port order in its calculations.

Let L_a be the fixed layer and L_b be the free layer to be reordered. For each node $v \in L_a$, let $P'(v) \subseteq P(v)$ be the subset of ports that have connections to nodes in L_b. For each port $p \in P'(v)$, we require a *rank* value $r(p)$ that is defined in Section 3.1. We redefine the barycenter value of a node $w \in L_b$ by

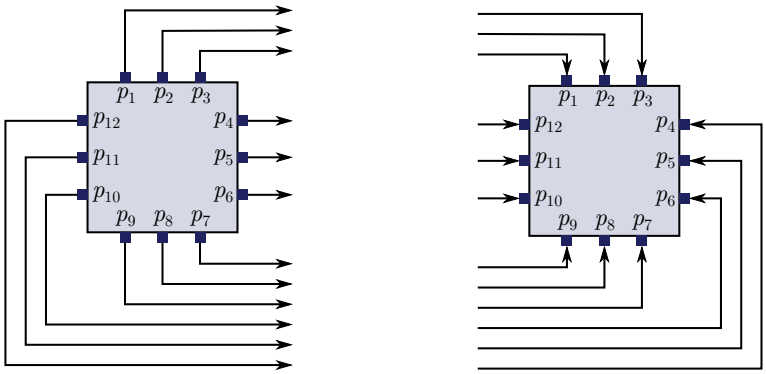
$$b(w) = \frac{1}{|E_\mathrm{i}(w)|} \sum_{e \in E_\mathrm{i}(w)} r(p_\mathrm{s}(e)) \ . \tag{2}$$

Here the source port $p_\mathrm{s}(e)$ of incoming edges e is considered instead of the source node. Outgoing edges can be processed symmetrically by using the target port $p_\mathrm{t}(e)$ and the set of outgoing edges $E_\mathrm{o}(w)$ in the calculation above, which is necessary for backwards sweeps of the layer sweep algorithm. The resulting barycenter values are used for sorting the nodes of L_b in the same way as for graphs without ports.

3.1. Port Ranking

The remaining question is how to determine the rank $r(p)$ for each port p in L_a. Waddle proposed to use the actual coordinates of the ports [28], but that can only be done if the port constraints are set to $\textsc{FixedPos}$ since for the other constraint levels the port coordinates are set after the crossing minimization phase has finished. Another option is to determine an index of p considering all other ports in the layer L_a, which we call the *layer-total* approach.

Let $v_i \in L_a = \{v_1, \ldots, v_{|L_a|}\}$ be a node in the ordered layer. At first we assume all port orders to be fixed, so let $p_j \in P'(v_i) = \{p_1, \ldots, p_{|P'(v_i)|}\}$ be a



(a) Outgoing edges (b) Incoming edges

Figure 5: Edge order for outgoing edges (for forward layer sweeps) and incoming edges (for backwards layer sweeps). Taking a set of ports indexed in clockwise order, outgoing edges are ordered clockwise starting with the leftmost north-side port, while incoming edges are ordered counter-clockwise starting with the rightmost north-side port. For inverted ports, i. e. west-side ports in (a) and east-side ports in (b), two alternative routings are feasible, either above or below the node, but this cannot be decided locally (see Section 4).

port in the ordered port set of v_i. As a convention, we assume this order to be clockwise, starting with the leftmost port on the north side of v_i. As shown in Figure 5a, this convention corresponds to the expected order of outgoing edges of v_i, hence it can be applied to forward layer sweeps. Backwards sweeps are based on the incoming edges and require a different order, namely counter-clockwise starting with the rightmost port on the north side (see Figure 5b). Let g be the greatest index such that $p_1, \ldots, p_g$ are all assigned to the north side of v_i. Then the edge order is induced by $o(p_j) = j$ for forward layer sweeps, $o(p_j) = g - j + 1$ for backwards layer sweeps if $j \leq g$, and $o(p_j) = g + |P'(v_i)| - j + 1$ otherwise. Let $\text{range}(v_k) = |P'(v_k)|$ be the range of port ranks occupied by a node $v_k \in L_a$ with fixed port order. We define the layer-total rank of p_j as

$$r_{\text{LT}}(p_j) = \left(\sum_{k<i} \text{range}(v_k) \right) + o(p_j) \ , \tag{3}$$

which gives unique integer rank values for all ports in L_a. With this kind of ranking, nodes with many ports occupy a greater range of ranks than nodes with few ports.

An alternative is to assign each node an equal range of 1, which we call the *node-relative* approach:

$$r_{\text{NR}}(p_j) = i + \frac{o(p_j)}{\text{range}(v_i) + 1} \ , \tag{4}$$

where again i is the node index and j is the port index.

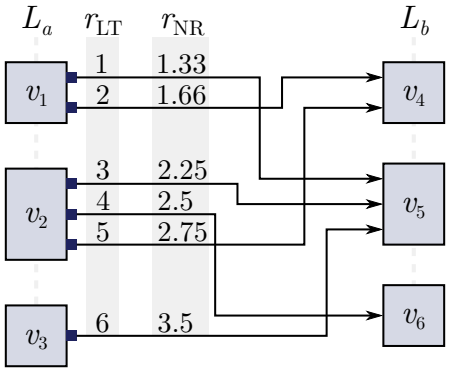


Figure 6: Ranks computed with the layer-total (r_{LT}) and the node-relative (r_{NR}) methods. The barycenter values for the layer-total ranks are $b(v_4) = 3.5$, $b(v_5) = 3.33$, and $b(v_6) = 4$, which results in the node order (v_5, v_4, v_6). In contrast, the node-relative ranks produce $b(v_4) = 2.21$, $b(v_5) = 2.36$, and $b(v_6) = 2.5$, resulting in the node order (v_4, v_5, v_6). With both variants the drawing has 5 edge crossings.

Both the layer-total and the node-relative ranking approach are very effective for crossing minimization with ports. See Section 6.2 for an evaluation of the two approaches. An example illustrating their operation is shown in Figure 6.

The two port ranking methods can be adapted to port constraint levels that do not imply a specific order of ports. The basic idea is to create groups of ports of which the order can be chosen freely, and to assign the same rank value to all members of a group. As described in Section 2, all nodes with constraint level FREE are set to FIXEDSIDE before or after phase one of the layer-based approach, hence we need to consider only the case FIXEDSIDE as alternative to the constraint levels with fixed port order. Since in our model each node can be assigned an individual port constraint, both rank calculation methods must be extended such that the rank of each port of a node v can be calculated differently than that of ports of other nodes. We denote the set of non-empty sides of v with S_v, i. e. $s \in S_v$ if there exists a port $p \in P'(v)$ such that $\mathrm{side}(p) = s$. Now we redefine the range of ranks occupied by v by

$$\mathrm{range}(v) = \begin{cases} |S_v| & \text{if } \mathrm{PC}(v) = \textsc{FixedSide}, \\ |P'(v)| & \text{otherwise.} \end{cases}$$

Given a node $v_i \in L_a$ for which $\mathrm{PC}(v_i) = \textsc{FixedSide}$, all ports $p \in P'(v_i)$ that are assigned to the same side of v_i are also given the same rank value. The four sides $\{s_1, s_2, s_3, s_4\}$ are ordered in the same way as already done for fixed-order constraints (see Figure 5). For forward layer sweeps, it is $s_1 = \mathsf{north}$, $s_2 = \mathsf{east}$, $s_3 = \mathsf{south}$, and $s_4 = \mathsf{west}$, while for backwards layer sweeps, it is $s_1 = \mathsf{north}$, $s_2 = \mathsf{west}$, $s_3 = \mathsf{south}$, and $s_4 = \mathsf{east}$. For each side s, let $\sigma(s) = 1$ if $s \in S_v$ and $\sigma(s) = 0$ otherwise. Furthermore, let $p \in P'(v_i)$ be a port and $\mathrm{side}(p) = s_j$ be

its assigned side. The edge order for FixedSide constraints is induced by

$$o(p) = \left(\sum_{k<j} \sigma(s_k) \right) + 1 \ .$$

By applying these new definitions of the range and edge order functions to Equations 3 and 4 we obtain new versions of the layer-total and node-relative ranking methods that assume an arbitrary order of ports on each side.

3.2. Counting Crossings

In order to effectively use the port-aware barycenter heuristic in the layer sweep algorithm for crossing minimization, we need a method for counting the number of crossings with proper consideration of port orders (see Section 1.2). In a properly layered graph, two edges can cross only if their source and target nodes are in the same layers. As a consequence, the total number of crossings can be determined as the sum of the crossings counted for each pair L_a, L_b of consecutive layers.

Let $\mathcal{A}$ be an algorithm for counting the number of crossings of edges connecting nodes in L_a with nodes in L_b. We can extend this to consider port constraints by replacing each node v with fixed port order by a set of nodes $v_1, \ldots, v_{|P(v)|}$ according to the ports $P(v) = \{p_1, \ldots, p_{|P(v)|}\}$. In a similar way, we replace each node v' with FixedSide constraint by nodes $v'_{\mathsf{n}}, v'_{\mathsf{e}}, v'_{\mathsf{s}}, v'_{\mathsf{w}}$ representing the groups of ports located on each of the four sides of v'. After this transformation we execute algorithm $\mathcal{A}$, which possibly results in a higher number of crossings as compared to the unmodified version. A good choice for $\mathcal{A}$ is the algorithm of Barth et al. [1].

3.3. Sorting Ports

Nodes for which the order of ports is not prescribed have to be processed after an ordering of all layers has been determined. The goal is to find an order of ports with minimal number of edge crossings. Siebenhaller presented an approach for ordering free edges at nodes that can also have fixed edges by transforming the problem into a flow network and finding a minimum cost flow [24, Section 4.5.1.2]. In our model of port constraints this mixed scenario is not allowed, thus the ports of a node are all subject to the same ordering constraint. According to Section 2 only the case FixedSide has to be considered for crossing minimization since for nodes with fixed port order the port ordering step is skipped, and the Free constraint level is processed earlier.

Let v be a node with FixedSide constraint. In order to find a suitable ordering of the ports of v we apply an adapted variant of the barycenter heuristic. Each port $p \in P(v)$ is assigned a barycenter value $b(p)$ defined by

$$b(p) = \frac{1}{|E(p)|} \left(\sum_{e \in E_{\mathrm{o}}(p)} r(p_{\mathrm{t}}(e)) \ - \sum_{e \in E_{\mathrm{i}}(p)} r(p_{\mathrm{s}}(e)) \right) \ . \tag{5}$$

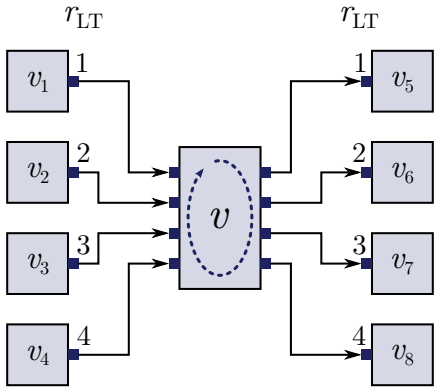


Figure 7: Ranks of ports connected by outgoing edges of v (right layer) conform to the clockwise order of ports around v. Ranks of ports connected by incoming edges (left layer), however, are contrary to that clockwise order.

Then the ports $P(v)$ are sorted with a primary and a secondary key: the primary key is the side assigned to each port, and the secondary key is the barycenter value. The ranks $r(p_{\mathrm{s}}(e))$ and $r(p_{\mathrm{t}}(e))$ are computed in the same way as previously described for the ordering of layers. Note that the barycenter calculation for ports considers ranks for incoming edges as well as outgoing edges. The ranks of incoming edges are considered with a negative sign because the order of the corresponding ports is contrary to the convention that the ports of v are ordered clockwise around v (see Figure 7). Usually a port has either incoming or outgoing edges, but not both, hence one of the two sums in Equation 5 is zero. Since fixing the order of $P(v)$ influences the ranks of these ports, care must be taken to properly update the rank values. The process of sorting ports for the whole graph is outlined in Algorithm 1.

Algorithm 1: Sorting ports

```
Input: a graph with layers L_1, ..., L_k
for i = 1...k do
    if i < k then
        Compute ranks of the ports in L_{i+1}
    for each v ∈ L_i do
        if PC(v) = FixedSide then
            for each p ∈ P(v) do
                Compute the barycenter b(p)
            Sort P(v) by assigned sides and barycenter values
            PC(v) ← FixedOrder
    // Rank values may now be different due to updated constraints.
    Recompute ranks of the ports in L_i
```

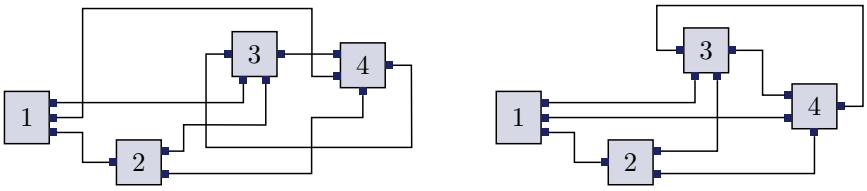
A problem with the approach of using the ranks of both the preceding and the subsequent layer in the barycenter calculation is that the rank values of the two layers are determined independently of each other, and thus it makes little sense to compare them with each other in the sorting algorithm. We solve this problem using preprocessing techniques, as a consequence of which all outgoing edges of a node are incident to ports on the east side, and all incoming edges are incident to ports on the west side. This property ensures that only ranks of ports from the same layer are compared with each other by the sorting algorithm.

4. Edge Routing with Port Constraints

The way an edge incident to a node v should be routed depends on the side to which the edge is connected, and whether it is an incoming or an outgoing edge. We call a port $p \in P(v)$ *regular* if either $\text{side}(p) = \mathsf{east}$ and $E_\text{i}(p) = \emptyset$, or $\text{side}(p) = \mathsf{west}$ and $E_\text{o}(p) = \emptyset$. For instance, all ports in Figure 7 are regular and thus conform to the left-to-right orientation of edges. In contrast, we call p an *inverted* port if either $\text{side}(p) = \mathsf{east}$ and $E_\text{i}(p) \neq \emptyset$, or $\text{side}(p) = \mathsf{west}$ and $E_\text{o}(p) \neq \emptyset$. If all ports are regular, we can apply standard routing methods [7, 22]. If we have inverted ports or north/south-side ports, however, the standard methods are not sufficient and additional bend points are required (see Figure 5).

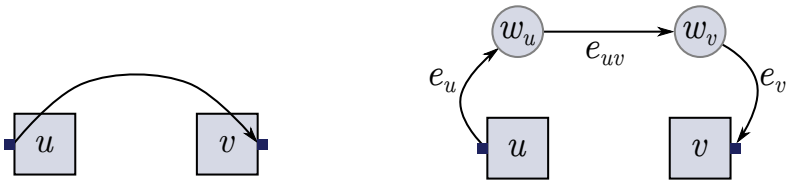
In previous contributions for handling north/south-side ports, the affected edges of a node v were replaced by dummy nodes constrained to be placed next to v. Hence, the edges are routed locally around v without allowing other edges to be routed in between [21, 23, 24, 25]. We call this the *local* approach for routing with port constraints. As the term suggests, the approach restricts the routing of edges to a specific area surrounding the node v and does not take the global graph structure into account. Such a local routing is also implied by the edge order function $o(p)$, $p \in P(v)$, defined in Section 3.1. Figure 5 reveals that, according to this order function, edges incident to inverted ports are always assumed to be routed below the node. This is not always a good choice since in some cases routing above the node would yield a more readable drawing. Furthermore, the local approach does not allow other nodes to be placed inside the reserved routing area surrounding v.

We propose a *global* routing approach which is based on the idea of representing edge segments by dummy nodes that can be placed with certain constraints during the crossing minimization phase [16, Section 3.3.4]. This approach allows a more flexible arrangement of edge segments in order to increase the freedom for minimizing edge crossings and bends. The graph shown in Figure 8 has 5 crossings and 17 bends when drawn with the local approach, but only 1 crossing and 11 bends when drawn with our global approach. This is mainly due to two properties of our global method that are exploited in the example: the feedback edge $(4, 3)$ is drawn above instead of below the nodes, and the edge $(1, 4)$ intersects the area between node 3 and the bend point of edge $(2, 3)$. In contrast, the local method first fixes the routing of the edges $(1, 3)$, $(2, 3)$, and



(a) Local: 5 crossings, 17 bends (b) Global: 1 crossing, 11 bends

Figure 8: Two approaches for routing edges subject to port constraints: (a) the *local* approach reserves an exclusive area around each node without regard to the structure of the graph, while (b) the *global* approach generates dummy nodes that can be placed with constraints that are less strict, and thus enables solutions with fewer edge crossings and bends.



(a) An edge with inverted ports (b) Dummy nodes inserted into (u, v)

Figure 9: An edge (u, v) connecting inverted ports is split with dummy nodes w_u and w_v. The new edges $e_u = (u, w_u)$ and $e_v = (w_v, v)$ are in-layer edges.

$(4, 3)$ locally around node 3 and thereby prevents edge $(1, 4)$ to run between the edges $(1, 3)$ and $(2, 3)$.

More details on the global routing approach are given in the remainder of this section.

4.1. Inverted Ports

The basic scheme for handling inverted ports is illustrated in Figure 9: given an edge $e = (u, v)$ for which the source port $p_s(e)$ is inverted, a dummy node w_u is inserted in the same layer as u, and e is split into $e_u = (u, w_u)$ and $e_{uv} = (w_u, v)$. If the target port $p_t(e)$ is inverted, a dummy node w_v is inserted in the same layer as v, and e is split into $e_{uv} = (u, w_v)$ and $e_v = (w_v, v)$. If both $p_s(e)$ and $p_t(e)$ are inverted, as shown in Figure 9b, the edge sequence replacing e is $e_u = (u, w_u)$, $e_{uv} = (w_u, w_v)$, and $e_v = (w_v, v)$. As a result of this preprocessing, the new edge e_{uv} can be treated as a regular edge. The edges e_u and e_v, however, have their source and target in the same layer, which breaks the general requirement of a proper layering introduced in Section 1.2. We call this new kind of edges *in-layer edges* and restrict them to only connect ports on the same side.

While in-layer edges can be ignored during node placement (phase 4), and while it is straightforward to include them in the orthogonal edge routing (phase 5), more intricate adaptions are necessary for crossing minimization (phase 3).

The complexity of these adaptions can be greatly reduced by exploiting the fact that for the processing of inverted ports either the source or the target node of an in-layer edge is a dummy node. Let v be a regular node connected to a dummy node w_v via an in-layer edge $e = (w_v, v)$, as shown in Figure 9b. We have to correct the barycenter calculation for v in case of a forward layer sweep with v and w_v both in the free layer L_b: the normal processing would include the rank $r(p_s(e))$ in the sum computed for $b(v)$ (see Equation 2), but that rank would be undefined because ranks are defined only for ports in the fixed layer L_a. The dummy node w_v, however, has only incoming edges with their source in L_a, hence the value $b(w_v)$ can be computed normally. The solution is to use $b(w_v)$ in place of $r(p_s(e))$ in the computation of $b(v)$. This can be written as

$$b(v) = \frac{1}{|E_{\mathrm{i}}(v)|} \left(\sum_{\substack{e \in E_{\mathrm{i}}(v), \\ v_{\mathrm{s}}(e) \in L_a}} r(p_{\mathrm{s}}(e)) \; + \sum_{\substack{e \in E_{\mathrm{i}}(v), \\ v_{\mathrm{s}}(e) \in L_b}} b(v_{\mathrm{s}}(e)) \right) . \tag{6}$$

In the example shown in Figure 9b this would mean that $b(v) = b(w_v)$ since (w_v, v) is the only edge incident to v. The barycenter values can be computed by first processing all dummy nodes of L_i, for which we ignore the in-layer edges, then all remaining nodes. The adaption of backwards layer sweeps is symmetric. As a result, dummy nodes created for inverted ports are placed near their corresponding regular nodes. After all five phases of the layout algorithm have finished, the dummy nodes are removed in the same way as those created to split long edges, thus restoring the original edges.

An additional extension is necessary regarding the barycenter calculation for sorting ports with FixedSide constraints defined in Equation 5. For regular ports this extension can be done in the same way as shown for the ordering of layers (Equation 6). For inverted ports, however, we need to obtain barycenter values in a different way. Let $p \in P(v)$ be an east-side port of v connected to a dummy node w via an in-layer edge. At the time when the ports of v are sorted, the relative position of v and w in their respective layer L is already determined, so let $i(v)$ and $i(w)$ be the indices of these nodes. If $i(w) < i(v)$, we want p to be placed at the top of the east side, and otherwise we want it at the bottom. A simple solution is to first compute the barycenters of all regular ports and determine their minimum $b_{\min}$ and maximum $b_{\max}$. Let i_{avg} be the average index of dummy nodes connected to p via in-layer edges (p may have more than one in-layer edge). The barycenter of p is

$$b(p) = \begin{cases} b_{\min} - i_{\mathrm{avg}} & \text{if } i_{\mathrm{avg}} < i(v), \\ b_{\max} + |L| + 1 - i_{\mathrm{avg}} & \text{otherwise.} \end{cases}$$

As can be seen in Figure 10, the effect of the negative sign of i_{avg} is that the order of ports is inverted with respect to the order of dummy nodes, but that is correct if crossings of in-layer edges are to be avoided. The handling of west-side inverted ports is symmetric.

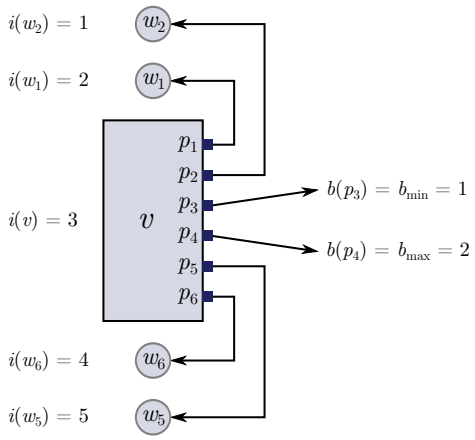


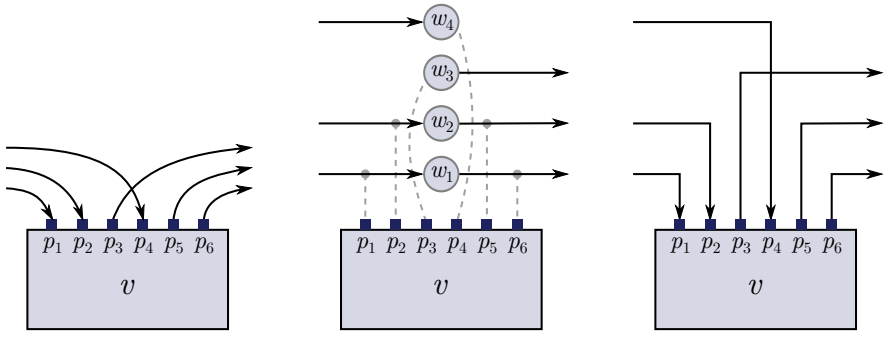
Figure 10: Sorting ports with in-layer edges: $b(p_1) = b_{\min} - i(w_1) = -1$, $b(p_2) = b_{\min} - i(w_2) = 0$, $b(p_5) = b_{\max} + |L| + 1 - i(w_5) = 3$, and $b(p_6) = b_{\max} + |L| + 1 - i(w_6) = 4$. Assuming $b(p_3) = 1$ and $b(p_4) = 2$ according to Equation 5, we obtain the port order as depicted above.

Finally, the algorithm for counting crossings of edges between consecutive layers must be complemented for counting the crossings caused by in-layer edges. With orthogonal edge routing, we cannot predict exactly how many crossings an in-layer edge will cause. We can, however, compute an upper bound in two passes over a layer's nodes. Remember that we have restricted in-layer edges to only connect ports on the same side. The first pass iterates over the ports on the eastern and on the western side separately from top to bottom. Each port is assigned a number that equals the maximum number assigned so far for ports on its side plus the number of edges incident to the port since those could cause crossings with in-layer edges. The second pass then iterates over the ports again looking for in-layer edges. The sum of the differences of port numbers of the ports connected by the edges constitutes the maximum number of crossings. Since the final number of crossings will usually be lower than this upper bound, future research could go into finding algorithms that give more exact estimates.

4.2. North/South-Side Ports

Ports on the north or south side of a node v are handled by adding dummy nodes in order to determine where to draw the necessary bend points. The general idea is illustrated in Figure 11 for north-side ports. Each dummy node is associated with either one or two ports. The edges are redirected to the generated dummy nodes, hence the node v does not have any connections to the north or south side after this preprocessing. After the five phases of the layer-based algorithm have finished, the original edges are restored and bend points are added at the vertical coordinates that have been assigned to the dummy nodes.

Let v be a node, let $V_{v,\mathrm{N}}$ be the sequence of dummy nodes generated for north-side ports of v, and let $V_{v,\mathrm{S}}$ be the sequence of dummy nodes generated



(a) North-side ports (b) Dummy nodes (c) Resulting drawing

Figure 11: Edges connected to the north side are redirected to dummy nodes in the fixed-order approach. In this example four nodes $w_1, \ldots, w_4$ are required, of which w_1 and w_2 are associated with two ports (indicated by the dashed lines in Fig. (b)). In the final drawing, the original edges are restored and the dummy nodes are replaced by bend points.

for south-side ports of v. There are two methods for creating the dummy nodes in $V_{v,\mathrm{N}}$ and $V_{v,\mathrm{S}}$: one method optimizes the number of edge crossings locally to v (*fixed-order* approach), the other takes surrounding layers into account (*variable-order* approach). In the following, we will assume that no north/south-side port of v has both, incoming and outgoing edges, which is usually the case. However, extending the methods to handle this case is straightforward.

Fixed-Order Approach. The first method locally optimizes the number of crossings caused by edges connected to north and south ports, without taking other nodes into account. We iterate over the north ports of a node, starting with the two outermost unprocessed ports p_1, p_2, working our way inwards. As Figure 11c shows, the bend points later inserted can share the same vertical coordinate as long as the left port p_1 has only incoming edges and the right port p_2 has only outgoing edges; we thus create one dummy node for each such pair of ports. Once we do not find such pairs anymore, we create one dummy node for each port we have not iterated over yet. South ports are processed similarly.

Note that the order of the dummy nodes in the layer is crucial for edge crossings to be locally minimized and thus has to be preserved by the crossing minimization phase. For each node u with $u \in L$ for some layer L, we introduce *node successor constraints* $S_u \subseteq L$ that define a possibly empty set of nodes that u must precede in L. When creating the dummy nodes for north and south ports, each dummy node is defined as the successor to the previously created dummy node. Additionally, the node v is the successor of the last node of $V_{v,\mathrm{N}}$ and the first node of $V_{v,\mathrm{S}}$ is the successor of v. Support for node successor constraints can be easily added to the crossing minimization phase. Once an ordering is computed, the method of Forster [8] can be used to find successor constraint violations and to resolve them.

For this approach to work, the order of ports on the north and south side

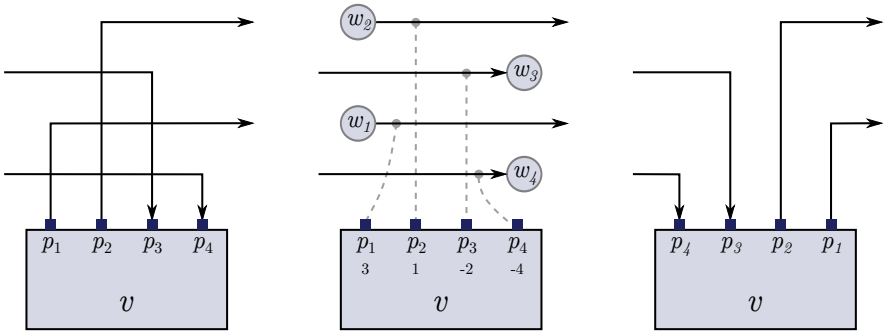
must already be fixed (hence the name *fixed-order* approach). If $\mathrm{PC}(v) = \textsc{FixedSide}$, the ports on the north and south sides of v have to be sorted before their respective dummy nodes are created. Since they are created before the crossing minimization phase, it is not possible to consider the global graph structure when sorting the ports. As a consequence, they can be sorted using only local information, that is the number of incoming and outgoing edges. Let $\Delta_E(p) = |E_\mathrm{o}(p)| - |E_\mathrm{i}(p)|$ for each port p, then ports with high Δ_E value should be placed towards the subsequent layer, while those with low Δ_E should be placed towards the preceding layer.

Variable-Order Approach. The previous approach fixes the order of dummy nodes and can thereby prevent the crossing minimization phase from finding a globally optimized ordering. A better alternative seems to be to relax the ordering constraints such that the dummy nodes $V_{v,\mathrm{N}}$ and $V_{v,\mathrm{S}}$ can be ordered arbitrarily. Instead of deriving the dummy node order from the port order, we first apply constrained crossing minimization and then derive the port order from the dummy node order. However, we still require constraints to ensure that v is placed between the nodes in $V_{v,\mathrm{N}}$ and $V_{v,\mathrm{S}}$.

Creating the dummy nodes is similar to the creation process in the fixed-order approach, with two exceptions. First, we do not insert successor constraints between dummy nodes. And second, we do not allow two ports to share a single dummy node since this would limit the freedom of both the crossing minimization and the node placement phase to find optimal results.

This approach can only be applied with success if the crossings between edges connected to north/south side ports are included in the total crossing number of a layer sweep (see for instance the edges connected to p_3 and p_4 in Figure 11). This enables the layer sweep algorithm to select the ordering for which the number of crossing is truly minimal. Counting the crossings for a given dummy node order and port order is straightforward: given two north-side edges e_1, e_2 with corresponding dummy nodes w_1, w_2 and ports p_1, p_2, the edge e_1 crosses the vertical segment of e_2 if w_1 is below w_2 and either e_1 is outgoing and p_1 is left of p_2, or e_1 is incoming and p_1 is right of p_2. If e_1, e_2 are on the south side, the condition is nearly the same, but w_1 must be above w_2. Checking these conditions for each pair of edges takes quadratically many computation steps depending on the number of edges on the north and south side of v, but usually that number is rather low.

In contrast to the fixed-order approach, this approach does not require the order of ports to be fixed beforehand. Instead, the order of ports can be easily inferred by the order in which their dummy nodes appear in the layer, with the goal of minimizing the number of crossings. To this end, we assign a barycenter value to each north-side port p with corresponding dummy node $w \in V_{v,\mathrm{N}}$: $b(p) = -i(w)$ if p has only incoming edges, $b(p) = i(w)$ if p has only outgoing edges, and $b(p) = 0$ otherwise, where $i(w)$ is the index of w in its containing layer. We treat dummy nodes in $V_{v,\mathrm{S}}$ similarly. The resulting barycenter values can be integrated in the sorting process described in Section 3.3, where the port side is the primary key and the port barycenter is the secondary key for sorting.



(a) Unsorted ports (b) Sorting the ports (c) Sorted ports

Figure 12: Edges connected to the north side are redirected to dummy nodes in the variable-order approach. In this example four nodes $w_1, \ldots, w_4$ are required, one for each port (indicated by the dashed lines in Fig. (b)). Since the crossing minimization determines the order of dummy nodes in this approach, the ports are sorted once the dummy node order is known.

Figure 12 shows an example of this method.

One advantage of the fixed-order approach is that vertical space can be saved by allowing edges connected to different ports to share the same dummy node. Since the variable-order approach requires each port to have a separate dummy node created for it, this would only be possible if the node placement phase was extended accordingly. Such extensions are left for future research. The fixed-order and the variable-order approaches are compared in Section 6.3.

Layout Units. The two approaches just described differ in how much they fix the relative order of created dummy nodes using node successor constraints. They are similar, however, in two other requirements we have not discussed thus far:

1. The dummy nodes created for a node v_1 must not be interleaved with the dummy nodes created for another node v_2. Otherwise, edges might overlap each other, resulting in ambiguity.
2. A regular node v_1 must not be placed among the dummy nodes created for another node v_2. Otherwise, edges connected to north-side or south-side ports of v_2 would cross v_1.

To meet these requirements, we propose the concept of *layout units*: nodes of one layout unit cannot be placed among nodes of another layout unit, thus keeping the different layout units separate from each other. The layout unit of a regular node v is $U_v = V_{v,\mathrm{N}} \cup \{v\} \cup V_{v,\mathrm{S}}$. Whenever a layer L is ordered during the execution of the layer sweep algorithm, new node ordering constraints are dynamically inserted and then resolved using Forster's method. Let $u, v \in L$ be regular nodes and let v be the next node following u in the given order of L. Node successor constraints are inserted from all nodes in U_u to all nodes in U_v in order to prevent overlaps between layout units.

Note that dummy nodes inserted to split long edges may well be placed among the nodes of a layout unit. While such a placement would cause edge crossings, it would not cause overlaps and would thus remain clear of ambiguity.

5. Implementation and Integration

The algorithmic concepts described so far are implemented in a Java-based layout algorithm called *KLay Layered*. Since many popular graph layout libraries such as Graphviz[1] and OGDF[2] are implemented in C, the KLay project is an effort to develop layout algorithms purely in Java. The algorithms are integrated into the *KIELER Infrastructure for Meta Layout* (KIML),[3] a framework based on Eclipse[4] which provides the glue between diagram viewers and layout algorithms. KIML defines an intermediate graph format called *KGraph* that is used to pass graphs to the layout algorithms as well as to return layout information to the diagram viewers. Connecting a diagram viewer to all layout algorithms provided through KIML thus becomes a matter of providing a function that produces a KGraph and one that applies the computed layout information back into the diagram. We already made such functions available to graphical editors based on the popular frameworks GMF[5] and Graphiti.[6] We have found that the GMF functions in particular work well without adaptations for editors that follow GMF standards.

The basic architecture of KLay Layered follows the five phases outlined in Section 1.2. One of the goals of its development is to keep the algorithm flexible and to allow the user to adapt it to different types of diagrams. This is done by providing different implementations for each phase, each geared towards different optimization goals or different types of layout. For example, there are three implementations of the edge routing phase: the first implements orthogonal edge routing as presented in this paper, the second implements edge routing using spline connections, and the third implements polyline edge routing, similar to the one originally proposed for the layered approach by Sugiyama et al. [26]. The actual implementation used is determined by the user, with the algorithmic architecture based on the well-known *strategy pattern* [9].

Providing different implementations of a phase also has its drawbacks, however. As shown in Figure 2, each implementation of the crossing minimization phase has to end by making sure that all nodes have their port constraints transitioned to at least FixedOrder. This is currently done the same way in all implementations and could thus be easily factored out, reducing code duplication. Another problem is supporting layout options across the algorithm.

[1] `http://www.graphviz.org/`
[2] `http://www.ogdf.net/`
[3] `http://www.informatik.uni-kiel.de/rtsys/kieler/`
[4] `http://www.eclipse.org/`
[5] `http://www.eclipse.org/modeling/gmp/`
[6] `http://www.eclipse.org/graphiti/`

Suppose that the layout algorithm has to be adaptable to different layout directions (e. g., left-to-right and top-to-bottom). Any parts of the algorithm that compute coordinates will differ only in details that depend on the chosen layout direction, leading to code littered with lengthy conditional statements with almost identical code. A better way to support this option would be to transform the layout problem into one that assumes a left-to-right layout direction and to transform the solution back to the original problem afterwards. Then, the bulk of the algorithm can simply assume a left-to-right layout direction, which in turn greatly reduces code complexity.

We extend the basic architecture of KLay Layered to solve these issues by introducing the concept of *intermediate processors*: modules that contain code factored out of different phase implementations, or code that implements pre- or post-processing to reduce complex layout problems to ones that the phase implementations can handle.

The remainder of this section is devoted to a more detailed explanation of intermediate processors and our experience with this concept. This is followed by a case study in which we integrated KLay Layered into Ptolemy, an application not based on Eclipse.

5.1. Intermediate Processors

Intermediate processors simplify the regular phases by factoring out shared functionality, or by reducing complex layout problems to simpler ones that can be handled by the regular phases. As already hinted at, the exact place where a processor fits differs from processor to processor. We thus extend KLay Layered's basic architecture by adding *intermediate processing slots* before, between, and after the algorithm's sequence of regular phases, each capable of holding an arbitrary number of intermediate processors.

The intermediate processing configuration of the algorithm is determined by the phase implementations chosen by the user. Before the algorithm's execution, we instantiate each phase implementation and query it for the intermediate processors it requires in the different processing slots. The returned configuration usually depends on the actual graph to be laid out: if there are no nodes with northern or southern ports, the corresponding intermediate processors can simply be left out. KLay Layered thus dynamically configures itself for each layout task, building a list of intermediate processors and phase implementations that form a concrete instance of the algorithm.

Since the algorithm configures its intermediate processors already dynamically, the obvious question is why we constrain this flexibility to predefined processing slots; one could imagine a completely dynamic algorithm that is free of a predefined structural skeleton. The reason for this limitation is that intermediate processors have dependencies on each other: an intermediate processor that computes the amount of space required by ports for each node may require an intermediate processor that computes the port positions to have already run. The number of possible dependencies increases quadratically with the number of intermediate processors. By defining discrete slots for intermediate processors

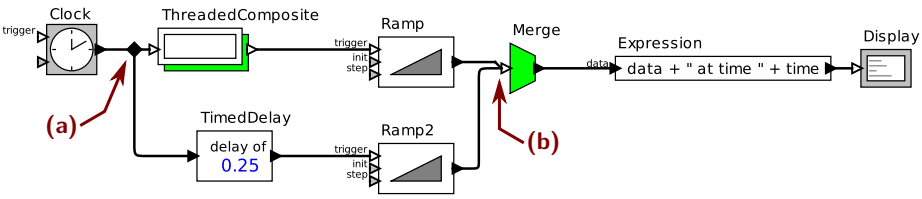


Figure 13: A Ptolemy model demonstrating parallel execution, laid out using the method presented in this paper (example by Edward A. Lee). This model exhibits two graphical features specific for Ptolemy models: relation vertices (a) and multiports (b).

to be placed in, we have to manage only the dependencies of processors that share the same slot.

Even so, dependencies from one processor to other processors must be carefully managed. This is done by judiciously specifying the preconditions and postconditions of each intermediate processor and defining the dependencies such that the preconditions of each processor are met.

Since the introduction of intermediate processors, KLay Layered has grown to include 35 of them. While we were worried at the beginning that managing the dependencies between the processors might become too complex, this has turned out not to be the case if proper care is taken defining the preconditions and postconditions of the processors. In fact, we were surprised at just how easy it is to add new features to the algorithm. For example, adding support for edge labels by using dummy nodes became a matter of adding a few simple intermediate processors to the algorithm, without having to change any phase implementation. The most obvious downside is that moving tasks to intermediate processors causes more iterations over the graph. For practically-sized diagrams, however, the performance impact this has on the algorithm has turned out to be negligible, as shown in Section 6. In our experience, the advantages in terms of easier implementation and well-structured code far outweigh the performance impact.

5.2. Ptolemy Integration

Ptolemy[7] is an open source modeling environment developed at UC Berkeley that targets the modeling and semantics of concurrent real-time systems [6]. Ptolemy models are actor-oriented data flow diagrams and can contain nested state machines (*modal models*). Since Ptolemy is a Java program not based on Eclipse, it serves as a case study for using the KLay Layered algorithm outside Eclipse. To that end, a standalone version of the algorithm is available that optionally includes all required external libraries.

While the majority of diagram viewers based on popular Eclipse technologies can readily use layout algorithms through KIML without further modification,

[7] `http://ptolemy.eecs.berkeley.edu/`

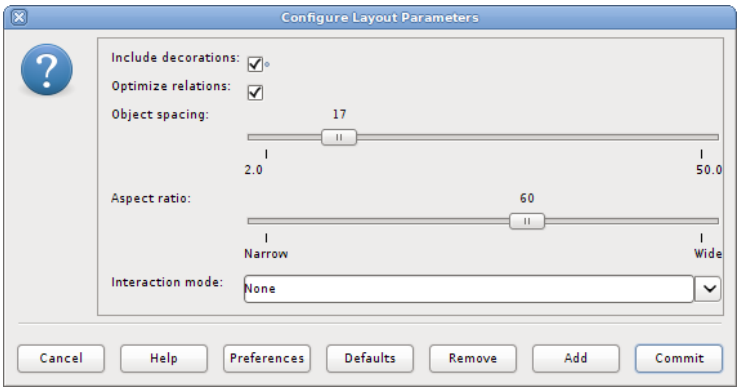


Figure 14: Layout preferences are integrated into Ptolemy using the Ptolemy UI conventions, with an emphasis on presenting the options in as understandable a way as possible.

viewers not based on Eclipse require some programming effort to integrate automatic layout. How complex this is strongly depends on the employed drawing library and on how well the graph model fits the KGraph format. Ptolemy posed a few challenges in these regards.

To start with, even though Ptolemy defines a data flow language that transfers tokens of data between ports, the flow of data can be ambiguous: ports can be input ports, output ports, or even both. More severely, edges in the underlying data model are undirected. This conflicts with both the KGraph model, which requires edges to be directed, and KLay Layered, which uses edge directions to compute a layering that emphasizes the flow of data. The transformation therefore has to rely on a heuristic to infer edge directions from port type information.

Ptolemy also distinguishes between ports with at most one connected edge and *multiports* that can have an arbitrary number of edges connected to them, as shown in Figure 13. Multiports are special in that their edges do not connect to the same point, but are in fact offset from each other, which is contrary to the way KLay Layered handles edges connected to the same port. The transformation works around this disparity by generating one port in the KGraph model for each edge connected to a multiport, with each port offset accordingly.

Another problem is caused by Ptolemy's *relation vertices*, as also illustrated in Figure 13. While edges can connect only two end points, it is often necessary to connect more than two actors to each other, for example to feed the output of one actor to several other actors. This is done in Ptolemy by introducing relation vertices to the model: all ports reachable through the same set of relation vertices are semantically treated as being directly connected to each other. Without special consideration, relation vertices would be placed in layers alongside regular nodes by the layout algorithm, which is aesthetically not very pleasing and also often a waste of space. Instead, users would expect relation vertices to be placed on *junction points*, that is, on points where two edges

branch off from each other. KLay Layered computes junction points while routing edges orthogonally, and ongoing work is concerned with ensuring that relation nodes are placed on these points.

A final challenge was how to expose the rather large number of often very specialized layout options supported by KLay Layered in the user interface. If not filtered properly, these for example include options such as the edge routing algorithm to be used, which can well be inferred automatically based on the kind of Ptolemy model to be laid out: data flow diagrams use orthogonal edge routing while modal models use spline-like edge routing. As shown in Figure 14, we decided to hide the vast majority of options: most options can simply be programmatically set to sensible defaults or be dynamically inferred, depending on the kind of model to be laid out. The few remaining options are presented using terms that are as intuitive as possible to users of Ptolemy, and are integrated following Ptolemy's established user interface standards.

In conclusion, we have found that the effort required to integrate automatic layout into a diagram viewer depends on how different the models to be laid out are from the KGraph format as well as on the assumptions made by the layout algorithm.

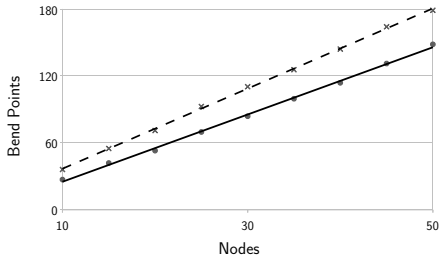
6. Experimental Evaluation

The quality of layouts is usually measured through *aesthetics criteria*, of which the number of edge crossings and the number of bend points rank among the most important according to Purchase et al. [20, 29]. In this section, we evaluate KLay Layered regarding the number of edge crossings and bend points produced as well as its runtime performance. We also evaluate the two methods for assigning port ranks described in Section 3.1 and the two methods for handling north/south-side ports described in Section 4.2 against each other.

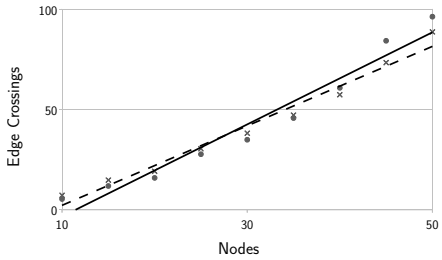
6.1. Layout Quality and Runtime Performance

To evaluate the performance of KLay Layered, we compared it to its predecessor algorithm called KLoDD (*KIELER Layout of Dataflow Diagrams*) [25]. For a first visual impression, Figure 1b shows a drawing created with KLay Layered, while Figure 1a shows a drawing of the same model created with KLoDD.

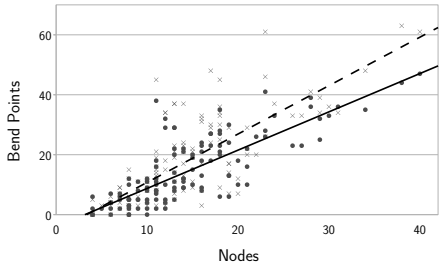
We applied the algorithms to two sets of diagrams in order to evaluate the layout quality. The first set consisted of 270 random graphs with 10 to 50 nodes each and an average of 1.2 outgoing edges per node. This corresponds to the average number of edges found in the demonstration models of the Ptolemy project; a similar value can be derived from the statistics for Simulink models reported by Klauske [16, Section 2.1.2]. Port sides were chosen randomly: input ports would usually be placed on the west side and output ports on the east side, with a probability of 0.05 of this being the other way round, and with a probability of 0.2 of a port being placed on the north or south side. For the second set, we wanted to focus on real-world diagrams. Therefore we used a selection of 141 models taken from the demonstration model repository of the



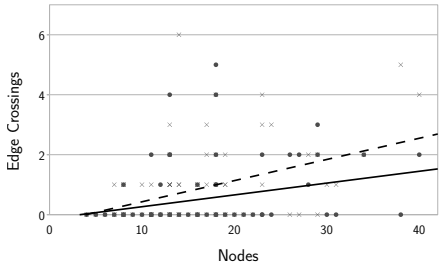
(a) Number of bend points (Random models)



(b) Number of edge crossings (Random models)



(c) Number of bend points (Ptolemy models)

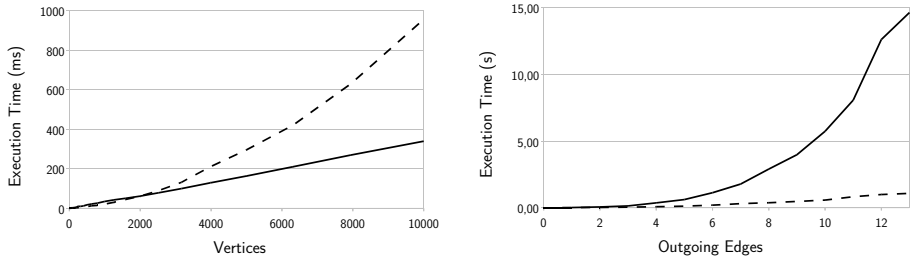


(d) Number of edge crossings (Ptolemy models)

Figure 15: The number of bend points and the number of crossings produced by the KLay Layered algorithm presented here (solid lines and circles) and by the KLoDD algorithm, which follows a previous approach [25] (dashed lines and crosses), applied to our set of random graphs (a, b) and to our selection of Ptolemy models (c, d). Since only few of the Ptolemy models exceed 40 nodes, these larger models were excluded from the diagrams for enhanced legibility.

Ptolemy project. Contrary to the set of random graphs, the graph structure of most Ptolemy models is hierarchical, i. e. some nodes may contain nested subgraphs. The average number of nodes in these subgraphs is 8.98, and their maximum is 43. The layout algorithms were applied separately to each subgraph. Further experiments using flattened versions of the Ptolemy models, where the hierarchical structure was eliminated by directly connecting nodes from different subgraphs, have led to similar results as those shown here.

During the development of KLay Layered, we placed an emphasis on reducing the number of bend points and thus expected it to be lower compared to KLoDD. Due to improved crossing minimization we also expected the number of crossings to be slightly lower. The results of our quality evaluation are shown in Figure 15. Indeed they indicate that the number of bend points produced by KLay Layered is almost consistently lower compared to KLoDD. On average, KLay Layered produced 0.78 and 0.87 times as many bend points as KLoDD for random diagrams and Ptolemy diagrams, respectively, with standard deviations of 0.08 and 0.35. Regarding the number of crossings, the algorithms produce similar results for random diagrams, but KLay Layered does better for Ptolemy



(a) Performance relative to the number of nodes in the graph

(b) Performance relative to the number of outgoing edges per node

Figure 16: The runtime performance of KLay Layered algorithm (solid line) and the earlier KLoDD algorithm (dashed line), plotted against (a) the number of nodes and (b) against the number of outgoing edges per node.

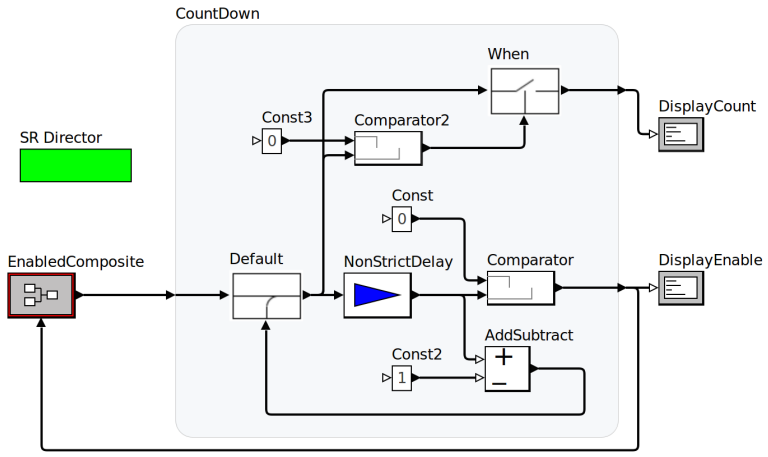
diagrams. On average, KLay Layered produced 0.99 and 0.44 times as many crossings as KLoDD for random diagrams and Ptolemy diagrams, respectively, with standard deviations of 0.40 and 0.67.

For the performance evaluation we used randomly generated diagrams with nearly the same characteristics as the ones already described. Since we wanted to measure the reaction of the algorithms to both changes in the number of nodes and changes in the number of outgoing edges per node, we used two sets of random diagrams. For the first set, we kept the number of outgoing edges per node between 0 and 2, generating graphs with between 10 and $10,000$ nodes. The second set was fixed at 100 nodes, with the number of outgoing edges varying between 0 and 15.

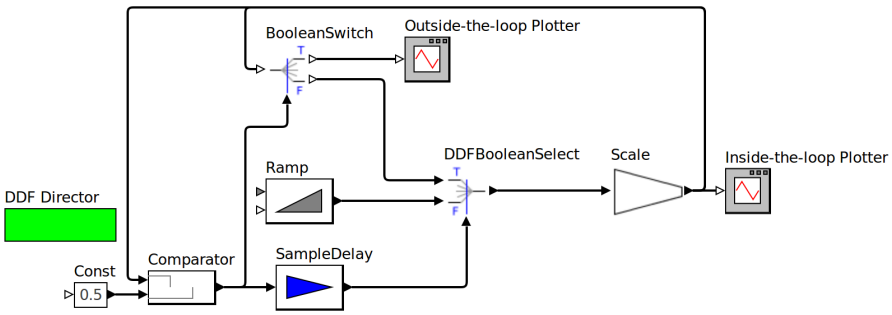
Due to its more complex architecture, we expected KLay Layered to be considerably slower than KLoDD. We were surprised to see that this is not the case, as can be seen in Figure 16. In fact, for large diagrams, KLay Layered shows a linear correlation with the number of nodes. It does not react quite as well to the number of outgoing edges per node, however, which is very likely due to its extensive use of dummy nodes, which KLoDD uses more conservatively. This is consistent with the results of Eiglsperger et al. [5], who found that using as few dummy nodes as possible resulted in a considerable speedup of their algorithm. However, our results show that even with the amount of dummy nodes inserted, the performance is still good enough for KLay Layered to be used in interactive applications. This is even more true considering that Eiglsperger et al. target the layout of large diagrams with over five hundred nodes; Klauske reports data flow diagrams in real-world Simulink models to average 22 nodes, with only 4% of the diagrams exceeding 64 nodes and 100 edges.

We conclude that the advantages gained through dummy nodes in terms of ease of implementation and modular structure are well worth the performance degradation caused by them.

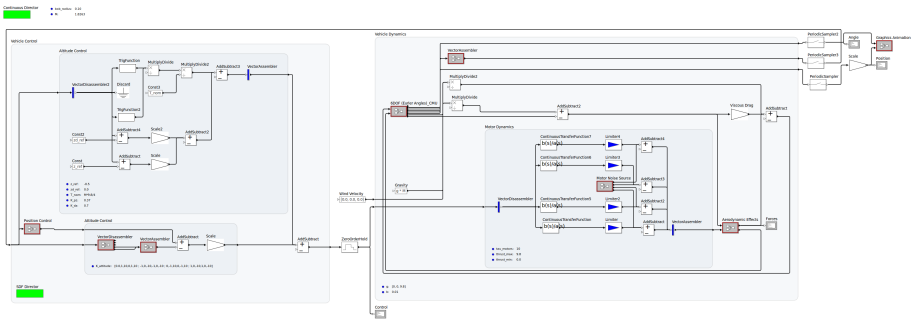
Three diagrams from the Ptolemy demonstration models, drawn with our KLay Layered algorithm, are shown in Figure 17.



(a) Guarded Count (SR domain)
Author: Edward A. Lee



(b) Loop (DDF domain)
Author: Gang Zhou



(c) Starmac (Continuous domain)
Authors: Mankit Leung, Gabe Hoffman

Figure 17: Ptolemy demonstration models drawn with the KLay Layered algorithm. Here we used node-relative port ranking and variable-order handling of north/south ports.

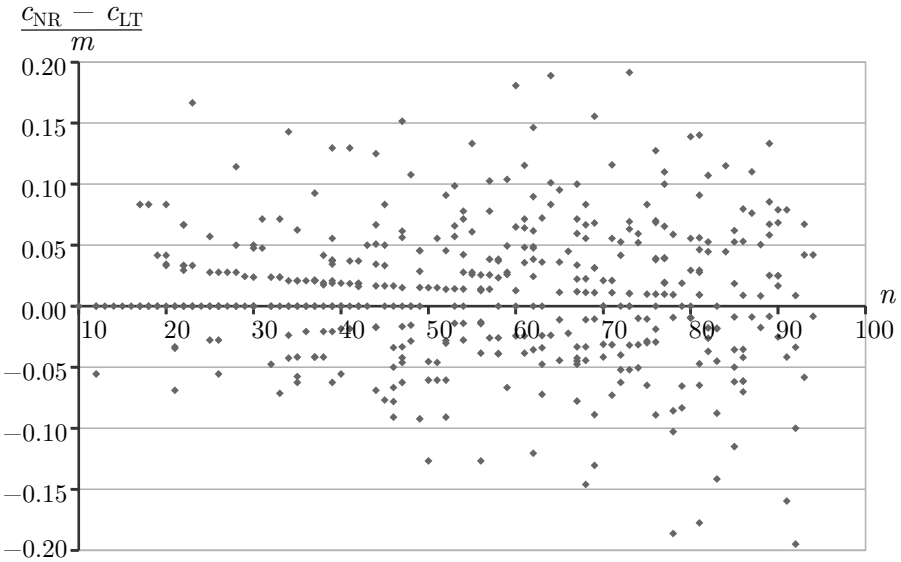
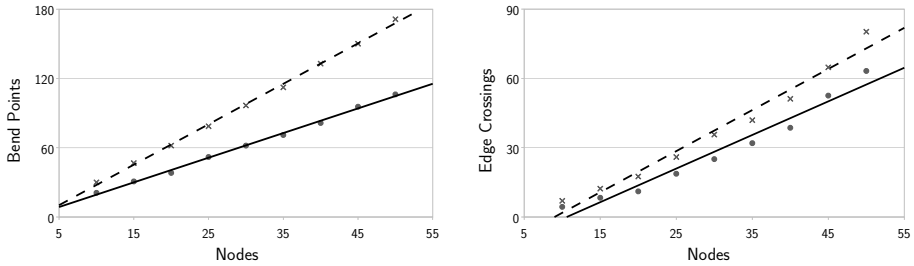


Figure 18: Difference of crossing numbers per edge for each of the random test graphs. The horizontal axis represents the number of nodes n of the test graphs, and the vertical axis represents the value $(c_{NR} - c_{LT})/m$, where c_{NR} is the number of crossings with the node-relative approach, c_{LT} is the number of crossings with the layer-total approach, and m is the number of edges of the corresponding graph.

6.2. Comparison of Port Ranking Methods

In Section 3.1, we introduced two methods for assigning port ranks: the node-relative approach and the layer-total approach. We evaluated them using a set of 570 randomly generated graphs with between 10 and 94 nodes and a similar edge density as used for the random graphs in Section 6.1. The port constraints were set to FixedPos for all nodes. For each of the random graphs the best result out of 1000 randomized executions of the layer sweep algorithm was chosen.

The average number of edge crossings when using the layer-total ranking approach was $\bar{c}_{LT} \approx 58.81$, while with the node-relative approach the value $\bar{c}_{NR} \approx 59.35$ was measured. Although nearly equal in average, the number of edge crossings resulting from the two approaches can be very different when considering a particular graph, as seen in Figure 18. The chart reveals that for many graphs the two ranking approaches yield quite different results both in the positive and the negative direction, but without a significant advantage for any of the approaches. Further experiments have shown that the number of crossings is not improved significantly by applying both the layer-total and the node-relative ranking approach and choosing the better result. Hence the two approaches can be regarded as of equivalent quality, and the only effective means for reducing the number of edge crossings is to increase the number of randomized layer sweeps [15].



(a) Number of bend points (b) Number of edge crossings

Figure 19: The number of bend points and the number of crossings produced by the variable-order approach (solid lines and circles) and the fixed-order approach (dashed line and crosses) with port constraints set to FixedSide.

6.3. Comparison of North/South-Side Port Handling Methods

We compared the two approaches to handling north/south-side ports presented in Section 4.2 in terms of the numbers of bend points and edge crossings produced. We used the same set of random diagrams already used for the layout quality evaluation, with the exception that all ports were moved to the north or south side. We expected the variable-order method to fare much better in both metrics since it gives the crossing minimization algorithm the highest amount of flexibility.

We began with all port constraints set to FixedPos. The results, shown in Figure 19, confirmed our expectations: the variable-order approach fared consistently better than the fixed-order approach. Further experiments with port constraints set to FixedSide yielded the same results: here, too, the variable-order approach produced consistently better results.

7. Conclusion

A large class of graphical languages, including data flow diagrams used for embedded software development, impose port constraints on their drawings. We presented in fair detail a method for the automatic layout of such diagrams, building on the layered approach. A key element of our method is the creation and special treatment of dummy nodes. This comes at a certain cost in terms of runtime performance, but simplifies the handling of port constraints and allows the algorithm to be structured modularly. As shown in the evaluation, the performance of our algorithm is still good enough for it to be used in interactive applications, especially given the size of typical data flow diagrams. Compared to previous approaches, our contributions result in significantly lower numbers of bend points and crossings for realistically sized diagrams.

We also presented our experience with a dynamic and modular structure of layout algorithms based on intermediate processors. Provided that preconditions and postconditions of the processors are carefully defined, this structure

allows algorithms to adapt to the user's requirements and to omit those processors that the current layout problem does not require.

The layout algorithm presented here is implemented in the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER), which is our primary testbed for such algorithmic developments. It is also now part of the Ptolemy distribution and has received quite positive feedback there, regarding quality of the results, performance, and usability. To quote a user: "I have to say we have crossed a tipping point. I am now using automatic layout every time I use vergil [the Ptolemy graphical editor]. I don't see how we ever did without it ...".[8]

In summary, we believe that the method presented here constitutes a significant step towards making the automatic, high-quality drawing of data-flow diagrams and other drawings with port constraints practical and widely used. However, there are still areas for improvement, which we leave for future work:

- The calculation of port positions can contribute to avoiding bend points: the even distribution of ports in Figure 4a causes two bend points in the topmost edge, which can be eliminated by moving up the first port of node 1 as shown in Figure 4b. Port placement methods that target edge straightening would have to be realized either as part of the node placement phase or as a post-processing.
- The layer-sweep crossing minimization approach requires a method for counting the number of crossings in order to find an appropriate terminating condition. While there exist efficient counting methods for plain graphs [1], these are inaccurate when hyperedges are involved because their actual number of crossings is determined later in the edge routing phase [22]. While we introduced several worst case estimates, it is unclear if and how these can be improved to be more accurate.
- We currently process hierarchical diagrams by recursively applying the layout algorithm to each hierarchy level, starting with the innermost ones. This procedure is not optimal when the ports of a compound node are rearranged since the algorithm processing the content of that node does not take into account its external connections.

Acknowledgments

Several people have contributed to the results presented here. We thank in particular Petra Mutzel and Lars Kristian Klauske, for valuable exchanges on the general subject of layout with port constraints, and Hauke Fuhrmann, for the first integration of KIELER layout technology into Ptolemy. We also thank Edward A. Lee, Christopher Brooks, and other members of UC Berkeley's Ptolemy group for being highly supportive of this integration effort. We also thank the users of KIELER and Ptolemy for their feedback regarding usability and quality of automatic layout.

[8] Edward A. Lee, leader of the Ptolemy project, personal communication

References

[1] Wilhelm Barth, Michael Jünger, and Petra Mutzel. Simple and efficient bilayer cross counting. In *Proceedings of the 10th International Symposium on Graph Drawing (GD'02)*, volume 2528 of *LNCS*, pages 331–360. Springer, 2002.

[2] Ulrik Brandes and Boris Köpf. Fast and simple horizontal coordinate assignment. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, volume 2265 of *LNCS*, pages 33–36. Springer, 2002.

[3] Manfred Broy. Challenges in automotive software engineering. In *Proceedings of the 28th International Conference on Software Engineering (ICSE'06)*, pages 33–42, 2006.

[4] Peter Eades, Xuemin Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993.

[5] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of Sugiyama's algorithm for layered graph drawing. In János Pach, editor, *12th International Symposium on Graph Drawing (GD'04)*, volume 3383 of *LNCS*, pages 155–166. Springer-Verlag, 2004.

[6] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003.

[7] Thomas Eschbach, Wolfgang Guenther, and Bernd Becker. Orthogonal hypergraph drawing for improved visibility. *Journal of Graph Algorithms and Applications*, 10(2):141–157, 2006.

[8] Michael Forster. A fast and simple heuristic for constrained two-level crossing reduction. In *Proceedings of the 12th International Symposium on Graph Drawing (GD'04)*, volume 3383 of *LNCS*, pages 206–216. Springer, 2005.

[9] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. Design patterns: Abstraction and reuse of object-oriented design. In Oscar M. Nierstrasz, editor, *ECOOP' 93 - Object-Oriented Programming*, volume 707 of *Lecture Notes in Computer Science*, pages 406–431. Springer Berlin / Heidelberg, 1993.

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

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

[12] Carsten Gutwenger, Karsten Klein, and Petra Mutzel. Planarity testing and optimal edge insertion with embedding constraints. *Journal of Graph Algorithms and Applications*, 12(1):73–95, 2008.

[13] Martin Harrigan and Patrick Healy. Practical level planarity testing and layout with embedding constraints. In *Proceedings of the 15th International Symposium on Graph Drawing (GD'07)*, volume 4875 of *LNCS*, pages 62–68. Springer, 2008.

[14] Patrick Healy and Nikola S. Nikolov. Hierarchical drawing algorithms. In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, pages 409–453. CRC Press, 2013.

[15] Michael Jünger and Petra Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, 1:1–25, 1997.

[16] Lars Kristian Klauske. *Effizientes Bearbeiten von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus*. PhD thesis, Technische Universität Berlin, 2012.

[17] Lars Kristian Klauske and Christian Dziobek. Improving modeling usability: Automated layout generation for Simulink. In *Proceedings of the MathWorks Automotive Conference (MAC'10)*, 2010.

[18] Lars Kristian Klauske, Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. Improved layout for data flow diagrams with port constraints. In *Proceedings of the 7th International Conference on the Theory and Application of Diagrams (DIAGRAMS'12)*, volume 7352 of *LNAI*, pages 65–79. Springer, 2012.

[19] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. Actor-oriented design of embedded hardware and software systems. *Journal of Circuits, Systems, and Computers (JCSC)*, 12(3):231–260, 2003.

[20] Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of *LNCS*, pages 248–261. Springer, 1997.

[21] Georg Sander. Graph layout through the VCG tool. Technical Report A03/94, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, October 1994.

[22] Georg Sander. Layout of directed hypergraphs with orthogonal hyperedges. In *Proceedings of the 11th International Symposium on Graph Drawing (GD'03)*, volume 2912 of *LNCS*, pages 381–386. Springer, 2004.

[23] Falk Schreiber. *Visualisierung biochemischer Reaktionsnetze*. PhD thesis, Universität Passau, Fakultät für Informatik und Mathematik, 2001.

[24] Martin Siebenhaller. *Orthogonal Graph Drawing with Constraints: Algorithms and Applications*. PhD thesis, Universität Tübingen, Mathematisch-naturwissenschaftliche Fakultät, 2009.

[25] Miro Spönemann, Hauke Fuhrmann, Reinhard von Hanxleden, and Petra Mutzel. Port constraints in hierarchical layout of data flow diagrams. In *Proceedings of the 17th International Symposium on Graph Drawing (GD'09)*, volume 5849 of *LNCS*, pages 135–146. Springer, 2010.

[26] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, February 1981.

[27] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, 18(1):61–79, 1988.

[28] Vance Waddle. Graph layout for displaying data structures. In *Proceedings of the 8th International Symposium on Graph Drawing (GD'00)*, volume 1984 of *LNCS*, pages 98–103. Springer, 2001.

[29] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 1(2):103–110, 2002.