Improved Layout for Data Flow Diagrams with Port Constraints

Lars Kristian Klauske[1], Christoph Daniel Schulze[2], Miro Spönemann[2], and Reinhard von Hanxleden[2]

[1] Daimler Center for Automotive Information Technology Innovations, Berlin
lars.klauske@dcaiti.com
[2] Real-Time and Embedded Systems Group, Christian-Albrechts-Universität zu Kiel
{cds,msp,rvh}@informatik.uni-kiel.de

Abstract. The automatic generation of graphical views for data flow models and the efficient development of such models require layout algorithms that are able to handle their specific requirements. Examples include constraints on the placement of ports as well as the proper handling of nested models. We present an algorithm for laying out data flow diagrams that improves earlier approaches by reducing the number of edge crossings and bend points. We validate the quality of our algorithm with a range of models drawn from Ptolemy, a popular modeling tool for the design of embedded systems.

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 [2, 24]. To keep up with the growing complexity and tightening time-to-market requirements, embedded software domains such as the automotive, rail or aerospace industry increasingly take advantage of graphical model-based development tools that follow the *actor-oriented* approach of data flow models [10] such as Simulink (The MathWorks, Inc.), SCADE (Esterel Technologies), ASCET (ETAS), or Ptolemy (UC Berkeley). Herein, graphical diagrams are used as input representations for simulators, rapid prototyping systems, and code generators. Fig. 1 shows a typical data flow diagram from Simulink and reveals the basic components of such a diagram, namely *actors* (also called *blocks* or *operators*), connections between the actors, and *ports* specifying 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, their readability strongly depends on the diagrams' layout. Therefore, when creating or changing a model, an estimated 30% of a user's time is spent on manual layout adjustments according to Klauske and Dziobek [8]. Additionally, interactive applications employing methods such as automatic model generation and transformation gain importance, requiring diagram layouts to be generated from scratch. Both of these problems imply the need for an adequate automatic view generation using methods of graph layout.

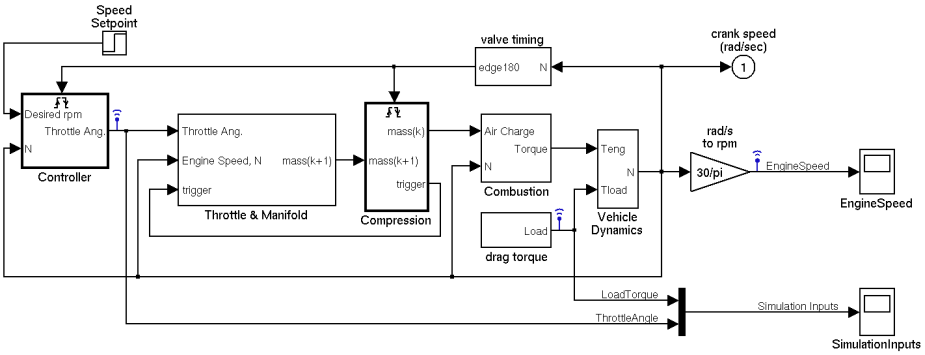


Fig. 1. A Simulink model for engine control (example by The MathWorks, Inc.)

Contributions. In this paper, we address the problem of automatic layout of data flow diagrams. While the difficulties of port constraints and hyperedges in crossing reduction and edge routing, as well as some basic solutions, have already been introduced [19, 9], we show how to further reduce the number of edge crossings and bend points through the creation and handling of additional dummy nodes and extensions of the crossing minimization phase. We also describe an improved method to handle the layout of nested diagrams.

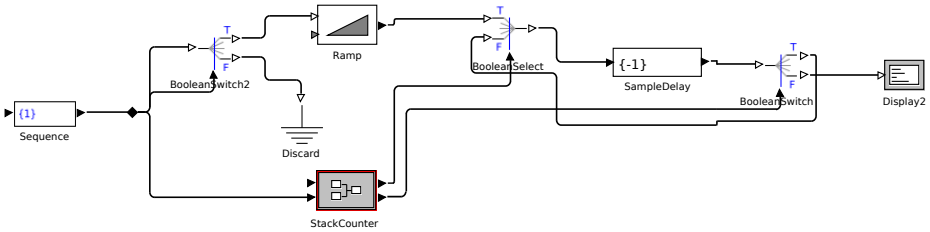
Outline. We begin by introducing two example applications in Sect. 2, give an overview of related work in this area in Sect. 3, and continue by defining the necessary mathematical notation in Sect. 4. In Sect. 5 we provide the description of our algorithm, followed by the evaluation and its results in Sect. 6. Finally we conclude in Sect. 7.

2 Data Flow Models

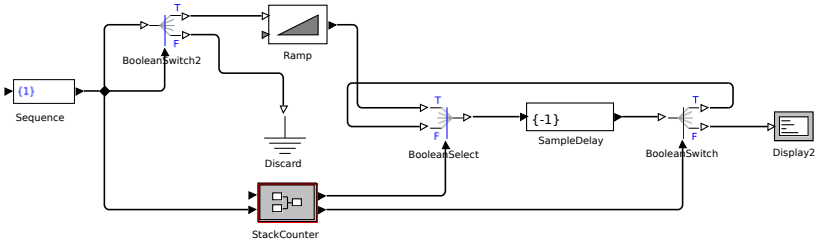
For a closer look at the application domains of our graph drawing method, we present two exemplary modeling tools: Simulink and Ptolemy.

2.1 Simulink

Simulink is a graphical modeling language based on data flow diagrams with optional Statechart diagrams encapsulated inside data flow diagram nodes. It is of widespread use for embedded software development in the automotive domain. Its models are used for specification, simulation, rapid prototyping, and production code generation. Using real-world Simulink models of automotive body control modules with a total of 40,000 nodes and 50,000 edges as our reference, the average Simulink diagram has about 20 nodes and 30 edges, with 90% of all models counting 60 edges or less. While large diagrams of more than 100 nodes do exist (about 1% of our reference diagrams), they usually follow a very simple structure with few or no potential edge crossings and only a couple of layers.



(a) Layout using a previous approach [19] (3 edge crossings, 30 edge bends)



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

Fig. 2. A Ptolemy model representing a stack (example by Edward A. Lee)

The algorithm presented in this paper can directly be applied to Simulink diagrams: Simulink edges can be taken as hyperedges with one source and one or more targets. Ports are arranged on the rectangular node borders, with all output ports on one side, most input ports on the opposite side, and up to three input ports on the remaining sides (see Fig. 1 for a typical example).

The port side and order in Simulink diagrams is always fixed, with port positions that depend roughly linearly on the node size. For the scope of this paper, we assume Simulink node sizes to be fixed, which results in fixed port positions. Simulink diagrams with variable node sizes can be processed using an LP-based edge straightening method [8, 9].

2.2 Ptolemy

Ptolemy[3] is an open source modeling environment developed at UC Berkeley that targets the modeling and semantics of concurrent real-time systems [4]. Ptolemy models are *actor-oriented* data flow diagrams and can contain nested state machines (*modal models*). A Ptolemy data flow model of the process networks domain is shown in Fig. 2.

The layout algorithm presented in this paper has been integrated in the modeling environment of Ptolemy and is now part of its official distribution. Thereby automatic layout can be used as an aid for the creation of Ptolemy models and for the visualization of generated or transformed models.

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

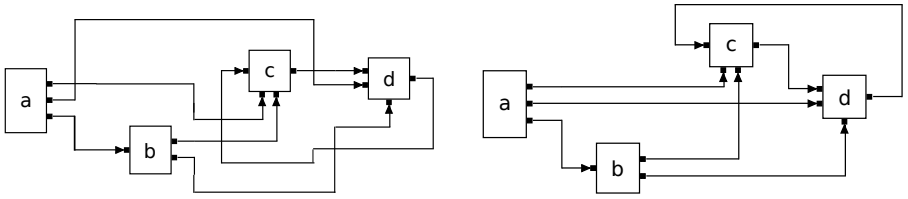
3 Related Work

The foundations for the layout of directed graphs were laid by Sugiyama et al. [21], who introduced the *layered* (a.k.a. *hierarchical*) approach for graph drawing. The basic idea is to organize the nodes in subsequent *layers* such that edges point from layers of lower index to those of higher index. This kind of ordering helps to emphasize the direction of flow, which is quite natural for data flow diagrams. Afterwards the nodes of each layer are reordered so as to minimize the number of edge crossings. This is followed by the calculation of suitable coordinates for node positions, and optionally by an edge routing phase.

The first contributions to the problem of integrating port constraints in the layered 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 [6]. Sander introduced the idea of handling side ports by adding dummy nodes in order to route the respective edges [14]. The problem of crossing minimization with port constraints was first discussed by Waddle, who adapted the standard node ordering heuristic to consider port positions [22]. 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.

Schreiber proposed different solutions in the context of drawing bio-chemical networks [17]. 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. This can be seen in Fig. 3(a), where the incoming edge at the South side port of node d has two additional bend points. The approach of Siebenhaller suffers from the same problem, because it also routes edges of side ports locally [18]. However, it supports more flexible port constraints, since constraints are associated with individual edges instead of nodes. The consequence is that a node may have some edges that are constrained to ports, and some that are not. The crossing minimization problem that results from this additional degree of freedom can be solved by reducing it to a network flow problem. 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, but data flow diagrams usually do not require such mixed constraints.

A previous approach [19] for layout of data flow diagrams applied local routing not only to the side ports of a node, but also to inverted ports, which leads to layouts such as the one shown in Fig. 3(a). Although it simplifies the crossing minimization phase, the obvious drawback is that it cannot take into account the global structure of the graph, and thus leads to an unnecessarily high number of edge crossings and bend points. In this paper we therefore employ a different approach based on dummy nodes, which is able to route the feedback edge (d, c) in



(a) Previous approach [19]: edges at side ports and inverted ports routed locally (5 edge crossings)

(b) Our proposal: edges at side ports and inverted ports routed using dummy nodes (1 edge crossing)

Fig. 3. Two alternatives for handling port constraints.

Fig. 3(b) with four bend points and without any crossings, as opposed to six bend points and three crossings for the previous approach. Other previous contributions adapt the barycenter heuristic for handling different port constraints [19], and add a specialized node placement for FixedRatio constraints [8, 9]. This type of constraint allows changing the size of nodes in order to minimize the number of bend points of incident edges [9].

Orlarey et al. generate data flow diagrams out of textual specifications [12]. Instead of generating a graph and applying a graph layout algorithm to it, they derive the layout directly from an algebraic representation [11]. The compositional nature of this representation implies a geometric node ordering: sequential composition leads to horizontal order, and parallel composition leads to vertical order. Since our contribution is based on graph representations, we will not go into further details on the algebraic approach.

4 Definitions

A *directed port-based graph* consists of a finite set of *nodes* V and *ports* P, a set of *edges* $E \subseteq P \times P$ connecting the ports, and a function $\mathsf{n} : P \to V$ that maps ports to their nodes. An edge $e = (p_1, p_2) \in E$ is an *outgoing edge* of p_1 and v_1 and an *incoming edge* of p_2 and v_2 if $v_1 = \mathsf{n}(p_1)$ and $v_2 = \mathsf{n}(p_2)$; e is said to be *incident* to p_1, p_2, v_1, and v_2. We call p_1 and v_1 the *source* of e, while p_2 and v_2 are called its *target*.

A *layering* of a graph is a partition $\mathcal{L} = (L_1, \ldots, L_k)$ of the nodes into *layers* $L_1, \ldots, L_k$ such that for all edges e with source node $v_1 \in L_i$ and target node $v_2 \in L_j$ we have $i < j$. If we have $i \leq j$, $\mathcal{L}$ is called a *weak layering*, and an edge connecting two nodes in the same layer is called an *in-layer edge*. If $i < j - 1$ then e is called a *long edge* and is split into a sequence of edges that span only consecutive layers by adding *edge dummy nodes*. Each layer has a specific ordering of its nodes which can be altered by the algorithm. The current index of node v in this ordering is written as $\mathsf{idx}(v)$.

Usually ports are drawn on the border of their respective nodes. The function $\mathsf{side} : P \to \{\text{North}, \text{South}, \text{West}, \text{East}\}$ assigns ports to one of the node's

sides. Ports with only incoming edges are called *input ports* and are usually placed on the WEST side. Ports with only outgoing edges are called *output ports* and are usually placed on the EAST side. Input ports that are placed on the EAST side and output ports that are placed on the WEST side are called *inverted ports*. Ports on the NORTH or SOUTH side are called *side ports*.

Port constraints control how much influence a layout algorithm has over the positioning of the ports of a node. The function $\mathsf{cons} : V \to PC$ maps nodes to their port constraints, with PC containing the available port constraints. They are, in increasing order of strictness:

FREE Ports may be drawn at arbitrary positions on the border of a node.
FIXEDSIDES The side is prescribed for each port, but the order of ports is free on each side.
FIXEDORDER The side is fixed for each port, and the order of ports is fixed for each side.
FIXEDRATIO The side is fixed for each port, and the ratio between the port's position on the side and the side's length is fixed.
FIXEDPOS The exact position is fixed for each port.

Due to constraints from the application domains, we assume the graph to be drawn such that the prevalent direction of edges is from left to right. Hence the nodes of a layer L_i are placed on a vertical line, L_{i+1} is drawn right of L_i, and nodes within layers are indexed from top to bottom. Other publications (e. g. Sugiyama et al. [21]) and applications (e. g., UML diagrams) assume a top-down drawing, but our definitions and approaches can be applied symmetrically.

5 The KLay Algorithm

The KLAY Layered algorithm is part of the *Kiel Integrated Environment for Layout Eclipse RichClient* (KIELER)[4] project, a test bed for layout algorithms and modeling pragmatics. The algorithm expects a set of nodes and edges as its input and computes coordinates and bend points to arrive at a layout. It is structurally based upon the layered approach by Sugiyama et al., being divided into several phases as follows:

1. KLAY Layered does not assume the input graph to be acyclic, which requires the first phase to break possible cycles. This is done using the feedback arc set algorithm proposed by Eades et al. [3]. The goal is to have the vast majority of edges point in the same direction, so as to make the flow of data as obvious as possible.
2. As in Sugiyama's approach, a layer assignment phase computes a valid layering for the graph. Long edges are split into segments such that edges only connect nodes in neighboring layers. KLAY Layered provides an implementation of the network simplex layering algorithm by Gansner et al., which minimizes the length of edges [6].

[4] `http://www.informatik.uni-kiel.de/rtsys/kieler/`

3. The order of nodes in a layer determines the number of edge crossings. Solving this problem is NP-complete even for two layers [7], making the use of heuristics necessary. A popular heuristic is the barycenter approach, which works with two layers of which one is fixed. Its nodes are assigned rank values reflecting their order in the layer. For the free layer's nodes, rank values are computed based on the ranks of their fixed-layer neighbors. The nodes are then sorted by their computed ranks to arrive at an ordering for the free layer [21].
   KLay Layered performs forward and backward sweeps through the layers, each time randomizing the order of the sweep's first fixed layer. After a predefined number of sweeps, the result with the least number of edge crossings is chosen.
4. The node placement phase determines the position of nodes inside each layer, making sure not to change the ordering determined by the crossing minimization phase. KLay Layered uses Sander's method for node placement, which partitions the graph's nodes into *linear segments* whose elements are to be kept on a straight line [15]. Typically, edge dummy nodes inserted to divide long edges form a linear segment to keep long edges free of bend points.
5. While Sugiyama's approach uses straight lines to connect nodes, KLay Layered routes edges orthogonally, with bend points set in a way that each segment of an edge runs either horizontally or vertically. This requires the addition of a final edge routing phase, which is based on Sander's hyperedge routing algorithm [16].

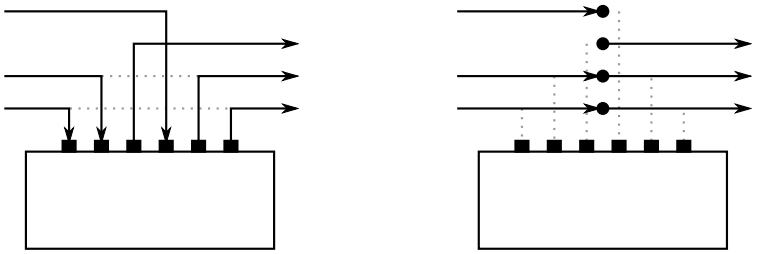
Having already split the algorith into five distinct phases, it is only a small step to allow the concrete implementations to be exchanged at runtime. This way, the algorithm can also be used for different applications. For instance, our main implementation of the edge routing phase routes edges orthogonally, which is the expected method for data flow diagrams. For other types of diagrams, however, edges may be preferred to simply be straight lines, which is supported by another—in that case rather trivial—implementation of the edge routing phase.

We introduce an additional level of modularity by adding *intermediate processing phases* before, between, and after the five main phases. During these intermediate phases, additional modules can be executed that simplify the main phases by factoring out shared functionality, or by reducing complex layout problems to simpler ones that can be handled by the five main phases. Which modules are executed depends on the graph's features: if there are no inverted ports, no corresponding modules need to be executed.

The remainder of this section describes our methods of handling northern and southern ports, inverted ports, and hierarchical ports.

5.1 Side Ports

The usual case for data flow diagrams is for a node to have its input ports on the West side and its output ports on the East side. The situation becomes



(a) A node with side ports and with edges drawn as KLay Layered would. Note that the local number of crossings is minimal.

(b) Inserted bend dummy nodes. (solid circles) Edge dummy nodes can be inserted among them without problems, but the relative order of bend dummy nodes must be fixed to avoid ambiguity.

Fig. 4. Side ports and how bend dummy nodes are created to handle them.

more complicated, however, once nodes are allowed to have ports on the NORTH or SOUTH side, also called *side ports*. Note that this can only happen with port constraints set to FIXEDSIDES or higher. Previous methods transform such nodes to the usual case, either by doing a node-local routing first [17, 19], or by adding a dummy node for the northern and for the southern side which encapsulates the necessary edge routing [14]. All of these approaches suffer from the problem that long edges must be routed around edges connected to side ports, introducing unnecessary bend points or crossings, as can be seen in Fig. 3.

Our method resembles the latter approach, but solves its limitations by allowing more than one dummy node to be created for each side: if the side has x ports, we create between $\lceil x/2 \rceil$ and x *bend dummy nodes* for those ports as shown in Fig. 4. These dummy nodes are created just prior to crossing minimization, and are removed after the last phase, inserting bend points at their position.

This method allows edge dummy nodes to be placed between the bend dummy nodes, which was not previously possible. However, it puts two constraints on the result of the crossing minimization phase: First, generated bend dummy nodes must retain their order to avoid ambiguity due to overlapping edges. And second, the bend dummy nodes generated for different nodes must not be interleaved.

To satisfy these constraints, we add appropriate *successor constraints* on the bend dummy nodes and remember which node they were created for. A successor constraint is a tuple $(v_1, v_2) \in V \times V$ which requires v_1 to be placed above v_2 in a layer. Once an initial order is computed, violated constraints are resolved through a method proposed by Forster [5].

Placing an edge dummy node between two bend dummy nodes causes edge crossings usually not counted by the crossing minimization algorithm, which may lead to inferior results. However, these crossings can be easily counted with a time complexity linear to the number of nodes in a layer.

(a) A dummy node placed in the previous layer. The edge e needs to be reconnected and reversed appropriately, and a new edge connects the dummy node with the original target of e.

(b) A dummy node placed in the same layer. The edge e is not reversed, but reconnected to the dummy node. A new edge connects the dummy node with its original target.

Fig. 5. An inverted port and two approaches for handling it.

One limitation of this method is that the way bend dummy nodes are created is designed to minimize edge crossings locally. Future research could go into finding methods to also take surrounding layers into account.

5.2 Inverted Ports

With port constraints set to at least FixedSides, inverted ports may appear in a diagram. Edges connected to inverted ports need to be routed around the port's node to avoid overlapping. There are two basic previous approaches to handle this situation, both based on turning inverted ports into regular ones. The first does so by applying node-local edge routing, as described in Sect. 5.1 [19]. The second approach handles inverted ports through the addition of a dummy node [8, 9]. Take p to be an inverted port on the West side with an outgoing edge e (Fig. 5(a)). Then a dummy node is added to the preceding layer, and the source of e is changed to the new dummy node. Finally, the dummy node is connected to p.

While the problems of the former approach have already been discussed, the latter approach works reasonably well. However, additional work is required to make sure that the dummy node does not take up space in its layer that could well be used by other nodes. In particular, the inserted dummy node needs additional handling when it is later removed, adding complexity.

KLay Layered therefore uses a different approach, illustrated in Fig. 5(b). After the layer assignment phase, edge dummy nodes are added for edges connected to inverted ports similar to the second approach. The differences are that the dummy node is placed in the same layer, and that the dummy node does not only have outgoing edges. One advantage of this approach is that the inserted dummy node can be treated just like a regular edge dummy node inserted to break long edges.

This of course comes at the cost of turning the layering into a weak layering by the addition of in-layer edges, which has consequences for the crossing minimization phase. For barycenter-based algorithms, it is not immediately clear

what to do with in-layer edges. A problem arises when a barycenter value is to be calculated for a node n_1 which is connected to another node n_2 in the same layer, since n_2 does not have a rank value assigned. We solve this by pretending edges incident to n_2 to also be incident to n_1, and thereby effectively treating n_2 as not being there at all. This has the positive effect of making n_1 and n_2 be closer together, thereby reducing the length and the possibility of crossings due to the in-layer edge connecting them.

This approach also has consequences for cross counting. Usually, cross counting algorithms only count crossings between two layers, not in the same layer. However, a worst-case estimate for crossings caused by in-layer edges can be easily computed in time linear to the number of ports in a layer. First, all ports with incident edges are numbered from top to bottom. Then, for each in-layer edge e, we calculate the difference of the numbers of the ports it connects. We get the maximum number of ports between them whose incident edges will cause crossings with e.

5.3 Hierarchical Ports

In order to control the complexity of large systems, data flow models are broken into hierarchically structured levels using *composite actors*, which are also called *submodules*. Although the nested content of a composite actor is usually displayed in a new window, it is also possible to draw it directly inside the composite actor's bounding box in the containing diagram by enlarging the bounding box accordingly. This leads to a *compound graph* structure [20], where composite actors are represented by *compound nodes*. This kind of visualization allows to directly connect the ports of a composite actor with its content, thus emphasizing the flow of data across hierarchy levels. We call such ports with connections to the inside as well as the outside *hierarchical ports*. Our approach regards each compound node as a separate diagram to be laid out. The hierarchy tree is traversed bottom-up, applying the layout algorithm to the deeper hierarchy levels prior to the containing ones. This method requires the layout algorithm to determine positions for hierarchical ports.

In a previous approach [19], hierarchical ports on the NORTH or SOUTH side were handled by routing their incident edges around a diagram's nodes to dummy nodes inserted into the first layer. This produced long edges and a cluttered diagram, two problems that our new approach solves by eliminating the need to route edges around the diagram.

With port constraints set to FREE, this is staightforward. Dummy nodes are added to the graph and placed in the first or last layer, depending on how many outgoing and incoming edges they have. In the end, the position of hierarchical ports can be directly inferred from where the algorithm placed their dummy nodes.

With port constraints set to FIXEDSIDES or higher, the hierarchical equivalents of side ports and inverted ports can appear. Treating hierarchical ports assigned to the WEST or EAST side is similar to the FREE case. However, for

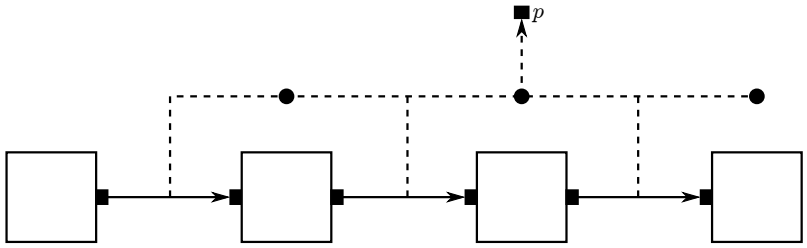


Fig. 6. Inserted dummy nodes to handle hierarchical ports on the NORTH side. Solid circles are inserted dummy nodes the regular nodes connect to. The dashed line indicates how our algorithm routes the edges to the hierarchical port p.

hierarchical ports assigned to the NORTH or SOUTH side, some more work is required. In these cases, KLAY Layered creates dummy nodes for nodes connected to hierarchical ports to connect to instead, as shown in Fig. 6. These dummy nodes are placed above or below all other nodes inside a layer, depending on whether they belong to a NORTH or SOUTH port. In a separate edge routing phase, these dummy nodes are connected to another dummy node representing the hierarchical port itself, the edges between them routed with the orthogonal edge routing algorithm also used for normal edge routing. The position of the hierarchical ports is derived from the position of their dummy node, calculated using a force-based approach that takes the position of connected nodes into account.

With port constraints set to at least FIXEDORDER, this calculation of dummy node positions can lead to invalid results. In these cases, the positions are corrected to adhere to the given hierarchical port order.

In the FIXEDRATIO and FIXEDPOS cases, the position of the hierarchical ports is explicitly prescribed.

One shortcoming of this approach is that our treatment of hierarchical ports does not take external connections into account. Thus, hierarchical ports can be placed in a way that works well within an actor, but leads to unnecessary crossings in the upper hierarchy levels. Ongoing research within our group aims to solve this problem.

6 Evaluation

The quality of layouts is usually measured using a selection of *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. [13, 23]. We evaluated the KLAY Layered algorithm against its predecessor, the KLODD (*KIELER Layout of Dataflow Diagrams*) algorithm [19], comparing the number of produced crossings and bend points. Since both are meant to be used in interactive applications

with users actively waiting for a layout to be generated, we also compared their runtime performance.

For a visual impression, Fig. 2(b) shows a drawing created with KLAY, while Fig. 2(a) shows a drawing of the same model created with KLODD.

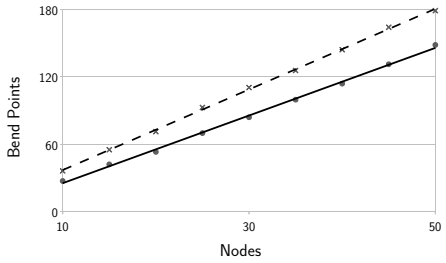
We applied the algorithms to two sets of diagrams in order to evaluate the layout quality. The first set consisted of 270 random graphs with 10 to 50 nodes each and an average of 1.2 outgoing edges per node, which is roughly what we find in real-world data flow diagrams. Port sides were chosen randomly: 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 Ptolemy II tool developed at UC Berkeley and imported them into KIELER. Contrary to the set of random graphs, the graph structure of most Ptolemy models was hierarchical, with each compound node averaging 8.98 child nodes, up to a maximum of 43 child nodes.

During the development of KLAY Layered, we placed some 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 Fig. 7. Indeed they indicate that the number of bend points produced by KLAY Layered is almost consistently lower compared to KLODD. Regarding the number of crossings, the algorithms average fairly similar results, with KLAY Layered having a slight advantage for smaller diagrams.

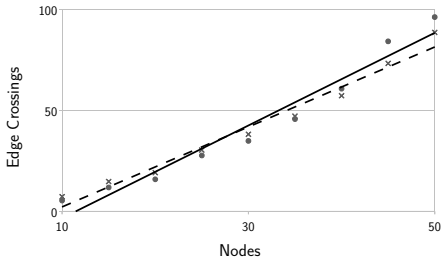
For the performance evaluation we used randomly generated diagrams with nearly the same characteristics as the ones already described. Since we wanted to measure the reaction of the algorithms to both changes in the number of nodes and changes in the number of outgoing edges per node, we used two sets 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.

As for the results, we expected KLAY Layered to be considerably slower than KLODD due to its more complex architecture. We were surprised to see that this is not the case, as can be seen in Fig. 8. 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. This is very likely due to its extensive use of dummy nodes, which KLODD uses more conservatively.

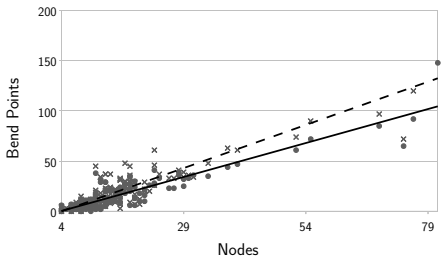
All in all, KLAY Layered performs very well with diagrams from our application domain and is well suited to be used in interactive applications.



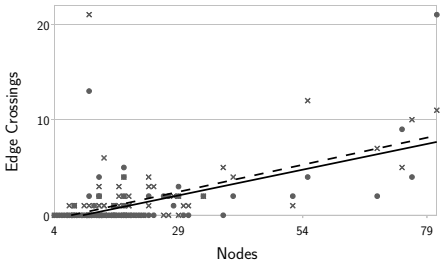
(a) Number of bend points. (Random models)



(b) Number of edge crossings. (Random models)



(c) Number of bend points. (Ptolemy models)



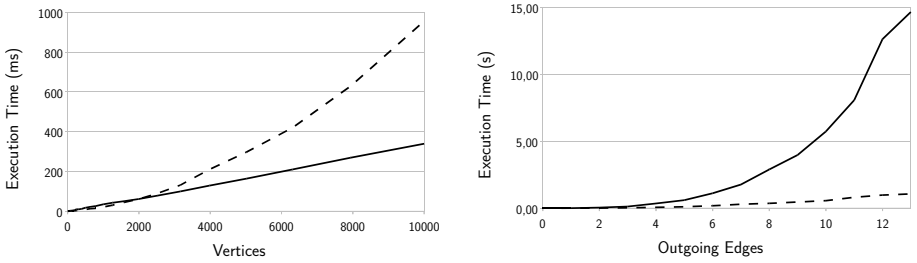
(d) Number of edge crossings. (Ptolemy models)

Fig. 7. The number of bend points and the number of crossings produced by the KLay Layered algorithm presented here (solid lines and circles) and the KLoDD algorithm, which follows a previous approach [19] (dashed lines and crosses), applied to our set of random graphs (a, b) and to our selection of Ptolemy models (c, d).

7 Conclusion

We presented new approaches for handling port constraints as they often appear in data flow diagrams of actor-oriented modeling languages such as Simulink or Ptolemy. These approaches involve the creation and special treatment of dummy nodes. To that end, we introduced enhancements to the crossing minimization phase of the layer-based graph layout method. Compared to previous approaches, our contributions result in significantly lower numbers of bend points and crossings for realistically sized diagrams. However, there is still room for improvements, which we leave for future work:

- 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 [16].
- We currently treat hierarchical diagrams by recursively applying the layout algorithm to each hierarchy level, starting with the innermost ones. This



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

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

Fig. 8. The runtime performance of KLay Layered algorithm (solid line) and the KLoDD algorithm (dashed line), plotted against the number of nodes (a) and against the number of outgoing edges per node (b).

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.

– *Relation nodes* are hypernodes that are used in Ptolemy to connect an arbitrary number of actors. Treating these nodes in the same manner as data flow actors leads to unsatisfying results, and it is not clear what an optimal solution would look like.

References

1. Barth, W., Jnger, M., Mutzel, P.: Simple and efficient bilayer cross counting. In: Goodrich, M., Kobourov, S. (eds.) Graph Drawing, Lecture Notes in Computer Science, vol. 2528, pp. 331–360. Springer Berlin / Heidelberg (2002), http://dx.doi.org/10.1007/3-540-36151-0_13
2. Broy, M.: Challenges in automotive software engineering. In: ICSE 06: Proceedings of the 28th international conference on Software engineering. pp. 33–42 (2006)
3. Eades, P., Lin, X., Smyth, W.F.: A fast and effective heuristic for the feedback arc set problem. Information Processing Letters 47(6), 319–323 (1993)
4. Eker, J., Janneck, J.W., Lee, E.A., Liu, J., Liu, X., Ludvig, J., Neuendorffer, S., Sachs, S., Xiong, Y.: Taming heterogeneity—the Ptolemy approach. Proceedings of the IEEE 91(1), 127–144 (Jan 2003)
5. Forster, M.: A fast and simple heuristic for constrained two-level crossing reduction. In: Proceedings of the 12th International Symposium on Graph Drawing (GD'04), LNCS, vol. 3383, pp. 206–216. Springer (2005), http://dx.doi.org/10.1007/978-3-540-31843-9_22
6. Gansner, E.R., Koutsofios, E., North, S.C., Vo, K.P.: A technique for drawing directed graphs. Software Engineering 19(3), 214–230 (1993)
7. Garey, M.R., Johnson, D.S.: Crossing number is NP-complete. SIAM Journal on Algebraic and Discrete Methods 4(3), 312–316 (1983), http://link.aip.org/link/?SML/4/312/1

8. Klauske, L.K., Dziobek, C.: Improving modeling usability: Automated layout generation for Simulink. In: Proceedings of the MathWorks Automotive Conference (MAC'10) (2010)
9. Klauske, L.K., Dziobek, C.: Effizientes Erstellen von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus. In: Tagungsband Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII. pp. 115–126 (2011), http://www.in.tu-clausthal.de/abteilungen/gi/Forschung/MBEES2011/
10. Lee, E.A., Neuendorffer, S., Wirthlin, M.J.: Actor-oriented design of embedded hardware and software systems. Journal of Circuits, Systems, and Computers (JCSC) 12(3), 231–260 (2003)
11. Orlarey, Y., Fober, D., Letz, S.: An algebraic approach to block diagram constructions. In: Actes des Journèes d'Informatique Musicale (JIM2002). pp. 151–158. GMEM Marseille (2002)
12. Orlarey, Y., Fober, D., Letz, S.: FAUST: an efficient functional approach to DSP programming. In: Assayag, G., Gerzso, A. (eds.) New Computational Paradigms for Computer Music. Editions Delatour France (2009)
13. Purchase, H.C.: Which aesthetic has the greatest effect on human understanding? In: Proceedings of the 5th International Symposium on Graph Drawing (GD'97). LNCS, vol. 1353, pp. 248–261. Springer (1997)
14. Sander, G.: Graph layout through the VCG tool. Tech. Rep. A03/94, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken (Oct 1994)
15. Sander, G.: A fast heuristic for hierarchical Manhattan layout. In: Proceedings of the Symposium on Graph Drawing (GD'95). LNCS, vol. 1027, pp. 447–458. Springer (1996)
16. Sander, G.: Layout of directed hypergraphs with orthogonal hyperedges. In: Proceedings of the 11th International Symposium on Graph Drawing (GD'03). LNCS, vol. 2912, pp. 381–386. Springer (2004)
17. Schreiber, F.: Visualisierung biochemischer Reaktionsnetze. Ph.D. thesis, Universität Passau, Innstrasse 29, 94032 Passau (2001)
18. Siebenhaller, M.: Orthogonal Graph Drawing with Constraints: Algorithms and Applications. Ph.D. thesis, Universität Tübingen, Wilhelmstr. 32, 72074 Tübingen (2009)
19. Spönemann, M., Fuhrmann, H., von Hanxleden, R., Mutzel, P.: Port constraints in hierarchical layout of data flow diagrams. In: Proceedings of the 17th International Symposium on Graph Drawing (GD'09). LNCS, vol. 5849, pp. 135–146. Springer (2010)
20. Sugiyama, K., Misue, K.: Visualization of structural information: automatic drawing of compound digraphs. IEEE Transactions on Systems, Man and Cybernetics 21(4), 876–892 (Jul/Aug 1991)
21. Sugiyama, K., Tagawa, S., Toda, M.: Methods for visual understanding of hierarchical system structures. IEEE Transactions on Systems, Man and Cybernetics 11(2), 109–125 (Feb 1981)
22. Waddle, V.: Graph layout for displaying data structures. In: Proceedings of the 8th International Symposium on Graph Drawing (GD2000). LNCS, vol. 1984, pp. 98–103. Springer (2001)
23. Ware, C., Purchase, H., Colpoys, L., McGill, M.: Cognitive measurements of graph aesthetics. Information Visualization 1(2), 103–110 (2002)
24. Wernicke, M.: AUTOSAR auf dem Weg in die Serie. Elektronik Praxis 02 (2008), http://www.elektronikpraxis.vogel.de/themen/embeddedsoftwareengineering/analyseentwurf/articles/105576/