Port Constraints in Hierarchical Layout of Data Flow Diagrams

Miro Spönemann[1], Hauke Fuhrmann[1],
Reinhard von Hanxleden[1], and Petra Mutzel[2]

[1] Real-Time and Embedded Systems Group, Christian-Albrechts-Universität zu Kiel
{msp,haf,rvh}@informatik.uni-kiel.de
[2] Chair of Algorithm Engineering, Technische Universität Dortmund
petra.mutzel@tu-dortmund.de

Abstract. We present a new application for graph drawing in the context of graphical model-based system design, where manual placing of graphical items is still state-of-the-practice. The KIELER framework aims at improving this by offering novel user interaction techniques, enabled by automatic layout of the diagrams. In this paper we present extensions of the well-known hierarchical layout approach, originally suggested by Sugiyama et al. [22], to support port constraints, hyperedges, and compound graphs in order to layout diagrams of data flow languages. A case study and experimental results show that our algorithm is well suited for application in interactive user interfaces.

1 Introduction

Graphical modeling languages have evolved to appealing and convenient instruments for the development and documentation of systems, both in hardware and in software. There are various examples for graphical modeling frameworks that have become an important part of modern development processes. An important class of modeling diagrams are *data flow diagrams*, which are graphical representations of *data flow models* for design of complex systems. Applications of data flow diagrams can be found in modern software and hardware development tools. Some of these, such as Simulink (The MathWorks, Inc.), LabVIEW (National Instruments Corporation), and ASCET (ETAS Inc.), are mainly used for model-based design and simulation of embedded systems and digital or analog hardware, while others, such as SCADE (Esterel Technologies, Inc.), are optimized for automatic code generation from high-level system models. The Ptolemy project [8] features data flow diagrams for *actor-oriented design*. All these examples feature a graphical editor for data flow diagrams, so that users can create diagrams in drag-and-drop manner.

Typical graphical modeling tools do not support the developer with automatic diagram layout, or do so only in a rudimentary fashion. This leads to unnecessarily high development times, as the developer has to manually adapt the layout after each structural change of the model. In this paper we present methods to apply the hierarchical layout approach [22] to data flow diagrams.

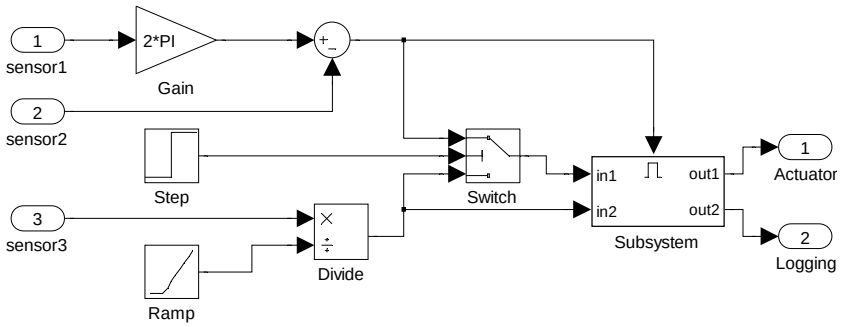


Fig. 1. A data flow diagram from Simulink

We describe constraints which are imposed by such diagrams and show how to extend existing methods to satisfy these constraints. This includes methods for crossing reduction with port constraints and routing of directed hyperedges.

A data flow model is described by a directed graph where the vertices represent *operators* that compute data and the edges represent data paths [6]. Such a data path has a specified *source port* where data is created and a *target port* where data is consumed. A source port may be connected with multiple target ports, thus forming a *hyperedge*. Furthermore, the edges of data flow diagrams are required to be drawn orthogonally. A diagram from Simulink is shown in Fig. 1, which demonstrates the use of ports and hyperedges.

We will proceed as follows. Port constraints, hyperedges and other specialties of data flow diagrams are presented in Section 2. Here, we also introduce four scenarios of port constraints that appear frequently in our applications. Related work is discussed in Section 3. Section 4 describes our methods to handle the special requirements of data flow diagrams within the hierarchical approach. Results of our implementation are shown in Section 5, and we conclude in Section 6. A much more in-depth presentation covering the full hierarchical layout algorithm and details on its implementation can be found on-line [19, 20].

2 Port Constraints and Hyperedges

A *port based graph* is a directed graph $G = (V, E)$ together with a finite set P of *ports*. For each $v \in V$ we write $P(v)$ for the subset of ports that belong to v, and we require $P(u) \cap P(v) = \emptyset$ for $u \neq v$. Each edge $e = (u, v) \in E$ has a specified *source port* $p_s(e) \in P(u)$ and a *target port* $p_t(e) \in P(v)$. We write $v(p)$ for the vertex u for which $p \in P(u)$.

A *drawing* of a port based graph G is a mapping Γ of the vertices, edges, and ports of G to subsets of the plane $\mathbb{R}^2$. In general graph drawing it is sufficient that the drawing of each edge $e = (u, v)$ contacts the drawings of u and v anywhere at their border. For port based graphs the drawing of each port $p \in P(v)$ has a specific position on the border of $\Gamma(v)$, and the edges that have p as source

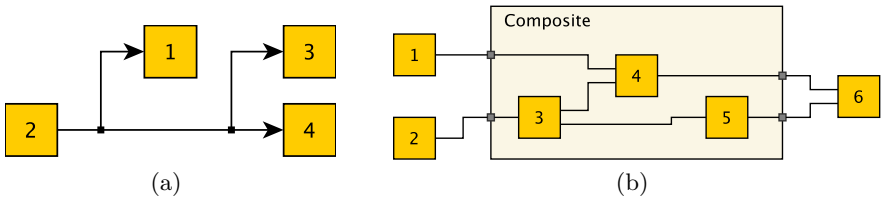


Fig. 2. (a) A hyperedge that connects four vertices (b) The vertex Composite contains connections to external ports, which are shown as small dark boxes on its border.

or target port may touch $\Gamma(v)$ only at that position. We consider four different scenarios for the positions of the ports $P(v)$ on a vertex v:

FreePorts Ports may be drawn at arbitrary positions on the border of $\Gamma(v)$.

FixedSides The side of $\Gamma(v)$ is prescribed for each port, i. e. the top, bottom, left, or right border, but the order of ports is free on each side.

FixedPortOrder The side is fixed for each port, and the order of ports is fixed for each side.

FixedPorts The exact position is fixed for each port.

Mixed-case scenarios, in which some ports of a single vertex have fixed positions and others are free, are not yet covered in our approach, because they require very complex handling and are not needed in our applications.

A *hyperedge* has an arbitrary number of endpoints, thus it may connect more than two vertices. Although there are approaches to directly handle hyperedges [10, 17], we split all hyperedges into sets of plain edges in order to simplify the algorithms. For this reason we consider all edges that are incident at the same port of a vertex as parts of a single hyperedge. For example, the hyperedge shown in Fig. 2(a) would be represented by the edges $(2, 1)$, $(2, 3)$, and $(2, 4)$. Such splitting of hyperedges is not unique if the hyperedge has multiple sources and multiple sinks, but many data flow languages do not allow multiple sources for hyperedges.

In data flow diagrams, each vertex may contain a nested diagram; in this context we have to extend our notion of a graph. A *compound graph* or *clustered graph* $G = (V, H, E)$ consists of a set of vertices V, a set of *inclusion edges* H, and a set of *adjacency edges* E [21]. The *inclusion graph* (V, H) must form a tree, hence for each vertex v we can write $V_{\text{ch}}(v)$ for the set of children of v and $v_{\text{par}}(v)$ for the parent of v. For data flow diagrams the adjacency edges are only allowed to connect vertices that have the same parent in the inclusion tree. However, we treat the ports $P(v)$ of each vertex v as *external ports* of the diagram contained in v, and the children $V_{\text{ch}}(v)$ may be connected to the ports $P(v)$ (see Fig. 2(b)). We employ special edge routing mechanisms to properly connect the ports of a node v with its children.

3 Related Work

Besides the context of system modeling, the term *data flow diagram* and its abbreviation DFD are used in the area of structured software analysis [5]. In this sense DFDs are used for software requirements specification and modeling of the interaction between processes and data. Layout of DFDs has been covered by Batini et al. [3] and Doorley et al. [7]. As DFDs have little in common with data flow diagrams for system modeling, these layout algorithms cannot be applied to our specific problem.

The main specialties that make layout of data flow diagrams for system modeling more difficult than layout of general graphs are ports, hyperedges, orthogonal edge routing, and compound graphs. Previous work on layout with port constraints includes that of Gansner et al. [11] and Sander [14], who gave extensions of the hierarchical approach to consider attachment points of edges. These methods are mainly designed for the special case of displaying data structures and are not suited for the more general constraints of data flow diagrams. A more flexible approach is chosen in the commercial graph layout library yFiles (yWorks GmbH), which supports two models of port constraints and hyperedge routing for the hierarchical approach[3], but no details on the algorithm have been published [23]. Either a *weak* port constraints model (corresponding to FixedSides) or a *strong* port constraints model (corresponding to FixedPorts) can be chosen in yFiles. Other unpublished solutions to drawing with port constraints include ILOG JViews [18] and Tom Sawyer Visualization[4]. Handling of hyperedges in hierarchical layout has been covered by Eschbach et al. [10] and Sander [17]. Sugiyama et al. [21] and Sander [16] showed how to draw general compound graphs, but due to the presence of external ports (see Section 2), our requirements for compound graphs are different. We adapt the orthogonal edge routing approach suggested by Eschbach et al. [10]; alternative approaches have been given by Sander [15, 17] and Baburin [2].

The topic of visualization of hardware schematics is quite related to drawing of data flow diagrams. While traditional approaches for layout of schematic diagrams follow the general *place and route* technique from VLSI design [1, 12], more recent work includes some concepts from the area of graph drawing [9]. However, these concepts are not sufficient for the needs of our application, since they do not address our scenarios for port constraints, but concentrate on partitioning and placement for large schematics and hyperedge routing.

4 Extensions of the Hierarchical Layout Approach

The hierarchical layout method is well suited for laying out directed graphs and aims at emphasizing the direction of flow, thus expressing the hierarchy of vertices in the graph. It was proposed by Sugiyama, Tagawa, and Toda [22] and

[3] yFiles Developer's Guide, `http://www.yworks.com/`
[4] Tom Sawyer Software, `http://www.tomsawyer.com/`

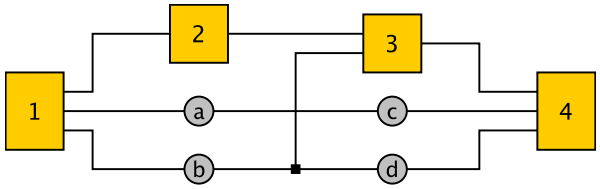


Fig. 3. A layered graph with four layers and three long edges, two of which are part of a hyperedge; the circular vertices are dummy vertices used to split the long edges.

has been extensively studied and improved afterwards. We chose the methods of Di Battista et al. [4] and Sander [15] as a base for our implementation.

Handling of port constraints, hyperedges, orthogonal edge routing, and compound graphs is not addressed in the basic versions of the hierarchical layout algorithm. The following sections will depict our approaches to handle these problems. In hierarchical drawings the directed edges are arranged either horizontally or vertically, but only horizontal layout direction is discussed here, as both variants are symmetric.

4.1 Assignment of Dummy Vertices

In the layer assignment phase we compute layers $L_1, \ldots, L_k$ for the vertices of the acyclic graph G using any standard method. A layering is called *proper* if all edges e connect only vertices from subsequent layers. As illustrated in Fig. 3, a proper layering is constructed from a general layering by splitting *long edges* using dummy vertices. We use *linear segments* to organize the dummy vertices: each vertex v in the layered graph is contained in exactly one linear segment $S(v)$, and a linear segment contains either a single regular vertex or all dummy vertices created for a long edge. These linear segments are used in the vertex placement phase to arrange the dummy vertices of each long edge in a straight line adapting Sander's methods [15]. In Fig. 3, the linear segment of the dummy vertex a is $S(\mathsf{a}) = \{\mathsf{a}, \mathsf{c}\}$.

We customized the linear segments approach to support hyperedges which span multiple layers. In this case, care must be taken to merge the dummy vertices of their corresponding point-to-point edges. For this reason we split long edges by processing them iteratively and associating the linear segment of their dummy vertices with their source and target port. If for any long edge there is already a linear segment associated with its source or target port, the dummy nodes of this linear segment are reused. An example is shown in Fig. 3, where the long edges $(1, 3)$ and $(1, 4)$ share the dummy vertex b.

If the diagram contains external ports, they are also added to the layered graph: input ports, which have only outgoing connections, are assigned to the first layer, while output ports, which have only incoming connections, are assigned to the last layer. In this way the external ports can be treated as normal vertices in the following phases of the algorithm.

4.2 Crossing Minimization

The problem of crossing minimization for layered graphs is usually solved with a *layer-by-layer sweep*: choose an arbitrary order for layer L_1, then for each $i \in \{1, \ldots, k-1\}$ optimize the order for layer L_{i+1} while keeping the vertices of layer L_i fixed. Afterwards the same procedure is applied backwards, and it can then be repeated for a specified number of iterations. We will only cover the forward sweep here, because the backward case is symmetric.

Since the standard layer-by-layer sweep is only applied to vertex positions, we will now look at our extensions for port positions. When ports are used to determine the source and target point of each edge, the number of crossings does not only depend on the order of vertices, but also on the order of ports for each vertex. For each vertex v we define *port ranks* for the ordered ports $P(v) = \{p_1, \ldots, p_m\}$ as $r(p_i) = i$. Furthermore we define extended vertex ranks so that for each $v \in L_i$ and $p \in P(v)$ the sum of the rank of v and the rank of p is unique. The *rank width* of a vertex $v \in L_i$ is $w(v) := 1$ if v was created for a dummy vertex of a long edge or for an external port, and $w(v) := |P(v)|$ otherwise. The extended vertex ranks of the ordered vertices in the layer $L_i = \{v_1, \ldots, v_h\}$ are defined as $r(v_j) := \sum_{g<j} w(v_g)$ for all $j \leq h$.

We implemented the *Barycenter* method for the two-layer crossing problem: first calculate values $a(v) \in \mathbb{R}$ for each $v \in L_{i+1}$, then sort the vertices in L_{i+1} according to these values. Let $E_i(v)$ be the set of incoming edges of v. In our approach, the $a(v)$ values are determined as the average of the combined vertex and port ranks for all source ports of incoming edges of v:

$$a(v) := \frac{1}{|E_i(v)|} \sum_{(u,v) \in E_i(v)} (r(u) + r(p_s(u,v))) \quad . \tag{1}$$

Vertices v_j that have no incoming edges should be assigned values $a(v)$ that respect the previous order of vertices, thus we define $a(v_j) := \frac{1}{2}(a(v_{j-1}) + a(v_{j+1}))$ if $E_i(v_{j+1}) \neq \emptyset$ and $a(v_j) := a(v_{j-1})$ otherwise. By setting $a(v_0) := 0$ and calculating the missing $a(v_j)$ values with increasing j we can assure that $a(v_{j-1})$ is always defined.

For vertices with FixedSides or FreePorts port constraints we have the additional task of finding an order of ports for each vertex that minimizes the number of crossings. We extend the method described above as follows: instead of calculating values $a(v)$ to order the vertices, calculate values $a(p)$ to order the ports first, then calculate

$$a(v) := \frac{1}{|P(v)|} \sum_{p \in P(v)} a(p) \quad . \tag{2}$$

For each port p let $E_i(p)$ be the set of edges which are incoming at that port. Analogously to Equation 1 we define

$$a(p) := \frac{1}{|E_i(p)|} \sum_{(u,v) \in E_i(p)} (r(u) + r(p_s(u,v))) \quad . \tag{3}$$

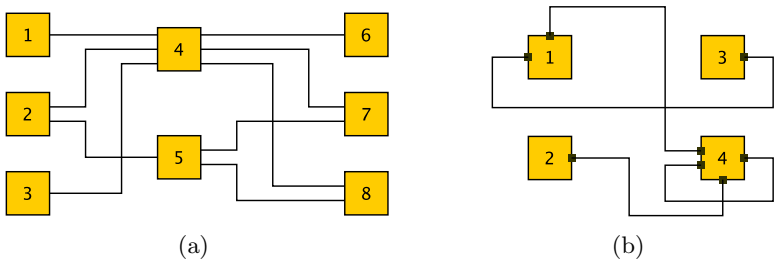


Fig. 4. (a) Routing between layers using vertical line segments (b) Routing around vertices due to prescribed port positions

If there are long hyperedges that share common dummy vertices, as described in Section 4.1, crossing reduction must be adapted to avoid inconsistencies in the following phases. If, for example, backwards crossing reduction is performed for the second layer of the graph in Fig. 3 while keeping the vertices of the third layer fixed as $(3, \mathsf{c}, \mathsf{d})$, it can happen that the dummy vertex b is placed above a because of its outgoing connection to vertex 3. This would lead to a crossing of the edges (a, c) and (b, d), thus the corresponding linear segments $\{\mathsf{a}, \mathsf{c}\}$ and $\{\mathsf{b}, \mathsf{d}\}$ could not be drawn as straight horizontal lines.

To resolve this problem, two new rules must be added for each long edge that is split into dummy vertices $v_1, \ldots, v_k$:

1. For each dummy vertex v_i, $i \in \{2, \ldots, k\}$, only one incoming connection may be considered for crossing reduction, namely (v_{i-1}, v_i).
2. For each dummy vertex v_i, $i \in \{1, \ldots, k-1\}$, only one outgoing connection may be considered for crossing reduction, namely (v_i, v_{i+1}).

4.3 Orthogonal Edge Routing

In order to achieve orthogonal edge routing, each edge that cannot be represented by a single horizontal line needs a vertical line segment (see Fig. 4(a)). A proper order of vertical line segments is important to avoid additional edge crossings, and grouping of hyperedges must be considered. To accomplish this, each port p of a vertex in layer L_i that contains outgoing connections to layer L_{i+1} is assigned a *routing slot* $s(p)$. The resulting routing slots are sorted and given appropriate horizontal positions, and each edge that has p as source port is given two bend points with the respective horizontal position of $s(p)$.

We employ the basic sorting of routing slots as depicted by Eschbach et al. [10], but have to extend it to support the different scenarios of port positions. An additional difficulty arises when the source port of an edge is not on the right side of the source vertex, or the target port is not on the left side of the target vertex. In these cases additional bend points are needed to route the edge around the vertex, as seen in Fig. 4(b). For this purpose routing slots must be assigned on each side of a vertex, similarly to layer-to-layer edge routing. This is done

in an additional phase after crossing reduction; all edges which need additional bend points are processed here, as well as self-loops. The *rank* of a routing slot indicates its distance from the corresponding vertex. For example, the self-loop $(4, 4)$ in Fig. 4(b) is assigned routing slots of rank 1 on the left, bottom and right side of vertex 4, while the edge $(2, 4)$ is assigned a routing slot of rank 2 on the bottom side of vertex 4.

As an output of this additional routing phase, the number of routing slots for the top and the bottom side of each vertex v, together with the given height of v, determines the amount of space that is needed to place v inside its layer. This information is passed to the vertex placement phase, so that the free space that is left around each vertex suffices for its assigned routing slots.

4.4 Compound Graphs with External Ports

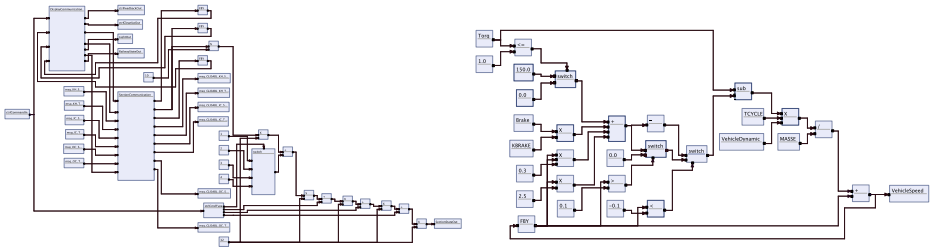
For general compound graphs $G = (V, H, E)$, the adjacency edges E are allowed to connect vertices from different levels of the inclusion tree (V, H). As this is not the case for data flow diagrams, we do not need to employ the special versions of the hierarchical layout method for compound graphs [21, 16], but can follow a simpler approach, which consists of executing the layout algorithm recursively, starting with the leaves of the inclusion tree.

However, the presence of external ports (see Fig. 2(b)) leads to the additional problem that edges of the nested graph may be connected to these ports, which may be subject to any of the four scenarios of port constraints described in Section 2. During edge routing such connections must be specially handled, in particular if there are input ports which are not on the left side of the nested diagram, or output ports which are not on the right side. These cases require additional bend points, and if there are multiple edges which need to be routed along the top or bottom side of the nested diagram, the order of these edges must be adjusted to minimize the number of crossings. We achieve this through similar techniques as those used for layer-to-layer edge routing.

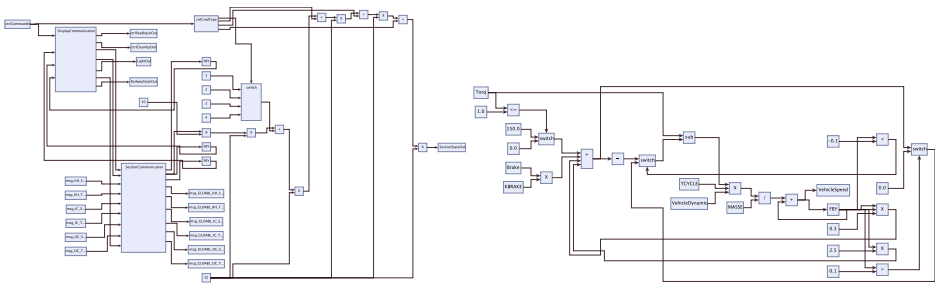
5 Implementation and Results

An implementation of our layout algorithm is part of the Kiel Integrated Environment for Layout for the Eclipse RichClientPlatform (KIELER)[5]. KIELER is a platform for experimental approaches to graphical model-based design and for combination of different aspects of graphical modeling, such as methods of model editing, visualization of simulation, and automatic layout. Unlike its preceding project, the Kiel Integrated Environment for Layout (KIEL), which was developed as a stand-alone Java application [13], KIELER builds on Eclipse, an extensible platform comprised of various integrated development environments. Our Eclipse interface enables the layout functionality for editors of the Eclipse

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



(a) left: 47 edge crossings, right: 4 edge crossings



(b) left: 41 edge crossings, right: 13 edge crossings

Fig. 5. Comparison with yFiles: (a) our layout method (b) layout with yFiles

Graphical Modeling Framework (GMF)[6] and hence for a wide variety of graphical editors. Since the algorithm is written in Java, it can also be used as a plain class library outside of Eclipse.

Fig. 5(a) shows results of automatic layout in the FixedPorts scenario for port positions. Here we see the effectiveness of our method of crossing minimization, as the order of vertices in each layer is adapted to the fixed port positions. Figure 5(b) shows the same diagrams with layouts created in yEd, a free graph editor of yWorks GmbH which includes the yFiles layout library. The results demonstrate that our layout method is comparable with the commercial library yFiles with regard to layout with port constraints.

To test our layout algorithm in an existing modeling framework, it was integrated into *Vergil*, the editor for Ptolemy II developed at UC Berkeley by Lee et al. [8]. Ptolemy II is a graphical modeling tool for exploration of the semantics of different models of computation of formalisms for embedded software design. Its heterogeneous nature enables to mix it with models of physical phenomena to result in full system models including the software controller and its physical environment.

Graphical representations of Ptolemy models can be mapped almost directly to the layout problem described in this paper. Ptolemy *actors* are the interconnected software components represented by nodes which consume and produce data at dedicated ports. Connections can be joined by *relation vertices* to ob-

[6] `http://www.eclipse.org/modeling/gmf/`

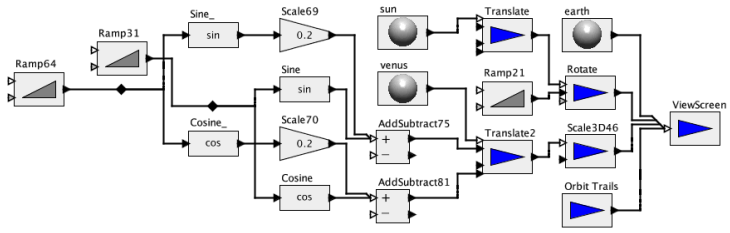


Fig. 6. Layout of a Ptolemy II model

tain hyperedges that share a common data source. However, Ptolemy does not yet support the setting of connection bendpoints, but dynamically routes them internally. Additionally, the data flow of ports is sometimes bidirectional, which results in undirected edges. As our algorithm requires directed graphs, a heuristic chooses explicit directions for all edges first.

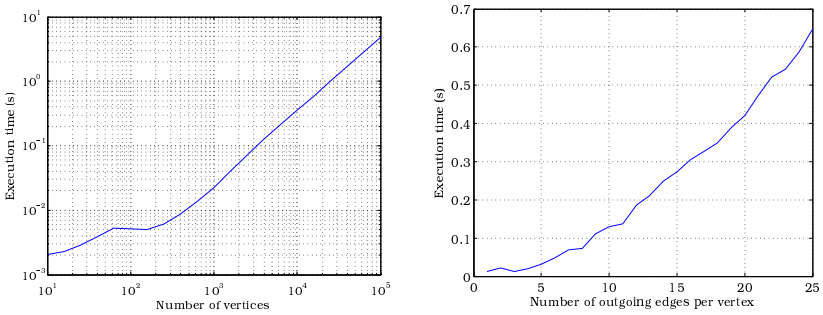
The Ptolemy editor Vergil is not based on Eclipse but implemented in plain Java. Hence we used our stand-alone algorithm library and interfaced it with the graphical drawing backend of Ptolemy. Initial results produce diagrams such as those depicted in Fig. 6 and show that the algorithm is applicable for an important set of real-world system modeling tools. More details about the Ptolemy integration can be found elsewhere [20].

Measurement data for the execution time of the hierarchical layout method are shown in Fig. 7. For graphs with about 25 000 vertices and the same number of edges the algorithm takes less than a second, which proves its suitability for automatic layout in a user interface environment. However, the execution time highly depends on the average vertex degree, since layout for a graph with 2 000 vertices and 2 000 edges is 8 times faster than layout for 100 vertices and 2 000 edges. One reason for this is that for vertices with a lot of incident edges the number of long edges that stretch over multiple layers is likely to be high, so that dummy vertices must be inserted to obtain a proper layering. The consequence is that the problem size rises with regard to the total number of vertices.

6 Conclusion

We introduced four scenarios of port constraints for graph drawing and presented methods to extend the hierarchical layout approach to handle ports, hyperedges, orthogonal edge routing, and compound graphs. These methods are implemented in KIELER, an Eclipse based framework for research on the pragmatics of graphical modeling. The results of our implementation and the low execution times demonstrate its suitability to enhance graphical modeling tools by automatic layout of data flow diagrams. Further work can be done to improve the layout quality:

– Additional support for layout of edge labels.



(a) (b)

Fig. 7. Execution time for (a) varying number of vertices and one outgoing edge per vertex (b) 100 vertices and varying number of outgoing edges per vertex

- Direct support of directed hyperedges with multiple sources and multiple targets.
- Some data flow languages such as SCADE allow to integrate *Statecharts* in their data flow diagrams. A layout algorithm should be able to handle this, i. e. arbitrarily mix nodes with and without port constraints and hyperedges.
- Some vertices in data flow diagrams are very large, thus forcing their respective layer to be large. This could be improved by possibly stretching large vertices over multiple layers.

References

1. Anjali Arya, Anshul Kumar, V. V. Swaminathan, and Amit Misra. Automatic generation of digital system schematic diagrams. In *DAC '85: Proceedings of the 22nd ACM/IEEE Conference on Design Automation*, pages 388–395. ACM, 1985.
2. Danil E. Baburin. Using graph based representations in reengineering. *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 203–206, 2002.
3. Carlo Batini, Enrico Nardelli, and Roberto Tamassia. A layout algorithm for data flow diagrams. *IEEE Transactions on Software Engineering*, 12(4):538–546, 1986.
4. Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.
5. Ned Chapin. Some structured analysis techniques. *SIGMIS Database*, 10(3):16–23, 1978.
6. Alan L. Davis and Robert M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, Feb 1982.
7. Michael Doorley and Anthony Cahill. Experiences in automatic leveling of data flow diagrams. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension*, pages 218–229, Washington, DC, USA, 1996. IEEE Computer Society.
8. 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.

9. Thomas Eschbach. *Visualisierungen im Schaltkreisentwurf*. PhD thesis, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, June 2008.
10. Thomas Eschbach, Wolfgang Guenther, and Bernd Becker. Orthogonal hypergraph drawing for improved visibility. *Journal of Graph Algorithms and Applications*, 10(2):141–157, 2006.
11. 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.
12. C. R. Lageweg. Designing an automatic schematic generator for a netlist description. Technical Report 1-68340-44(1998)03, Laboratory of Computer Architecture and Digital Techniques (CARDIT), Delft University of Technology, Faculty of Information Technology and Systems, 1998.
13. Steffen Prochnow and Reinhard von Hanxleden. Statechart development beyond WYSIWYG. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, Nashville, TN, USA, October 2007.
14. Georg Sander. Graph layout through the VCG tool. Technical Report A03/94, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, October 1994.
15. Georg Sander. A fast heuristic for hierarchical Manhattan layout. In *GD '95: Proceedings of the Symposium on Graph Drawing*, volume 1027 of *LNCS*, pages 447–458. Springer-Verlag, 1996.
16. Georg Sander. Layout of compound directed graphs. Technical Report A/03/96, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, June 1996.
17. Georg Sander. Layout of directed hypergraphs with orthogonal hyperedges. In *GD 2003: Proceedings of the 11th International Symposium on Graph Drawing*, volume 2912 of *LNCS*, pages 381–386. Springer-Verlag, 2004.
18. Georg Sander and Adrian Vasiliu. The ILOG JViews graph layout module. In *GD 2001: Proceedings of the 9th International Symposium on Graph Drawing*, volume 2265 of *LNCS*, pages 469–475. Springer-Verlag, 2002.
19. Miro Spönemann. On the automatic layout of data flow diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/msp-dt.pdf`.
20. Miro Spönemann, Hauke Fuhrmann, and Reinhard von Hanxleden. Automatic layout of data flow diagrams in KIELER and Ptolemy II. Technical Report 0914, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-0914.pdf`.
21. Kozo Sugiyama and Kazuo Misue. Visualization of structural information: automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man and Cybernetics*, 21(4):876–892, Jul/Aug 1991.
22. 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.
23. Roland Wiese, Markus Eiglsperger, and Michael Kaufmann. yFiles: Visualization and automatic layout of graphs. In *GD 2001: Proceedings of the 9th International Symposium on Graph Drawing*, volume 2265 of *LNCS*, pages 588–590. Springer-Verlag, 2001.