Christoph Daniel Schulze

Optimizing Automatic Layout for Data Flow Diagrams

Diploma Thesis
July 28th 2011

Christian-Albrechts-Universität zu Kiel
Department of Computer Science
Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden

Advised by
Dipl.-Inf. Miro Spönemann

Christoph Daniel Schulze

Optimizing Automatic Layout for Data Flow Diagrams

Diploma Thesis

Abstract

Data flow diagrams are a common tool to model data-driven systems. When used well, they can yield better readability and a more intuitive understanding of the modeled system than possible with textual languages. This, however, requires the diagram elements to be carefully placed and edges to be routed well—a very time-consuming task for the developer greatly simplified by the use of automatic layout algorithms.

While general layout algorithms usually do not work well for the unique requirements of data flow diagrams, there have already been attempts to develop specialized algorithms. However, the layouts they produce often suffer from too many bend points or edge crossings, thereby compromising readability.

This thesis aims at improving this by extending a basic layered layout algorithm to adapt it to the special requirements of data flow diagrams. Support for ports and port constraints is discussed as it applies to normal ports and to hierarchical ports, a method to handle self-loops is introduced, and several further enhancements are developed. Furthermore, a flexible new structure for the implementation of layout algorithms is proposed.

Experimental results show the resulting algorithm to fair considerably better with regard to several important layout aesthetics criteria than existing algorithms.

Eidesstattliche Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Hilfsmittel verwendet habe.

Kiel,

As soon as you concern yourself with the “good” and “bad” of your fellows, you create an opening in your heart for maliciousness to enter. Testing, competing with, and criticizing others weaken and defeat you.

— *Morihei Ueshiba, founder of Aikido*

Contents

Abbreviations

DTD	Document Type Definition A format to define the structure of XML-based documents.
ECU	Electronic Control Unit An embedded device that controls an electrical system.
EMF	Eclipse Modeling Framework A modeling framework popular among Eclipse projects.
GMF	Graphical Modeling Framework A framework for building graphical editors for EMF models.
KAOM	KIELER Actor Oriented Modeling A framework for actor-oriented models used inside KIELER.
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client A test bed for automatic layout.
KIML	KIELER Infrastructure for Meta Layout A layout framework that connects editors and layout algorithms.
KLay	KIELER Layouters An umbrella project for KIELER layout algorithms.
KLoDD	KIELER Layout of Dataflow Diagrams The predecessor of KLay Layered.
MoML	Modeling Markup Language An XML-based file format used by the Ptolemy tool.
OGDF	Open Graph Drawing Framework A set of layout algorithms developed at the University Dortmund.
SVG	Scalable Vector Graphics An open, XML-based vector graphics format.
XML	Extensible Markup Language A markup language designed to transport and store data.

1 Introduction

Hello and again welcome to the Aperture Science Computer-Aided Enrichment Center. We hope your brief detention in the relaxation vault has been a pleasant one. Your specimen has been processed and we are now ready to begin the test proper. Before we start, however, keep in mind that although fun and learning are the primary goals of the enrichment center activities, serious injuries may occur. For your own safety, and the safety of others, please refrain from–

— *GLaDOS, Portal*

To begin with, we take a look at graphical modeling, using data flow diagrams as a popular example. We briefly examine its merits and problems, and how automatic layout can elegantly solve many of them, paying special attention to the problems unique to the automatic layout of data flow diagrams. This will have introduced enough context to state the goals of this thesis. We then look at related work in this and surrounding areas, and how this thesis relates to it, and close with a short description of how the thesis is structured.

▶ 1.1 Graphical Modeling and Its Problems

Looking at the history of information technology, we can see an ever-increasing complexity of systems developed. A few decades ago, computers were used for calculations with hand-crafted single-purpose programs. Today, companies such as Google, Facebook and Amazon have complex software running on huge server farms. Systems have gone from being developed by small teams to being huge leviathans, consisting of several hundred thousand lines of code, developed by large teams distributed all over the world. Along the way, the necessity arose for tools able to meet the challenge of developing such systems.

Assembly languages, formerly the only tools available, were all but replaced by increasingly higher-level languages. At the sacrifice of giving up low-level control, such languages allow developers to abstract from the details and concentrate instead on the actual problem to be solved. Languages such as C and Java have introduced many features convenient to the programmer, but they can only be used to describe the way to solve a problem, not the problem itself. Languages such as Prolog take abstraction one step further, letting the programmer specify a model of the problem he wants to have solved instead of the actual program solving the problem. Domain-specific languages take abstraction to the extreme, being developed specifically to describe problems of a certain domain.

With textual languages, it can be hard to gain an understanding of existing systems. Graphical modeling languages try to remedy this by not relying on text, but on graphical symbols that can be much easier to read and quicker to understand than textual representations. As an example, think of a big software system written in Java with hundreds of classes and thousands of lines of code. Imagine how difficult it would be to understand the structure of the system by merely looking at the source code; now imagine how much easier it would be if there were some class diagrams to look at.

Graphical modeling languages can describe the structure of a system, as class diagrams do, but they can also describe the behavior of the system. For the rest of this thesis we focus our attention on an example of such behavioral diagrams.

Data Flow Diagrams

Data flow diagrams are a graphical representation of data flow models, modeling the flow of data through the components of a system. As the data flows through the different components, computations are performed and new data is passed along. The level of detail is arbitrary: data flow diagrams can be used both to provide a very high-level view of a system, or to describe the flow of data in minute detail. While data flow models can have different semantics [16, 19], we are only interested in the graphical representation in this thesis.

Figure 1.1 shows an exemplary data flow diagram taken from the repository of demo models shipped with Ptolemy, a modeling tool developed at UC Berkeley [9]. The general structure of the model—two loops—becomes apparent at first glance. It does not take much longer to realize that both loops are probably supposed to compute the same values, their difference being evaluated and displayed for monitoring. All of this is readily apparent from the way the components of the diagram are placed. Other tools that use data flow diagrams include Simulink,[1] a tool for the simulation and model-based design of embedded systems developed by MathWorks, and ASCET,[2] a model-based development and code generation tool developed by ETAS that specifically targets the automotive domain.

[1]http://www.mathworks.com/products/simulink/
[2]http://www.etas.com/en/products/ascet_software_products.php

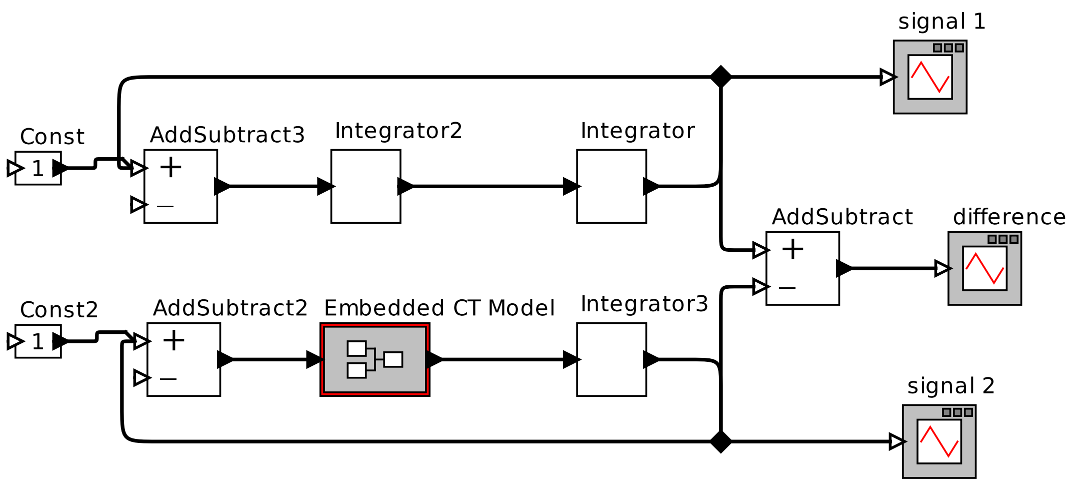


Figure 1.1. Example of a data flow diagram, taken from the repository of demo models shipped with Ptolemy. (The model was created by Haiyang Zheng.) Data are generated by the two components on the left, labeled `Const` and `Const2`, and are processed by two feedback loops, whose current values are displayed by the two components on the right, labeled `signal1` and `signal2`. The difference in value of the two loops is computed by the component labeled `AddSubtract` and displayed by the component labeled `difference`.

In this thesis, the different components of the diagram are called *vertices*. They are connected by *edges*. The placement of vertices and routing of edges constitute the diagram's *layout*. The first two terms are taken from graph theory, simply because data flow diagrams can be viewed as graphs. Section 2.1, defines all the necessary terms formally.

In this basic example we can already observe three properties unique to data flow diagrams. The vertex labeled `AddSubtract` has three incident edges. Instead of being connected directly to the vertex, though, there are three dedicated connection points for the edges. These points are called *ports*. Ports can be thought of as channels where the arguments for the computation come in, and where its results are sent out. The second property is the way in which edges are routed. For data flow diagrams, edges are usually routed orthogonally, with the segments of an edge being either horizontal or vertical, but never diagonal. The third property pertains to the flow of data. As can be seen in the example, data flow diagrams are usually structured in a way to emphasize the flow direction, with data originating on the left side and flowing to the right side.

Automatic Layout

Figure 1.2 shows another version of the diagram in Figure 1.1. Comparing the two, notice how in the first diagram the flow of data is readily apparent, while in the second diagram it is not, emphasizing how the readability of a diagram is a direct result of its layout. (I give a proper definition of "readability" in Section 5.1; for now, think of a diagram's readability as how well it conveys the flow of data through

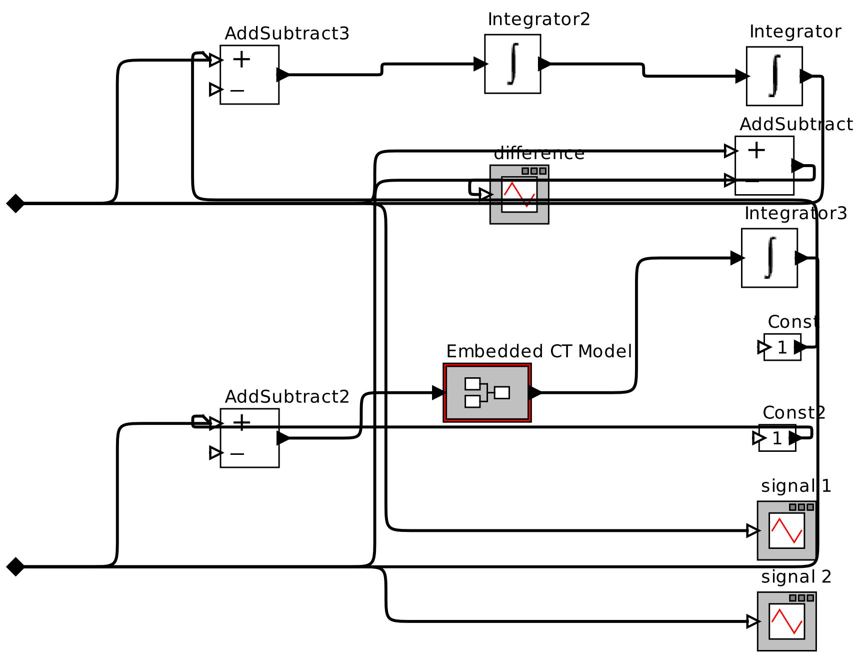


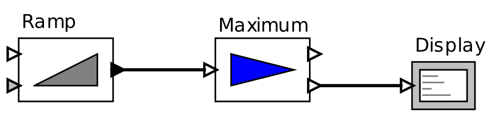
Figure 1.2. The diagram from Figure 1.1, with vertices placed in a way that makes the diagram virtually unreadable. The layout doesn't emphasize any data flow direction; loops are not recognizable; edges overlap vertices, and even other edges; in short, the diagram does away with all the potential benefits of graphical modeling. Of course, this example is exaggerated. But it does illustrate how much of its usefulness a diagram draws from its layout.

the depicted system.) Since readability is one of the main advantages of graphical modeling languages, Developers spend a great deal of their time just moving vertices around to improve their diagrams. Klauske and Dziobek speak of 30% of development time spent on layout-related tasks [17].

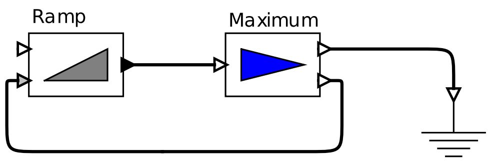
The problem gets worse when modifying existing diagrams. Suppose that in the diagram of Figure 1.1, we wanted to add another vertex just before the last integrator in each loop. We would have to make space available for the two new vertices, insert them, and make sure we didn't mess up the vertex placement along the way. While that is certainly feasible in this simple example, it is a major undertaking in more complex diagrams.

We have seen by now that a diagram's layout directly influences its readability, but also that getting the layout right is a major time factor. So, why not let the computer calculate layouts?

The discipline concerned with developing algorithms for that task is called *graph drawing*. Graph drawing algorithms take an undirected or directed graph as input. The output is a drawing of some kind, with vertex coordinates computed and edges routed accordingly. However, these algorithms usually have no concept of ports, giving rise to the need for specialized algorithms.



(a) The easy case. Edges always go from left to right, originating at an eastern port and going into a western port.



(b) The hard case. One edge originates from an eastern port and goes into a western port. There is even an edge going into a northern port. Special care has to be taken to handle such cases.

Figure 1.3. Easy and hard cases for port handling.

▶ 1.2 Goals of this Thesis

In short, the goals of this thesis were to extend an existing layer-based layout algorithm (named KIELER Layouters (KLay) Layered) to fully support port constraints, hierarchical ports, and self-loops. Further improvements to the algorithm were also welcome contributions. Experiments show the developed methods to increase the quality of generated layouts compared to previous approaches [26].

The rest of this section elaborates a little on these goals and closes with the main contributions presented in this thesis. Further details are given in Section 2.3 once KLay Layered is introduced.

Port Constraints

In data flow diagrams, vertices can usually be thought of as rectangles, with ports placed at one of the rectangle's four sides: north, south, west, and east. Extending a layered layout algorithm to support ports is easy if edges always go from an eastern port to a western port, as depicted in Figure 1.3a.

However, there is no necessity for input ports to always be on the western side, and output ports to always be on the eastern side. Figure 1.3b is an example of a model that does not fit the simple case. The rightmost vertex has a northern port. The `Ramp` and `Maximum` vertices form a loop. If the long edge connecting the two were drawn as a straight line, it would cross the vertices, making the diagram less readable.

This thesis aims at the development of effective methods for handling such cases.

Hierarchical Ports

To understand hierarchy in data flow diagrams, take another look at Figure 1.1. The vertex labeled `Embedded CT Model` actually contains another data flow diagram, said to be on a lower hierarchy level. Ptolemy always shows only one level, but other tools allow us to see more. Figure 1.4 shows what that can look like. When multiple levels of hierarchy can be visible at once, the layout algorithm needs to support that.

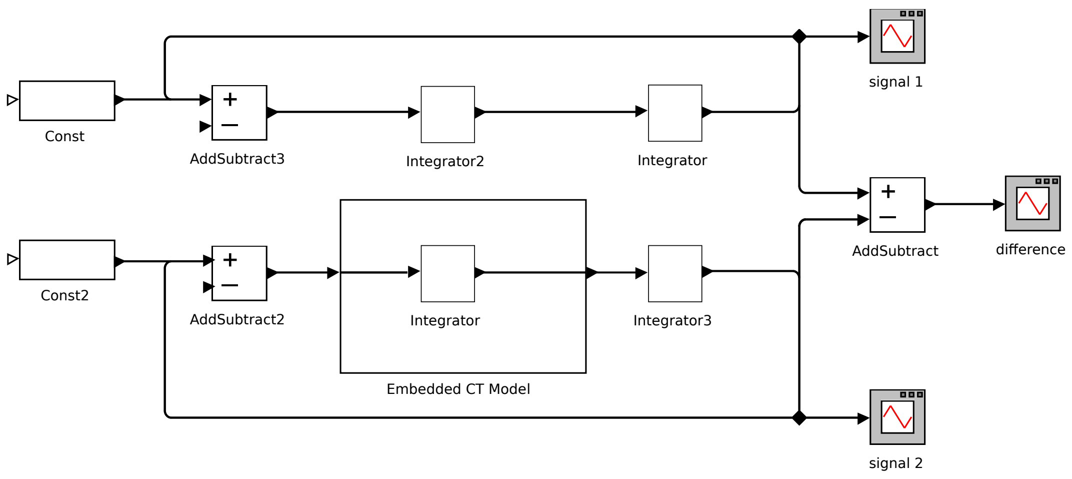


Figure 1.4. The diagram from Figure 1.1, displayed in a different way. Here, the hierarchical vertex `Embedded CT Model` is expanded—we can "look into it," seeing two levels of hierarchy at once.

Notice how the vertex inside `Embedded CT Model` is connected to its ports from the inside. In this thesis, such ports are referred to as *hierarchical ports*. Connecting them to the vertices on the inside needs special consideration.

Self-Loops

Self-loops are edges that connect two ports belonging to the same vertex. Such edges have to be routed around their vertex, but how this can be done has received little research interest in the past.

For this thesis, my goal was to integrate the handling of self-loops elegantly into KLay Layered which did not support them.

Main Contributions

My main contributions are the following:

- A method to handle edges that connect two vertices in the same layer, called *in-layer edges*. This is then used to develop an improved way of handling edges that must be routed around their source or target vertex to avoid having to cross it.
- A new kind of dummy vertex and a way of using it to handle ports placed on the northern or southern side of a vertex.
- A comprehensive way of handling self-loops by using dummy vertices and in-layer edges.
- A method to handle hierarchical ports subject to the port constraints that apply.

- An improved architecture for layout algorithms that dynamically adapts the algorithm to the layout tasks at hand.

▶ 1.3 Related Work

Sugiyama et al. propose the *layered approach* to drawing directed graphs [29]. The approach emphasizes direction by dividing the graph's vertices into numbered layers such that edges only go from lower layers to higher layers. Vertices in a single layer are drawn placed next to each other, with the different layers being drawn below one another. The approach works with three phases: *layer assignment*, which distributes the vertices into layers, *crossing reduction*, which changes the order of vertices in the different layers such that the number of edges crossings is minimized, and *vertex placement*, which determines the horizontal coordinates of vertices to minimize edge lengths. The layered approach assumes that graphs are acyclic and edges are drawn as straight lines. It is thus often extended by two additional phases: *cycle removal*, which makes an input graph acyclic, and *edge routing*, which routes edges when they are not supposed to be drawn as simple straight lines. KLay Layered is based on the layered approach with these two additional phases, but is extended in this thesis by introducing additional processing steps between the phases.

Eades et al. propose a greedy algorithm for cycle removal [7]. This algorithm is used by KLay Layered multiple times whenever graphs have to be acyclic. This is because the different phases of the algorithm often build auxiliary graphs.

Gansner et al. propose a layout algorithm based on the layered approach [12]. They propose a network simplex algorithm for layer assignment, on which the layer assignment phase of KLay Layered is based. Ports are considered, but only in a basic fashion: vertices are divided into several areas that edges may be connected to. In this thesis, ports are defined as dedicated connection points with the degree of liberty with respect to their placement controlled by port constraints. Contrary to the situation considered by Gansner et al., this may make it necessary to route edges around vertices.

For crossing reduction, heuristics such as the *barycenter heuristic* are often used: the order of vertices in one layer is computed from the order of connected vertices in the previous layer. This approach was extended to port based graphs by Spönemann et al., who considered the order of ports in the previous layer instead of the order of vertices [27]. This method is used in KLay Layered as well, with slight modifications. In this thesis, the barycenter heuristic is extended to support edges connecting vertices in the same layer.

Forster extends crossing reduction algorithms to support certain vertices whose relative order is fixed [10]. This thesis introduces applications for this algorithm, and extends it to *layout units*: sets of vertices whose members may not be mixed.

Barth et al. propose a fast algorithm to count the number of crossings during crossing reduction [1]. The algorithm only works for crossings between two layers, though. This thesis introduces two additional algorithms to supplement it that count certain kinds of crossings caused by edges that connect vertices in the same layer.

Sander proposes a method for vertex placement based on the concept of *linear segments*: sets of vertices that are placed along a straight line [24]. The concepts introduced in this thesis cause Sander's method to not work anymore by allowing situations in which it is not possible to place all vertices of a linear segment along a straight line. Therefore, this thesis modifies Sander's method to recognize such situations and split linear segments accordingly.

KLay Layered makes use of an edge routing algorithm proposed by Sander that routes edges orthogonally [25]. The algorithm is extended in this thesis to support different routing directions, and to reduce the spacing required between two layers.

Port constraints are considered by Spönemann [26]. In his method, edges are first routed locally to the respective vertices before global edge routing is applied. This thesis introduces a method that works without the local routing phase by using certain kinds of dummy vertices.

Klauske and Dziobek [17, 18] propose methods for handling ports whose positions depend on the size of their vertex. That kind of port constraint is not investigated further in this thesis.

Support for hierarchical ports with port constraints is also discussed by Spönemann [26]. In this thesis, a new method is introduced that tries to keep edges connected to hierarchical ports short.

To evaluate the quality of layouts, Purchase proposes a number of aesthetics criteria [21]. On ranking aesthetics criteria by importance, Purchase et al. provide empirical and deductive results that guided some of the design decisions for KLay Layered [20, 31].

Finally, an overview of graph drawing algorithms and aesthetics criteria is presented by Di Battista et al. [4]. Chapter 9, "Layered Drawings of Digraphs", is of particular interest to us as it gives an overview over the layered approach to graph drawing. However, only graphs without ports are considered.

▶ 1.4 Outline

With the introduction coming to a close, it is almost time to dive into the details of laying out data flow diagrams. But first, let us take a minute to see how the rest of this thesis is structured.

Chapter 2 provides the foundation for subsequent chapters. It starts by giving the mathematical definitions necessary for a treatment of graph drawing algorithms. This is followed by an overview of the algorithm that serves as our basis: the layered algorithm by Sugiyama et al. [29]. The chapter closes with an introduction of KLay Layered, as well as of its predecessor, which also allows to further elaborate on the goals of this thesis.

Chapter 3 starts the main part of the text and goes into great detail explaining all the extensions to KLay Layered. If you are interested in the theory, this is the chapter to read.

Once the theory is covered, Chapter 4 explains the implementation details, starting with a new structural unit for layout algorithms, the *intermediate processor*. All intermediate processors written for this thesis are then explained in detail, followed by a higher-level view of how the processors work together to implement the ideas presented in Chapter 3.

The performance of the algorithm has to be evaluated, which is what Chapter 5 is all about. Before we can take a look at results, we have to define how the quality of diagram layouts is measured. Since the algorithm is mainly meant for data flow diagrams, we have to settle on a collection of diagrams to test it with. We will find that the library of demo models shipping with Ptolemy is well-suited for that task; therefore, the chapter continues with some details on the work done to import Ptolemy models into our own data format. This is followed by an explanation of the evaluation process, after which we can at last turn our attention to the evaluation's results.

Chapter 6 closes this thesis. After a summary, I list a number of topics I am leaving for those succeeding me.

2

Background on Drawing Data Flow Diagrams

Look at me still talking when there's science to do. [...] I've experiments to run, there is research to be done!

— *GLaDOS, Portal*

In this chapter, we lay the groundwork for subsequent chapters. We start with the mathematical definitions necessary for a proper treatment of data flow diagram layout. With all definitions in place, we're all set for a quick tour of Sugiyama's layer-based algorithm. In the last section, we learn about the KLoDD algorithm before finally getting to KLay Layered, the algorithm extended in this thesis. The chapter closes with some more details on the goals of this work.

▶ 2.1 Introductory Remarks on Terminology

We start with the definition of a *directed, port-based graph.*

Definition 2.1. *A* directed, port-based graph *is a tuple $G = (V, E, P, \nu)$, where V is a finite set of* vertices *(also called* nodes*), $E \subseteq P \times P$ is a set of* edges *connecting the vertices, P is a finite set of* ports*, and $\nu : P \to V$ is a function mapping ports to their vertices. The set of vertices V may be further partitioned into a set of* dummy vertices *V_d introduced and removed again as part of the layout algorithm, and a set of* regular vertices *V_r.*

Note that the set of ports belonging to a vertex $v \in V$ is given by $\nu^{-1}(v)$. We now define some terms related to edges.

Definition 2.2. *An edge $e = (a, b) \in E$ is an* outgoing edge *of a and $\nu(a)$, and an* incoming edge *of b and $\nu(b)$. The ports a and b are said to be* adjacent *to one another, as are the vertices $\nu(a)$ and $\nu(b)$, and e is said to be* incident *to a, b, $\nu(a)$,* *and $\nu(b)$. a and $\nu(a)$ are called the* source *of e, while b and $\nu(b)$ are called its* target*. If $\nu(a) = \nu(b)$, then e is called a* self-loop*. We require $a \neq b$.*

We now need a way to refer to all the edges incoming to or outgoing from a port or vertex, with $\mathcal{P}(M)$ referring to the power set of a set M.

Definition 2.3. *We define a function $e_I : P \to \mathcal{P}(E)$ that maps ports to their incoming edges, and a function $e_O : P \to \mathcal{P}(E)$ that maps ports to their outgoing edges as follows:*

$$e_I(p) = \{(a, b) \in E \mid b = p\}$$
$$e_O(p) = \{(a, b) \in E \mid a = p\}$$

We extend the definitions to vertices as follows:

$$e_I(v) = \bigcup_{p \in \nu^{-1}(v)} e_I(p)$$
$$e_O(v) = \bigcup_{p \in \nu^{-1}(v)} e_O(p)$$

Similarly, we define the neighborhood of a vertex.

Definition 2.4. *The functions* succ $: V \to \mathcal{P}(V)$ *and* pred $: V \to \mathcal{P}(V)$ *map a vertex $v \in V$ to its* successors *and* predecessors *as follows:*

$$\operatorname{succ}(v) = \Big\{u \in V \mid \exists (a, b) \in E : a \in \nu^{-1}(v) \land b \in \nu^{-1}(u)\Big\}$$
$$\operatorname{pred}(v) = \Big\{u \in V \mid \exists (a, b) \in E : a \in \nu^{-1}(u) \land b \in \nu^{-1}(v)\Big\}$$

The function neighb $: V \to \mathcal{P}(V)$ *map a vertex $v \in V$ to its* neighbors *as follows:*

$$\operatorname{neighb}(v) = \operatorname{succ}(v) \cup \operatorname{pred}(v)$$

The following definitions relate to the number of edges connected to a port or vertex.

Definition 2.5. *The* indegree *of a port or vertex x is $|e_I(x)|$; its* outdegree *is $|e_O(x)|$. The* degree *of a port or vertex is the sum of its indegree and outdegree. The* net flow flow $: P \to \mathbb{N}$ *is defined as*

$$\operatorname{flow}(v) = |e_I(x)| - |e_O(x)|\,.$$

A vertex with an outdegree of 0 *is called a* source *of G. A vertex with an indegree of* 0 *is called a* sink *of G.*

We can now define *directed, port-based layered graphs.*

Definition 2.6. *A* directed, port-based layered graph *is a tuple $G = (V, E, P, L, \nu)$, where V, E, P, and ν have the same meaning as for directed, port-based graphs, and $L = (L_0, \ldots, L_k)$ is a finite sequence of finite sequences over V, called* layers*. We require each vertex $v \in V$ to be part of at most one layer. The i-th vertex of layer L_j is written as $L_j(i)$, and* $\operatorname{index}_{L_j}(v) = i$ *for $v = L_j(i)$.*

The following definition is necessary for our treatment of hierarchy.

Definition 2.7. *A* directed, port-based layered graph with hierarchical ports *is a directed, port-based layered graph* $G = (V, E, P, L, \nu)$ *with the definition of* ν *changed to* $\nu : P \to V \cup \{G\}$*. A port* $p \in P$ *with* $\nu(p) = G$ *is called a* hierarchical port*.*

Note that this definition does not allow for nested graphs; instead, we only allow one level of hierarchy, attaching the hierarchical ports to the graph itself.

Ports are usually drawn on the border of their respective vertices. We assume that vertices with ports are always drawn as rectangles and divide their border into the *north*, *south*, *west*, and *east* side. We now need a way to tell which side a port is assigned to.

Definition 2.8. *The function* side : $P \to \{\textsc{North}, \textsc{South}, \textsc{West}, \textsc{East}\}$ *maps a port to a side of its vertex.*

The usual case has ports with incoming edges on the western side, and ports with outgoing edges on the eastern side. But that need not be the case, as captured by the following definition.

Definition 2.9. *A port* $p \in P$ *is called a* regular port *if the following holds true:*

$$(\mathrm{side}(p) = \textsc{West} \wedge e_O(p) = \emptyset) \vee (\mathrm{side}(p) = \textsc{East} \wedge e_I(p) = \emptyset)\,.$$

It is called an inverted port *if the following condition is true:*

$$(\mathrm{side}(p) = \textsc{West} \wedge e_O(p) \neq \emptyset) \vee (\mathrm{side}(p) = \textsc{East} \wedge e_I(p) \neq \emptyset)\,.$$

Layout algorithms can have varying degrees of freedom in the way they place ports. These are captured in the following definition.

Definition 2.10. *The degree of freedom in the placement of ports is controlled by five different levels of* port constraints*:*

Free	*Ports may be placed freely on the border of their vertex.*
FixedSides	*The side of the vertex 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.*

The function

$$\mathrm{constr} : V \to \{\textsc{Free}, \textsc{FixedSides}, \textsc{FixedOrder}, \textsc{FixedRatio}, \textsc{FixedPos}\}$$

maps a vertex to the port constraints that apply to its ports.

The port constraints are based on those defined by Spönemann [26], with his FreePorts constraint renamed to Free, FixedPorts renamed to FixedPos, and the added FixedRatio constraint. There are certainly more possible constraints, but we have found these five to be enough for our applications.

▶ 2.2 The Layered Approach to Graph Drawing

Data flow diagrams can be interpreted as directed graphs, suggesting layout algorithms for directed graphs as a good starting point to lay out data flow diagrams. A classic approach for the layout of directed graphs was proposed in 1981 by Sugiyama, Tagawa and Toda: the *layer-based approach* [29]. (This was originally called the *hierarchical approach*, with hierarchy referring to partitioning the vertices into different layers; however, our notion of hierarchy is different, which is why we use a different name.) Therein, the set of vertices of a directed, acyclic graph is partitioned into layers, with all elements of a layer placed one below the other in the drawing.[1] The assignment of vertices to layers is such that the source of an edge is always placed in a layer left of the edge's target. With edges then only pointing to the right, this kind of layout emphasizes the flow direction in a graph, and is thus very well suited to data flow diagrams.

Figure 2.1 shows a tree laid out using two different layout algorithms. A force-based algorithm, which interprets edges as springs exerting forces on vertices and computes a balanced configuration, does not put special emphasis on the graph's tree structure. A layout algorithm based on the layered approach, on the other hand, makes this structure very apparent and keeps a consistent reading direction from left to right.

Sugiyama, Tagawa and Toda proposed a three-phase structure for layout algorithms, consisting of a *layer assignment* phase, a *crossing reduction* phase, and a *vertex placement* phase.

During layer assignment, layers are created and all vertices are assigned to them such that edges only go from left to right. This is captured in the following definition:

Definition 2.11. *A* standard layering *is a layering for which the following holds:*

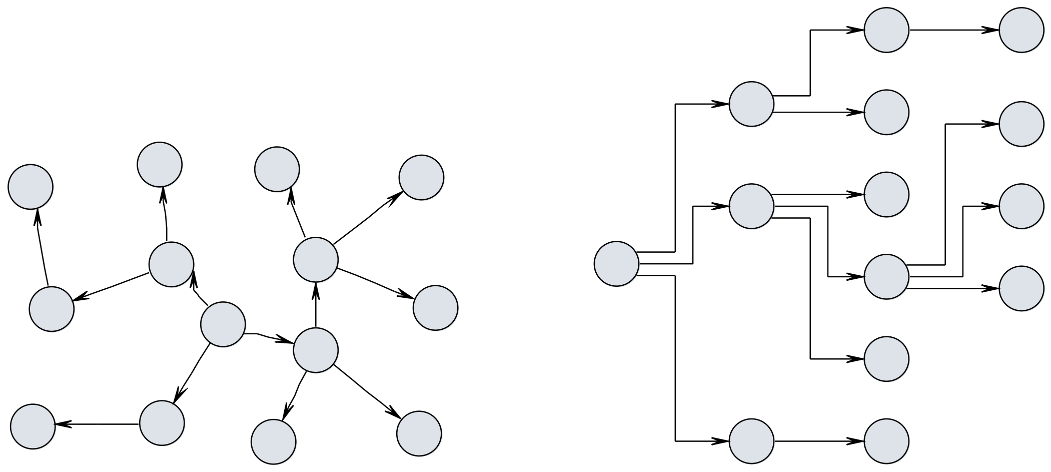
$$\forall\, 0 \leq i, j < |L| \;\forall\, v \in L_i, w \in L_j : (v, w) \in E \Rightarrow i < j.$$

A layering is said to be *proper* if edges only connect vertices placed in adjacent layers.

Definition 2.12. *A* proper layering *is a layering for which the following holds:*

$$\forall\, 0 \leq i, j < |L| \;\forall\, v \in L_i, w \in L_j : (v, w) \in E \Rightarrow i + 1 = j.$$

[1]While data flow diagrams are usually laid out from left to right, graph drawing literature commonly assumes top-down or bottom-up layouts, hence the term “upward drawing” is often used. In this thesis, we go with the former convention.



(a) A force-based layout algorithm does not emphasize the graph's hierarchical structure.

(b) A layer-based layout algorithm makes the graph's hierarchical structure much more apparent.

Figure 2.1. Two layout algorithms applied to a tree.

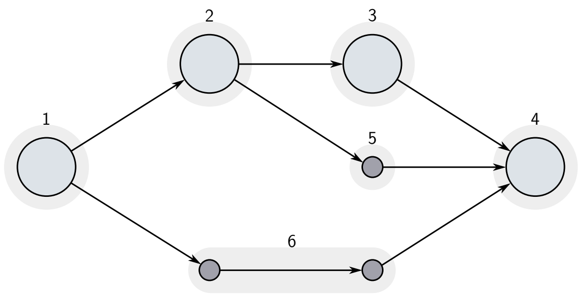
Since this is not always possible, edges spanning more than two layers, called *long edges*, are split by inserting *edge dummy vertices*. Dummy vertices are vertices inserted by the algorithm, and removed again before the algorithm has finished.

Different algorithms for layer assignment exist, all emphasizing different goals. For one, an algorithm might try to minimize the number of layers, which can be done in linear time using topological numbering. Another algorithm might try to minimize the maximum number of vertices per layer. Trying to minimize both at the same time was proven to be NP-complete by Eades and Sugiyama [8]. In practice, we apply a method that minimizes the length of edges [12].

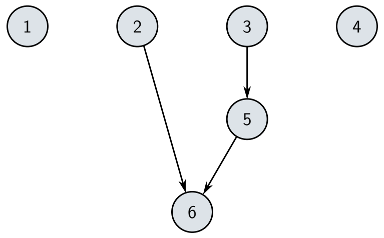
The crossing reduction phase is concerned with minimizing the number of edge crossings, determined by the order of vertices in the layers. Sadly, Garey and Johnson have proven that solving this problem optimally is NP-complete. Even worse, this is true even for only two layers [14], so we are left with using heuristics. The approach usually taken is to sweep through the layers, keeping the order of vertices for layer L_i fixed and reordering L_{i+1}. Once a sweep is complete, another is started in the reverse direction, this time keeping the layer L_{i+1} fixed and solving the problem for L_i. This is repeated until the number of crossings does not decrease anymore.

As far as the heuristics are concerned, the barycenter method and the median method are popular choices. Both methods work by assigning ranks $r(w)$ to the vertices w of the fixed layer, dependent on their relative position (for the moment, we set $r(p) \leftarrow r(\nu(p))$ for $p \in P$). Assuming that L_i is the fixed layer, and L_{i+1} is the free layer, the barycenter method assigns a vertex $v \in L_{i+1}$ the following value:

$$\mathrm{val}_b(v) = \frac{1}{|e_I(v)|} \sum_{(w,v) \in e_I(v)} r(w)$$



(a) The diagram, with long edges split by dummy vertices. The gray boxes in the background identify the different linear segments the vertices are part of.



(b) The segment ordering graph. Vertices represent the different linear segments, with an edge (v, w) inserted if, in some layer, a vertex from linear segment v is placed above a vertex from linear segment w.

Figure 2.2. A diagram with linear segments marked, and the diagram's segment ordering graph. Linear segments are one way of calculating a valid vertex placement.

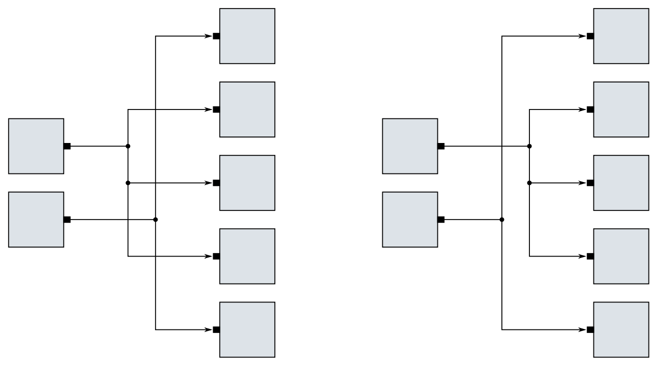
The median method chooses the middle rank value:

$$\mathrm{val}_m(v) = r(\bar{u}),$$

given $e_i(v) = \{(u_1, v), \ldots, (u_h, v)\}$, $r(u_1) < \ldots < r(u_h)$, and $\bar{u} = u_{\lfloor h/2 \rfloor}$. As Juenger and Mutzel show, the barycenter heuristic is generally superior to the median heuristic [15].

Another approach to crossing reduction is to solve the problem exactly by converting it into a linear integer program [15] or a semidefinite program [3]. This is viable for small graphs, but is no option for general-purpose algorithms that do not impose a maximum on number of vertices of their input graphs.

Having decided on the horizontal coordinates of the vertices in the layer assignment phase, the task of the vertex placement phase is to assign vertical coordinates. The goal is to place the vertices in a way that minimizes edge lengths, subject to two constraints: long edges are to be kept straight, meaning that the dummy vertices inserted to split a long edge e are assigned the same vertical coordinate; and the order of vertices as calculated during crossing reduction must be respected. To that end, Sander introduces the concept of *linear segments*: non-empty sets of vertices whose elements are laid out along a straight line [24]. According to the first goal, all dummy vertices that split a long edge e form a linear segment, as does every regular vertex. To respect the second goal, Sander's algorithm creates a *segment ordering graph* whose vertices represent the linear segments, with an edge between vertices v and w if, in some layer L_i, a vertex in segment v directly precedes a vertex in segment w according to the order calculated during crossing reduction. The segment ordering graph is guaranteed to be acyclic, and its topological ordering is then used to place the vertices in the linear segments. Figure 2.2 shows an example of a diagram with its linear segments, together with its segment ordering graph.


(a) A routing with four edge crossings.

(b) A routing with only one edge crossing.

Figure 2.3. Orthogonal routing of edges between two layers. The order of the vertical segments determines the number of edge crossings. Contrary to the graph in Figure 2.2a, the graphs shown here are port-based.

The drawing of a graph is produced by executing the three phases and removing the dummy vertices, adding a bend point at each dummy vertex's position.

Extensions to the Layered Approach

While Sugiyama, Tagawa and Toda have assumed graphs to be acyclic, this assumption often doesn't hold with data flow diagrams. Therefore, a *cycle removal* phase that reverses edges to remove a graph's cycles, if any exist, is usually added before the layer assignment phase. Unfortunately, this problem is NP-complete as well, since it is equivalent to the *feedback arc set problem* [13]. However, good heuristics exist [7], with some even applying knowledge about the data flow language to particularly reverse those edges that the user would classify as feedback edges [17]. Of course, such approaches are not possible for algorithms not tailored to a specific language.

As already hinted at in Section 1.1, edges in data flow diagrams are usually drawn orthogonally, with edge segments running either horizontally or vertically. This task is complex enough to add a fifth phase to the algorithm, the *edge routing* phase; its task is to insert vertical segments to edges that cannot be drawn as straight horizontal lines. As shown in Figure 2.3, the order of the vertical segments influences the number of edge crossings and thus needs to be chosen well. As proposed by Sander, this can be done by constructing a *segment crossing graph* whose vertices represent the vertical segments, with an edge going from vertex v to vertex w if placing the vertical segment represented by v left of the segment represented by w would produce edge crossings; weights are assigned to edges depending on the resulting number of crossings [25]. The optimal order of vertical segments is given by the topological order of the segment crossing graph, which requires cycles to be removed prior to the ordering.

▶ 2.3 The KLoDD and KLay Layered Algorithms

The Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[2] is a research project about enhancing the graphical model-based design of complex systems [11]. The basic idea is to build upon automatic layout algorithms to provide pragmatic editing capabilities, ultimately leading to higher productivity by freeing the user of layout-related tasks and by providing advanced editing features. Dynamic views of parts of a complex system, for instance, can be created in a matter of seconds with automatic layout, but would take a lot of work to lay out if done manually.

KIELER is implemented in Java as a collection of plug-ins for the Eclipse[3] platform, which was originally created by IBM in 2001 and is now maintained by an open source community; the platform enjoys widespread industrial use. Since Eclipse is an open platform, a given installation can provide different graphical editors, giving rise to the potential problem of having to add automatic layout capabilities to each one separately. To this end, KIELER introduces a generic framework for layout to connect editors and layout algorithms, the KIELER Infrastructure for Meta Layout (KIML).

Figure 2.4 shows the basic architecture of KIML and its central data structure, the `KGraph`. Since graphical editors usually operate on different data structures, KIML provides the `KGraph` as a means for abstracting away from the concrete structure used by an editor. Transforming an editor's internal data structure into a `KGraph` is thus the first step in automatic layout. The second step consists of transforming the `KGraph` into the data structure used by a given layout algorithm, since layout algorithms potentially use different data structures as well. Once the layout algorithm has calculated its result, the layout information need to be properly attached to the `KGraph`, which happens in the third step. The fourth and final step consists of applying these layout information to the graphical editor's internal data structure. Connecting a new editor to KIML—and thus giving it access to all connected layout algorithms at once—is a matter of implementing steps one and four, while connecting a new layout algorithm—and thereby enabling it for all connected editors at once—is a matter of implementing steps two and three.

In the world of Eclipse, most graphical editors build upon the Graphical Modeling Framework (GMF), with the new Graphiti framework gaining in popularity due to better performance and a simpler architecture. Because of this, it is enough to connect these two frameworks to KIML to add automatic layout capabilities to all graphical editors building on them. On the layout algorithm side, connections to several popular open source graph layout libraries exist, among them the Open Graph Drawing Framework (OGDF)[4] and Graphviz.[5]

[2]http://www.informatik.uni-kiel.de/rtsys/kieler
[3]http://www.eclipse.org
[4]http://www.ogdf.net
[5]http://www.graphviz.org

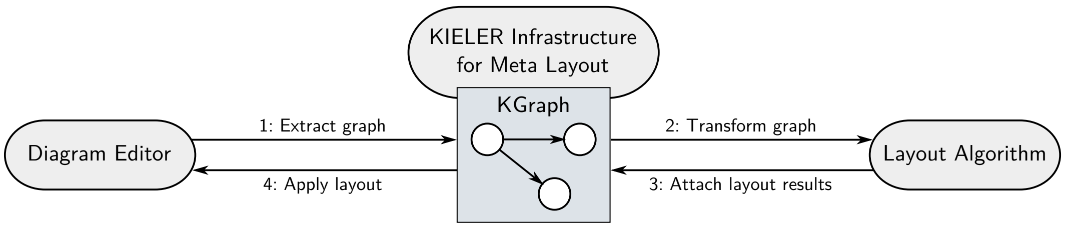


Figure 2.4. The architecture of KIML. A layer of abstraction between graphical editors and layout algorithms facilitates to allow a new editor access to automatic layout, or to add a new layout algorithm to all graphical editors tied to KIML.

One part of the KIELER project is concerned with the development of custom layout algorithms. At the time of writing, two such algorithms exist, and in the rest of this section we take a look at both of them.

The KLoDD Algorithm

KIELER Layout of Dataflow Diagrams (KLoDD) was the first foray into a custom layered layout algorithm tailored specifically to data flow diagrams. It was developed by Miro Spönemann as part of his diploma thesis [26] and consisted of the five phases described in Section 2.2. For each phase, existing approaches were extended or heavily modified, with a special emphasis on performance.

A major contribution was the introduction of port constraints, which control how much freedom the algorithm has in placing the ports of a vertex. The two most important constraints allow the algorithm either full or no flexibility; two further constraints mediate between these two extremes. We examine this in greater detail in Chapter 3.

The second major contribution concerns the treatment of hierarchical ports. As mentioned in Section 1.2, diagrams can be displayed with multiple levels of hierarchy at once, with edges connecting vertices inside a given level to the level's hierarchical ports requiring special consideration. In KLoDD, the common cases of hierarchical edges coming from the left and going to the right were handled as one would expect, while the other, less common cases received lesser treatment. A more comprehensive solution is the topic of Section 3.6.

One of the major shortcomings of KLoDD is that requirements were added during development that its architecture was not built for from the start. The algorithm became difficult to maintain, and hard to extend. For the sake of an example, the layout direction produced by KLoDD is configurable: a diagram can be laid out in a way that the predominant number of edges points either rightward, leftward, up or down. This distinction becomes important at many places in the code, resulting in quite a few places that read like the following:

```
if (layoutDirection == Direction.DOWN) {
    // 60 lines of code
} else {
    // 60 almost identical lines of code,
        // differing only in details
    }
```

The code is cluttered, and otherwise simple changes have to be applied to many places at once, increasing the probability of bugs.

A second shortcoming also relates to added requirements. Some of the solutions developed for KLoDD were added as more of an afterthought instead of being considered right from the start. They usually work well for the common case, but leave something to be desired in the areas of cleanliness and robustness.

In Chapter 3, I frequently start a section with a short description of how things were handled by KLoDD before introducing my new approaches.

The KLay Layered Algorithm

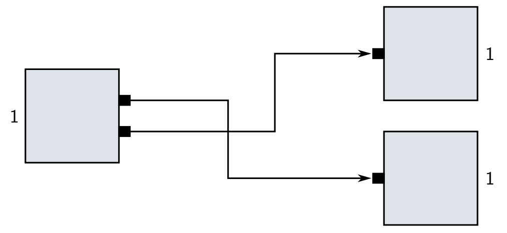
There were a few lessons to be learned from the development of KLoDD. First, the implementation of a complex algorithm greatly benefits from a simple, clear, and flexible architecture. Second, different cases that influence only details, but not the overall approach, should be abstracted from to keep the implementation from becoming cluttered and hard to maintain and extend. And third, for layout algorithms targeted at data flow diagrams, performance is not as big an issue as was initially assumed. This is because data flow diagrams in the real world are way too small for performance to be a problem. As a consequence, the implementation places a higher emphasis on a good architecture than on optimization.

KLay Layered was started as the successor to KLoDD, with the goal of addressing all the problems described above. As its predecessor, it follows the classical five-phase structure and operates on its own internal data structure since the `KGraph` has no concept of layers. Each phase can have different implementations the user can choose between. As an example, the edge routing phase has an implementation based on splines and another implementation based on orthogonal edge routing. All the while, the tasks of different phases are clearly defined.

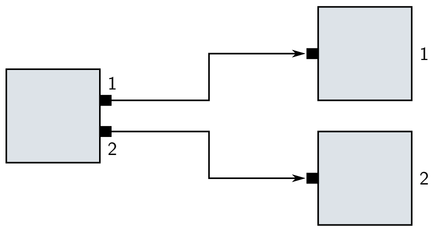
While in KLoDD existing approaches to different phases were extended or heavily modified, the first version of KLay was based almost entirely on faithful implementations of existing approaches, only modified where required for basic port support. What follows is a brief overview of the more important implementations for the different phases.

The cycle removal implementation is based on the GreedyCycleRemoval algorithm given by Eades, Lin and Smyth [7], which provides a very simple, but satisfactory heuristic.

Two implementations exist for the layer assignment phase. One is based on the the LongestPathLayering algorithm explained by Di Battista et al. [4]; while its time complexity is linear in the number of edges, the layerings produced often suffer from high maximum numbers of vertices in a layer. The other implements the NetworkSimplexLayering algorithm as presented by Gansner et al. [12]; this algorithm produces good results, at the expense of time complexity.



(a) Without special support for ports, the barycenter value computed for the two vertices to the right is the same. Some kind of tie-breaking method must ensure that the vertices don't overlap, possibly resulting in an unnecessary edge crossing.



(b) With support for ports, the two vertices to the right get different barycenter values. No tie-breaking method is necessary, and the crossing can be avoided.

Figure 2.5. This is an example of a port-based graph where the usual barycenter approach fails. The numbers are the rank values assigned to or calculated for the respective vertex or port.

The implementation of the crossing reduction phase is based on a layer-by-layer sweep with a barycenter heuristic, with modifications to support ports. While the original approach given by Di Battista et al. [4] uses rank values of connected vertices as the basis for the barycenter calculations, the modified approach uses rank values of connected ports (note that the explanation in Section 2.2 already takes this into account). As Figure 2.5 illustrates, this modification is especially important in the common case where different vertices are connected to different ports that belong to a single vertex. Without the modification, it is unclear how to order the vertices; with the modification, the order becomes obvious.

The vertex placement phase has one implementation based on the linear segments approach by Sander [24], as explained in Section 2.2.

There are two main implementations for the edge routing phase. The first implements the simple edge routing, with edges drawn as straight lines. The second implements orthogonal edge routing as proposed by Sander [25], giving results like those shown in Figure 2.3.

When work on this thesis began, KLay Layered couldn't replace KLoDD yet. There was only very basic support for ports (neither inverted ports nor northern and southern ports were supported), and hierarchical ports were not supported at all. In both areas, the goal was to develop and implement new solutions, with emphasis on not just porting over solutions developed for KLoDD, but to think about new and better methods. Regarding the implementation, I was to think about whether a five-phase structure is sufficiently flexible to handle the complex requirements for data flow diagrams.

3

Optimizing Automatic Layout for Data Flow Diagrams

The Enrichment Center regrets to inform you that this next test is impossible. Make no attempt to solve it.

— *GLaDOS, Portal*

This chapter introduces the theory behind the modifications to KLay Layered, organized around four main topics: handling inverted ports, handling northern and southern ports, handling self-loops, and handling hierarchical ports. Each of these has its own section, with necessary preliminary considerations prepended as required. The chapter closes with a few more sections about further enhancements to the algorithm.

▶ 3.1 In-Layer Edges

Before we can come to the topic of inverted ports we need to concern ourselves with *in-layer edges*: edges that connect two vertices $v, w \in L_i$ for some layer $L_i \in L$. Why we actually need them is covered in the next section; this section is committed to describing how they can be supported.

Our definition of a proper layering as given in Definition 2.12 does not allow in-layer edges, requiring us to change the definition a little:

Definition 3.1. *A* proper layering with in-layer edges *is a layering for which the following holds:*

$$\forall\, 0 \leq i, j < |L| \;\forall\, v \in L_i, w \in L_j : (v, w) \in E \Rightarrow j \in \{i, i+1\}$$

For an in-layer edge $(v, w) \in E$, we require either v or w to be a dummy vertex.

We now have three definitions for layerings: the standard layering as given in Definition 2.11, the proper layering as given in Definition 2.12 and the proper layering with in-layer edges defined above. Before going further, it is important to understand at which point in the algorithm which kind of layering can be assumed. During cycle removal, the graph is not even layered yet, so none of the definitions applies. During layer assignment, a standard layering is produced, which is turned into a proper layering, if necessary. Only then can modifications to the graph lead to a proper layering with in-layer edges.

This has a potential impact on crossing reduction, vertex placement, and edge routing. While the algorithms used in KLay Layered for the latter two phases do not need much modification, the crossing reduction algorithm does.

Crossing Reduction

Since the approach to crossing reduction used in KLay Layered is based on a layer-by-layer sweep, it is enough to consider only two layers $L_i, L_{i+1} \in L$ at a time. Without loss of generality, let L_i be the fixed layer, and let L_{i+1} be the free layer. To work out the order of vertices in L_{i+1}, the algorithm assigns a rank value to all ports of L_i according to their relative order, and then calculates the barycenter for all vertices in L_{i+1}, considering the sources of their incoming edges. A problem arises when the source of an incoming edge is also in L_{i+1} and thus has not been assigned a rank value—in other words, when the incoming edge is an in-layer edge.

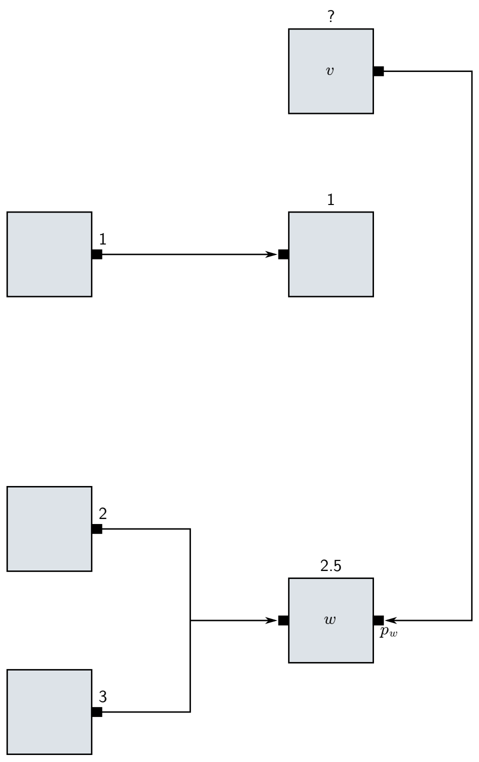
Figure 3.1 shows such a case. The barycenter of v would be computed using the ranks of the connected ports in layer L_i, of which there are none, plus the rank of p_w, which has not been assigned.

A first solution would be to simply ignore the in-layer edge, but this could lead to v and w being placed too far apart, as in Figure 3.1a.

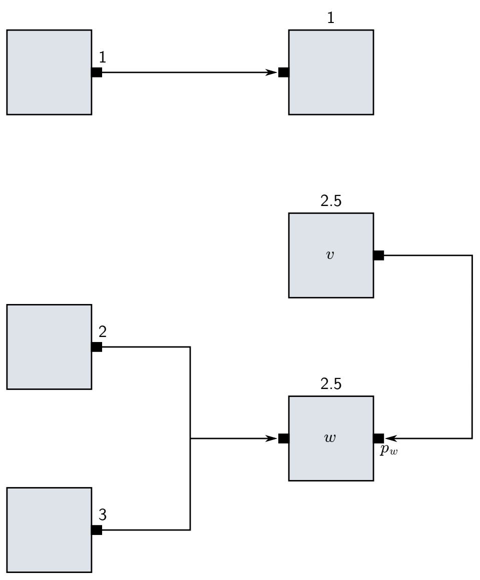
A second solution would be to simply assign a rank to p_w according to some criteria. However, since rank values are supposed to reflect the order of connected ports, the temporary rank of p_w would have no meaning: the rank depends on the order of ports in L_i, but v is in L_{i+1}.

To arrive at a viable solution, we have to think about what kind of vertex v and w can actually be. If v is a regular vertex, w cannot be a regular vertex as well or it would have been placed in another layer during layer assignment. If it is not a regular vertex, w must be a dummy vertex that will be removed at some later time. If that is the case, why not pretend that it does not exist in the first place? Once the rank of p_w is required for the computation of the barycenter of v, we can go ahead and compute the barycenter of w, and then use that in the computations for v, as illustrated in Algorithm 3.1. In effect, we have just made w invisible to v, and arrived at a barycenter value that makes perfect sense since v will be connected to the predecessors of w later anyway. The improvement is shown in Figure 3.1b. A similar argument can be made if w is a regular vertex, which then implies that v must be a dummy vertex.

While this takes care of the problem of ordering vertices, there is another problem to be solved: cross counting needs to include in-layer edges as well. The algorithm implemented in KLay Layered, proposed by Barth, Mutzel and Jünger [1], only



(a) Since p_w has no rank, its impact on the barycenter calculation for v is unclear without special support for in-layer edges, possibly leading to an unfortunate placement of v.



(b) With support for in-layer edges, w is treated as not being there at all, leading to a much better placement of v.

Figure 3.1. The crossing reduction phase requires special support for in-layer edges to still produce satisfactory results. The numbers denote the assigned rank and calculated barycenter values.

counts crossings between two layers. It has to be complemented by another algorithm that adds the number of in-layer crossings.

This can be done in two sweeps per layer: the first sweep numbers the western ports and the eastern ports that have incident edges from top to bottom according to their number of incident edges. The second sweep iterates over those ports again, looking for in-layer edges. Since we only allowed them to connect two eastern or two western ports, the difference of the port numbers we just computed minus one gives the highest number of crossings that can be caused by the in-layer edge. Figure 3.2 shows an example of this algorithm at work.

The approach works well, but only provides an upper bound on the number of edge crossings. However, this is consistent with the cross counting algorithm by Barth, Mutzel, and Jünger when used with an orthogonal edge routing algorithm: routing edges orthogonally may result in less crossings than calculated by that algorithm because edges may share line segments and thus contribute only one crossing.

```
function CalcBarycenter($L_k \in L$, $v \in L_k$, $S \subseteq L_k$)
    $edges \in \mathbb{N} \leftarrow 0$
    $ranksum \in \mathbb{R} \leftarrow 0.0$
    $S \leftarrow S \cup \{v\}$

5   for all $w \in \mathrm{pred}(v)$ do
        if $w \notin S$ then
            if $w \in L_k$ then
                $(wedges, wranksum) \leftarrow$ CalcBarycenter($L_k, w, S$)
                $edges \leftarrow edges + wedges$
10              $ranksum \leftarrow ranksum + wranksum$
            else
                $edges \leftarrow edges + 1$
                $ranksum \leftarrow ranksum + \mathrm{rank}(w)$
            end if
15      end if
    end for

    return $(ranksum, edges)$
end function
```

Algorithm 3.1. Calculating barycenters with in-layer edges. The set S contains the vertices whose barycenter is being or has already been calculated to avoid endless loops in the presence of in-layer circles. The actual barycenter can be calculated from the algorithm's return value with the formula $ranksum/edges$. This rendition of the algorithm assumes a left-to-right sweep. During a right-to-left sweep, the for loop in line 5 would iterate over the vertex's successors instead of its predecessors.

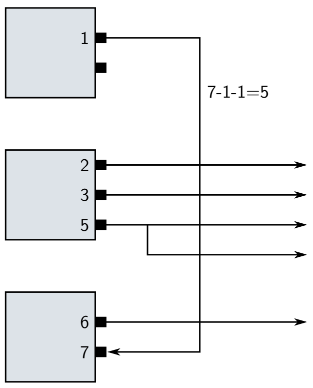
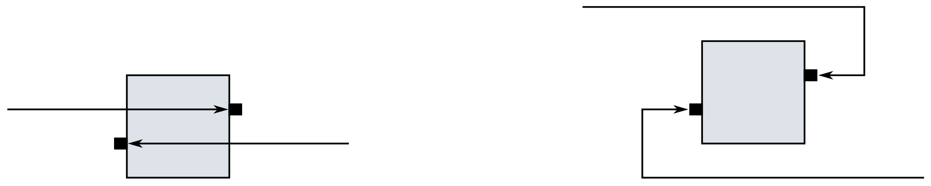


Figure 3.2. An example of the in-layer cross counting algorithm at work. The ports with incident edges are numbered from top to bottom. To find the number of crossings caused by the in-layer edge, the lower port number is subtracted from the higher port number, and the result is decreased by 1.



(a) Without special treatment, incident edges are routed through the port's vertex.

(b) With special treatment, incident edges are routed around the port's vertex.

Figure 3.3. Inverted ports are ports on the western side with outgoing edges, and ports on the eastern side with incoming edges. They require special treatment to keep edges from crossing vertices.

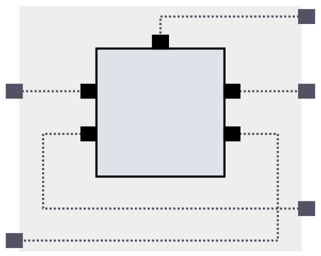


Figure 3.4. In KLoDD, a preliminary vertex-local edge routing phase took care of transforming non-regular cases into the regular case (edges coming in on the western side and going out on the eastern side) just before vertex placement. The dashed lines indicate the vertex-local edge routing, with the gray background indicating the vertex and its ports as viewed by the following phases.

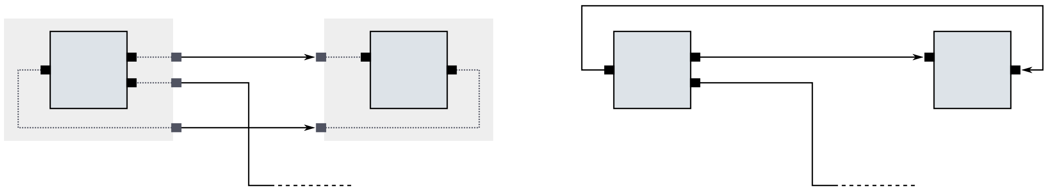
▶ 3.2 Inverted Ports

A considerable share of a diagram's readability stems from vertices not overlapping, and from edges staying clear of vertices. The latter part is straightforward if edges only go from eastern to western ports, but that changes once inverted ports are involved: as Figure 3.3a illustrates, not handling inverted ports results in edges crossing vertices. The expected solution is to route edges around the vertex of inverted ports, as shown in Figure 3.3b. This requires special consideration and is the topic of this section.

Previous Approaches

The vertex placement and edge routing phases of KLoDD expected vertices to have edges entering on the western side and exiting on the eastern side. As we just saw, this is not the case with inverted ports, and in fact northern and southern ports pose a similar problem which we come to in Section 3.4. To fix this, KLoDD adds a new phase just before vertex placement that does some preliminary, vertex-local edge routing. As illustrated in Figure 3.4, the result is vertices with edges coming in on the western side and going out on the eastern side—the regular case. Edges incident to inverted ports are always routed below the vertex.

The approach has the obvious benefit of keeping later phases simple by letting



(a) Edges connected to inverted ports are always routed below their vertex in KLoDD, which in this case results in an unnecessary edge crossing.

(b) With the approach implemented in KLay Layered, the crossing reduction phase has a higher degree of freedom in routing such edges, in this case getting rid of the crossing entirely.

Figure 3.5. An example of the limitations of the vertex-local edge routing phase implemented in KLoDD.

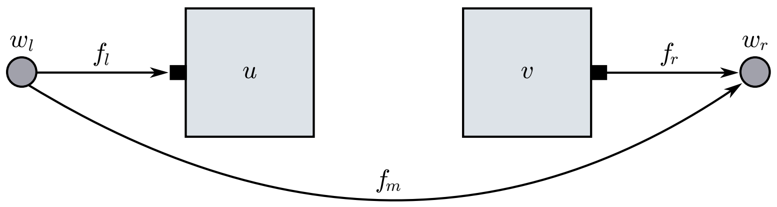


Figure 3.6. Klauske's method works by inserting two dummy vertices, w_l and w_r, and by replacing the original edge by three new edges, f_l, f_m, and f_r. Since f_m spans several layers, it is later split by inserting additional edge dummy vertices not shown here.

them assume the regular case of edge routing. However, there are two disadvantages. First, such vertex-local edge routing introduces unnecessary edge bends. The routing produced in Figure 3.4 already has five bends, and it is unrealistic to assume that the edges will be free of further edge bends after edge routing is finished. And second, the approach can easily result in unnecessary edge crossings because it does not take surrounding layers into account. Figure 3.5 shows an example in which the approach implemented in KLoDD leads to an unnecessary crossing while the approach to be presented here produces none.

Klauske proposes an approach involving two dummy vertices [18]. His method solves a specific scenario in which an edge runs from a western port to an eastern port, but can easily be adapted to other cases involving inverted ports. Let $e = (u, v) \in E$ be such an edge. Klauske's method inserts two dummy vertices w_l and w_r, and replaces e by three new edges $f_l = (w_l, u)$, $f_m = (w_l, w_r)$, and $f_r = (v, w_r)$, as shown in Figure 3.6. The edge connecting the new dummy vertices, f_m, is a long edge and is thus split by further edge dummy vertices.

Klauske's approach solves the problem of inserting too many bend points. The vertex-local edge routing found in KLoDD is replaced by dummy vertices that are properly considered during crossing reduction and vertex placement, inserted just before layer assignment. However, without further consideration w_l and w_r both take up space in their respective layers that could well have been used for regular vertices.

This is unfortunate since the dummy vertices are removed later anyway, and may lead to slightly odd layouts and even additional edge bends. The problem can be avoided by changing the vertex placement algorithm accordingly, but depending on the algorithm used this may not be easy. More importantly, the edges f_m and f_r are considered during crossing reduction even though they are later removed again and do not actually contribute any crossings, possibly leading to undesirable results.

I therefore propose a different approach.

A Different Approach

Let $e = (u, v) \in V$ again be an edge going from a western port to an eastern port. Similar to Klauske, I propose to insert two dummy vertices w_l and w_r, though not before, but after layer assignment. Indeed, w_l is inserted into the layer of u, and w_r is inserted into the layer of v. The original edge e is then replaced as explained previously by three new edges f_l, f_m, and f_r, with f_l and f_r requiring later phases to support in-layer edges.

The approach has several advantages. First, w_l and w_r can be treated as edge dummy vertices, simplifying later stages of the algorithm: upon removal, the incident edges are merged and bend points are added. Second, the crossing reduction phase does not have to consider edges between two layers that are removed later anyway. Third, w_l and w_r do not take up space that could be used for regular vertices. Since they are placeholders for a long edge, the space could not be used for vertices anyway. And fourth, our orthogonal edge routing algorithm can be used without further modification to correctly route the edges that would otherwise require special treatment.

Whether these advantages outweigh the disadvantage of having to adapt later phases to in-layer edges depends on the algorithms used. Note however that support for in-layer edges does not just benefit the treatment of inverted ports, but can also be used to solve other problems, such as self-loops (see Section 3.5).

▶ 3.3 Constrained Crossing Reduction

Usually, the order of vertices in their respective layers can be changed at will during crossing reduction. That said, there are cases when the order is subject to certain constraints. In this section, I introduce two kinds of constraints and explain how crossing reduction algorithms can be augmented to respect them. The method of handling northern and southern ports, presented in Section 3.4, is an example that makes use of the constraints presented here.

The first is called a *vertex successor constraint*: it restricts the relative order pairs of vertices may appear in. It was proposed by Forster [10] and can be defined as follows:

Definition 3.2. Vertex successor constraints *are defined by a set $C \subseteq V \times V$ of ordered pairs of vertices (v, w), and are satisfied if the following holds:*

$$\forall\, 0 \leq k < |L|\ \forall\, 0 \leq i, j < |L_k| : (L_k(i), L_k(j)) \in C \Rightarrow i < j.$$

In a left-to-right drawing, this would cause $L_k(i)$ to be placed above $L_k(j)$ (hence the name 'vertex *successor* constraint').

The second constraint is called a *layout unit constraint*: vertices may be assigned to layout units whose vertices may not be mixed. Formally, this is covered by the following definition:

Definition 3.3. Layout unit constraints *are defined by a function* lunit : $V \to V \cup \{\perp\}$*, and are satisfied if the following holds:*

$$\begin{aligned}\forall\, 0 \leq k < |L|\ \forall\, 0 \leq i < |L_k| : \quad & \mathrm{lunit}(L_k(i)) = \perp \vee (\neg\exists\, 0 \leq j, l < |L_k| : \\ & \quad \mathrm{lunit}(L_k(j)) \neq \perp \\ & \quad \wedge\ \mathrm{lunit}(L_k(j)) \neq \mathrm{lunit}(L_k(i)) \\ & \quad \wedge\ \mathrm{lunit}(L_k(j)) = \mathrm{lunit}(L_k(l)) \\ & \quad \wedge\ j < i < l).\end{aligned}$$

That is, the constraint is always satisfied for a vertex $L_k(i)$ if $L_k(i)$ is not assigned to a layout unit; if, on the other hand, $L_k(i)$ is assigned to layout unit v, it must not be placed between two other vertices $L_k(j)$ and $L_k(l)$ that are assigned to another layout unit $w \neq v$ with $w \neq \perp$. There are two things to be aware of: first, layout units are identified by a vertex which will usually be a regular vertex rather than a dummy vertex; second, vertices not assigned to layout units may be freely placed anywhere in their layer.

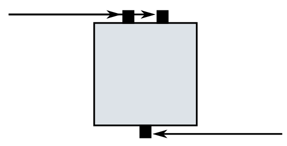
With Forster's algorithm, having crossing reduction observe vertex successor constraints is easy. Once the order of the vertices in a layer is established, Forster's algorithm can be run to look for any violated constraints. The result is an order that satisfies all vertex successor constraints.

The problem of satisfying layout unit constraints is slightly more difficult, but can indeed be reduced to the problem of satisfying vertex successor constraints by inserting an additional step right before the execution of Forster's algorithm. For a layer $L_k \in L$, let $V_u \subseteq L_k$ be the sequence of vertices that identify layout units, ordered by their relative order in L_k from top to bottom:

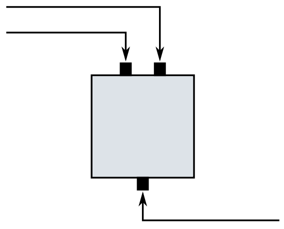
$$v \in V_u \Leftrightarrow \exists\, 0 \leq i < |L_k| : \mathrm{lunit}(L_k(i)) = v.$$

For all $(V_u(i), V_u(i+1))$, we insert vertex successor constraints from all vertices belonging to layout unit $V_u(i)$ to all vertices belonging to layout unit $V_u(i+1)$. Formally, we require that the following holds for each layer $L_k \in L$:

$$\forall\, 0 \leq i < |V_u| - 1\ \forall\, v \in \mathrm{lunit}^{-1}(V_u(i)), w \in \mathrm{lunit}^{-1}(V_u(i+1)) : (v, w) \in C.$$



(a) Without special treatment, edges connected to northern or southern ports may overlap.



(b) With special treatment, edges connected to different ports are properly separated.

Figure 3.7. Edges connected to northern or southern ports require special treatment. Without that, the routing looks unpleasant at best or introduces ambiguity in the worst case.

▶ 3.4 Northern and Southern Ports

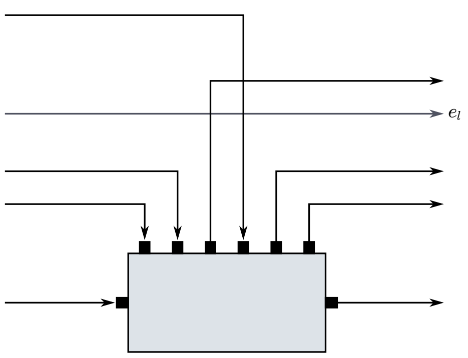
Similar to inverted ports, edges connected to northern and southern ports cannot be routed like regular edges. Consider the example shown in Figure 3.7. With the edge connected to the southern port of the vertex, the problem is merely one of aesthetics; the edges connected to the two northern ports, however, overlap and thereby introduce ambiguity. As Figure 3.4 suggests, KLoDD solved this problem by a vertex-local edge routing phase, which we wanted to avoid in KLay Layered for reasons already explained. The approach to be proposed here is thus again based on dummy vertices.

Let $v \in L_k$ be a regular vertex in some layer $L_k \in L$ with northern and southern ports. The general idea of our approach is to add special *bend dummy vertices* to L_k and to reconnect all edges incident to northern or southern ports of v to the dummy vertices. Bend dummy vertices differ from normal edge dummy vertices in how bend points are inserted when they are removed. The vertices created for northern ports are placed above v, and the vertices created for southern ports are placed below v.

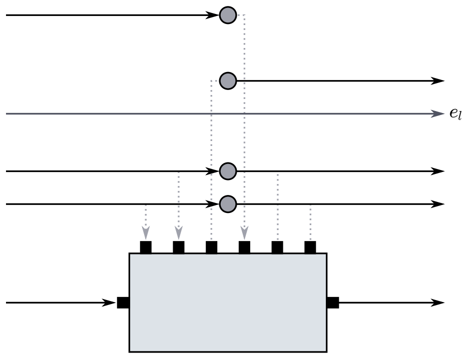
The rest of this section goes into the details of how this works.

Creating Dummy Vertices

Figure 3.8 shows a regular vertex with northern ports and with incident edges routed as our approach would do it. There are a few things to note here. First off, the bend points of the edges connected to the northern ports are chosen in a way that minimizes the vertex-local number of edge crossings. This is somewhat similar to KLoDD, with the important difference that KLoDD would not allow the long edge e_l to cross the northern edges. Second, the horizontal segments of some of the edges have the same vertical coordinate. This is by no means necessary, but saves vertical space and reduces visual clutter. And finally, the distances between horizontal edge segments is not uniform. In fact, contrary to KLoDD, edges connected to northern or southern ports can extend far above or below their regular vertex, giving the vertex placement phase a much better chance to minimize edge bends.



(a) An edge routing as our algorithm could have produced it.



(b) With dummy vertices and edges reconnected appropriately. Dashed lines hint at how the edges were connected before.

Figure 3.8. A vertex with northern ports and edges routed as the approach proposed here would do it. Note that edges may extend far above (or below) the vertex, giving the vertex placement phase a good chance of reducing edge bends. The long edge e_l is allowed to cross the northern edges.

It might not seem to make much sense at first to allow long edges to cross northern or southern edges. After all, what this does is to introduce additional crossings that could be avoided. However, there are two benefits. First, by allowing long edges to be routed anywhere, we give the crossing reduction phase a maximal amount of freedom to decide where best to place edge dummy vertices. By allowing for a few crossings with northern our southern edges, the crossing reduction phase may be able to avoid a whole bunch of additional crossings with other edges somewhere else. And second, we keep edge lengths short. Figure 3.9 shows a case where a vertex connected via a northern edge is placed high above another vertex. The long edge would usually have to be routed even higher, greatly increasing its length and decreasing readability; our approach accepts a crossing, but keeps the edge length short and readability up.

Creating the bend dummy vertices for a regular vertex $v \in L_k$ for some layer L_k starts by assembling a sequence of northern ports P_N and southern ports P_S, sorted by their x coordinate. The CreateDummyVertices algorithm—reproduced in pseudo code as Algorithm 3.2—creates the required sequences of bend dummy vertices V_N and V_S. The algorithm is fairly straightforward, the exception being the condition of the loop starting in line 7. This part of the algorithm is responsible for creating the bend dummy vertices shared by two ports. As Figure 3.8a suggests, this is only possible for two ports p_i and p_j if two conditions hold: first, p_i only has incoming edges and p_o only has outgoing edges; and second, p_i is left of p_o. Violating one of the two conditions would result in overlapping edges, introducing ambiguity.

The order of the vertices in the sequences V_N and V_S deserves some consideration. For two dummy vertices $V_N(i)$ and $V_N(i+1)$, $V_N(i)$ has to be positioned closer to v than $V_N(i+1)$. The same holds for V_S. To ensure that this order is preserved, the vertex successor constraint mechanism introduced in Section 3.3 is used. First

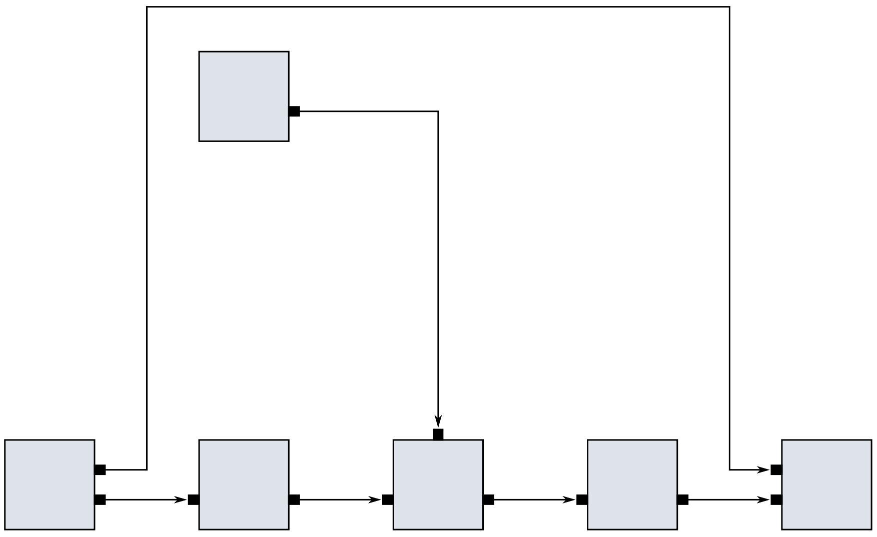


Figure 3.9. An extreme case of a long edge routed around a northern edge and becoming too long in the process. In such cases, accepting the crossing but keeping the edge length short improves readability more than avoiding the crossing at all cost.

off, the necessary constraints for dummy vertices in V_N are added to satisfy the following condition:

$$\forall 1 \leq i < |V_N| : (V_N(i), V_N(i-1)) \in C. \tag{3.1}$$

Note that this effectively reverses the order of vertices as specified by V_N, which is necessary to keep $V_N(i)$ closer to v than $V_N(i+1)$. The necessary constraints for dummy vertices in V_S can simply reproduce the order of vertices as specified by V_S to satisfy the following condition:

$$\forall 0 \leq i < |V_S| - 1 : (V_S(i), V_S(i+1)) \in C. \tag{3.2}$$

Finally, the first vertex in V_N must precede v, and v must precede the first vertex in V_S. The necessary constraints are added to satisfy the following condition:

$$V_N \neq \emptyset \Rightarrow (V_N(0), v) \in C \wedge V_S \neq \emptyset \Rightarrow (v, V_S(0)) \in C. \tag{3.3}$$

Once bend dummy vertices are created for all regular vertices and their order is ensured to be preserved, there is one more problem to be taken care of. Let v_N be a bend dummy vertex created for a northern port of some vertex v_i, and let v_S be a bend dummy vertex created for a southern port of some vertex $v_j \neq v_i$, all part of the same layer L_k. So far, nothing prevents the crossing reduction phase from placing v_S below v_N in L_k. With luck, this only introduces unnecessary crossings; Figure 3.10 however shows that it can lead to the vertical segments of edges overlapping each other, introducing ambiguity. To avoid this, bend dummy vertices created for v_i must not be placed between bend dummy vertices created for v_j, which is ensured

```
function CreateDummyVertices($P$)
    $V_D \leftarrow \emptyset$                                    ▷ Sequence of created dummies to be returned.
    $P_{In} \leftarrow \{p \in P \mid e_I(p) \neq \emptyset \land e_O(p) = \emptyset\}$          ▷ Filtered subsequences of $P$.
    $P_{Out} \leftarrow \{p \in P \mid e_I(p) = \emptyset \land e_O(p) \neq \emptyset\}$
    $P_{InOut} \leftarrow \{p \in P \mid e_I(p) \neq \emptyset \land e_O(p) \neq \emptyset\}$

    $idx_{In} \leftarrow 0,\ idx_{Out} \leftarrow |P_{Out}| - 1$          ▷ Create dummy vertices shared by two ports.
    while $idx_{In} < |P_{In}| \land idx_{Out} \geq 0 \land \mathrm{index}_P(P_{In}(idx_{In})) < \mathrm{index}_P(P_{Out}(idx_{Out}))$ do
        Add new bend dummy vertex for $P_{In}(idx_{In})$ and $P_{Out}(idx_{Out})$ to $V_D$.
        $idx_{In} \leftarrow idx_{In} + 1,\ idx_{Out} \leftarrow idx_{Out} - 1$
    end while

    while $idx_{In} < |P_{In}|$ do          ▷ Create dummy vertices for ports with incoming edges.
        Add new bend dummy vertex for $P_{In}(idx_{In})$ to $V_D$.
        $idx_{In} \leftarrow idx_{In} + 1$
    end while

    while $idx_{Out} \geq 0$ do          ▷ Create dummy vertices for ports with outgoing edges.
        Add new bend dummy vertex for $P_{Out}(idx_{Out})$ to $V_D$.
        $idx_{Out} \leftarrow idx_{Out} - 1$
    end while

    $idx_{InOut} \leftarrow 0$
    while $idx_{InOut} < |P_{InOut}|$ do          ▷ Create dummy vertices for remaining ports.
        Add new bend dummy vertex for $P_{InOut}(idx_{InOut})$ to $V_D$.
        $idx_{InOut} \leftarrow idx_{InOut} + 1$
    end while

    return $V_D$
end function
```

Algorithm 3.2. The CreateDummyVertices algorithm returns a sequence of dummy vertices for a specified sequence of ports P, which is expected to contain only northern or only southern ports, sorted by their x position.

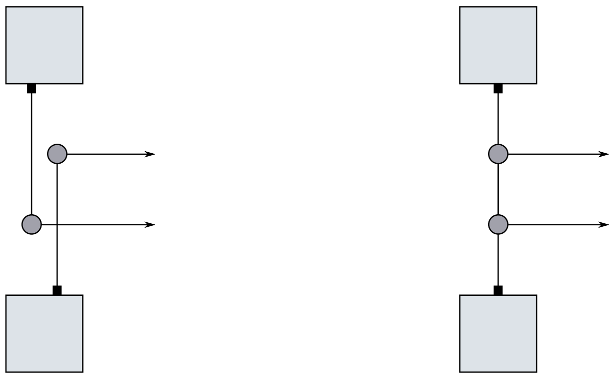
by using the layout unit mechanism introduced in Section 3.3. For a regular vertex v and its set of bend dummy vertices V_B, we set the following:

$$\mathrm{lunit}(v) \leftarrow v \land \forall d \in V_B : \mathrm{lunit}(d) \leftarrow v. \tag{3.4}$$

Impact on Cross Counting

Taking another look at Figure 3.8 we can see that the edge e_l causes two crossings in the final diagram, but does not cause any crossings while the bend dummy vertices are present. If these implicit crossings are not taken into account during crossing reduction, the algorithm is prone to compute inferior layouts.

Before introducing an algorithm to count these kinds of crossings, it is a good idea to think about how exactly they can be caused. Obviously, it is necessary to have an edge running between the bend dummy vertices of some regular vertex. Since crossing reduction requires a proper layering, it follows that this can only happen



(a) Introduction of unnecessary crossings.

(b) Introduction of ambiguity due to overlapping edge segments.

Figure 3.10. If a regular vertex and its bend dummy vertices are mixed with a bend dummy vertex of another regular vertex, additional crossings and even overlapping edges ensue.

if a vertex v is placed between the bend dummy vertices. This vertex cannot be another bend dummy vertex—we prevented that by using layout units. It cannot be another regular vertex either, for the same reason. The only thing left for it to be, then, is an edge dummy vertex, which makes developing a proper cross counting algorithm a little easier.

I propose an algorithm that requires two sweeps over a layer's vertices. During the first sweep, each bend dummy vertex is assigned a crossing hint value depending on the number of ports it was created from, and for each regular vertex the number of bend dummy vertices created for southern ports is remembered. This part of the algorithm is reproduced in pseudo code as Algorithm 3.3. The second sweep again iterates over the layer's vertices, remembering the last regular or bend dummy vertex v. If an edge dummy vertex d is encountered, the number of crossings is increased by $\text{crossHint}(v)$ if v was created from a northern port, or by $\text{sdMaxHint}(\text{lunit}(v)) - \text{crossHint}(v)$ otherwise. Figure 3.11 shows an example of the algorithm at work.

Removing Bend Dummy Vertices

Edge bend dummy vertices are removed by joining their incident edges and the associated lists of bend points. Removing bend dummy vertices takes a little more work, since creating them involves reconnecting edges and adding an additional bend point.

Let d be a bend dummy vertex created for regular vertex v, and let e be an edge incident to d, originally connected to port p with $\nu(p) = v$. To remove d, e has to be reconnected to p and a new bend point has to be added to e. Using the position of d as the new bend point does not make sense: its x position will probably differ from the x position of p, which will result in a vertical segment of e being routed at

```
function CROSSCOUNTINGSWEEP1($L_k$, crossHint : $V \to \mathbb{N}$, sdMaxHint : $V \to \mathbb{N}$)
    $\forall v \in V$ : crossHint($v$) $\leftarrow 0$
    $\forall v \in V$ : sdMaxHint($v$) $\leftarrow 0$
    $currHint \leftarrow 0$
    $north \leftarrow$ true
    $unit \leftarrow \perp$

    for all $v \in \{v \in L_k \mid v$ is no edge dummy vertex$\}$ do
        if lunit($v$) $\neq unit$ then                          ▷ New layout unit.
            $unit \leftarrow$ lunit($v$)
            $north \leftarrow$ true
        end if

        if $unit = v$ then             ▷ Regular vertex; switch from north to south.
            $north \leftarrow$ false
            $currHint \leftarrow 0$
        else if $north =$ true then                       ▷ Northern dummy vertex.
            $currHint \leftarrow currHint +$ PORTCOUNT($v$)
        else                                               ▷ Southern dummy vertex.
            $currHint \leftarrow currHint +$ PORTCOUNT($v$)
            sdMaxHint($u$) $\leftarrow currHint$
        end if
        crossHint($v$) $\leftarrow currHint$
    end for
end function
```

Algorithm 3.3. The first sweep of the bend dummy cross counting algorithm is responsible for assigning crossing hints to bend dummy vertices, and for counting the number of bend dummy vertices created for the southern ports of regular vertices. These are used by the second sweep to actually count the crossings caused by edge dummy vertices placed among bend dummy vertices.

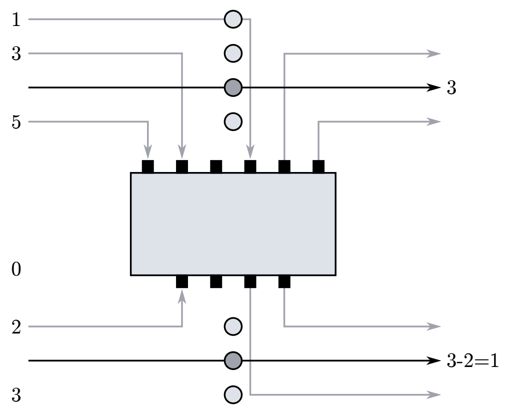


Figure 3.11. A case in which two edge dummy vertices created to split long edges are placed between bend dummy vertices. To count the number of additional crossings produced, we propose an algorithm that uses two sweeps over each layer's vertices. The numbers on the left are the cross hints computed during the algorithm's first sweep. The number on the right give the crossings caused by the edge dummy vertices as calculated during the second sweep.

an angle. Instead, the bend point must be inserted at the y coordinate of d, and at the x coordinate of p.

Advantages and Disadvantages

The approach introduced here has its clear advantages over the KLoDD approach when it comes to the number of bend points. Furthermore, it gives the crossing reduction phase the liberty to have long edges cross edges incident to northern or southern ports, thereby potentially reducing the vertical size of the diagram and making it more compact.

The biggest disadvantage is that the approach requires crossing reduction algorithms to support vertex successor constraints and layout units. However, these can be added quite easily on top of existing algorithms. Another disadvantage concerns the creation of dummy vertices. The algorithm currently favors vertex-local crossing reduction. An alternative would be to let the crossing reduction phase figure out the vertex order. Investigating this is left for future work.

▶ 3.5 Self-Loops

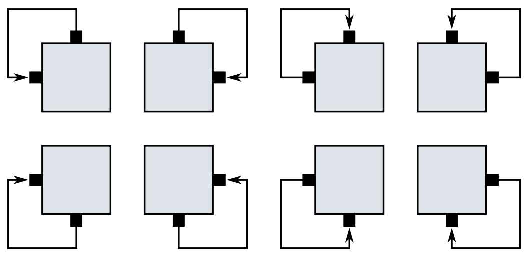
As mentioned in Definition 2.2, self-loops connect a vertex v to itself, but need to be connected to different ports. The problems posed by self-loops are a combination of the problems that occur with northern and southern ports as well as with inverted ports: they can connect ports on either side of the vertex, giving the five cases shown in Figure 3.12. (In fact, some of these cases could be combined or divided further, but these five turn out to be just enough to structure our approach.) For the remainder of this section, the North and South sides are referred to as *horizontal sides*, and the West and East sides are referred to as *vertical sides*.

As with northern, southern, and inverted ports, KLoDD handled self-loops in its vertex-local edge routing phase. While this lead to unnecessary edge bends in the other cases, it does work well with self-loops. In spite of this, KLay Layered uses a different approach, for the simple reason that only minor modifications were necessary to adapt the methods developed in the preceding sections to self-loops. The actual approaches differ for each of the five identified cases and are shown in Figure 3.13. In the following, let $v \in V$ be a vertex with a self-loop $e \in E$.

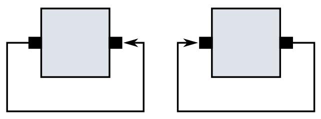
If e connects two ports on the same vertical side as shown in Figure 3.12c, nothing needs to be done since KLay Layered's orthogonal edge routing algorithm already supports this case.

If e connects two opposing vertical sides as shown in Figure 3.12b, this can be handled by breaking e up with an edge dummy vertex inserted into the layer of v.

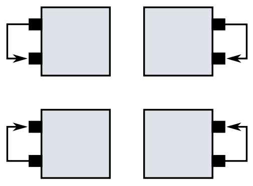
If e connects two opposing horizontal sides as shown in Figure 3.12e, it can be handled similar to other edges incident to northern and southern ports, with two exceptions: the bend dummy vertices created for self-loops must not be shared by multiple ports and they must be placed closer to their regular vertex than the regular bend dummy vertices.



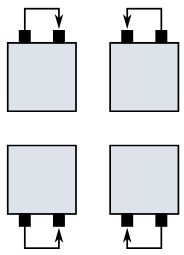
(a) Vertical and horizontal sides.



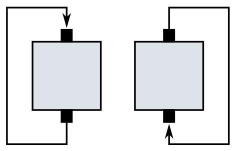
(b) Opposing vertical sides.



(c) Same vertical sides.

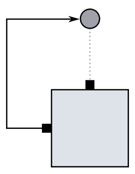


(d) Same horizontal sides.

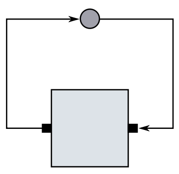


(e) Opposing horizontal sides.

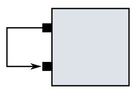
Figure 3.12. The five different cases in which self-loops can be grouped.



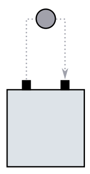
(a) Vertical and horizontal sides.



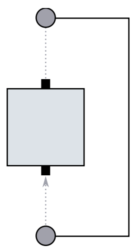
(b) Opposing vertical sides.



(c) Same vertical sides.



(d) Same horizontal sides.



(e) Opposing horizontal sides.

Figure 3.13. How the five different cases of self-loops are handled.

If e connects two ports on the same horizontal side as shown in Figure 3.12d, it can be handled similar to the previous case, with one difference: only one bend dummy vertex is created, which has no incident edges. It must be remembered which edge the dummy vertex replaces.

The remaining case has e connecting a port on a horizontal side to a port on a vertical side. The first step in handling this case is to make sure that e points to the right by reversing it as necessary. It is then enough to handle it like any other edge connected to a northern or southern port, with the exceptions described for the third case.

▶ 3.6 Hierarchical Ports

Data flow diagrams may be nested, or *hierarchical*: vertices may contain another data flow diagram instead of being atomic. As mentioned in Section 1.2, usually editing tools display only one level of hierarchy at once, not allowing the user to look inside hierarchical vertices. One design goal for KLay Layered, however, was to support the layout of hierarchical diagrams with arbitrary levels of hierarchy visible at once. To support layout algorithms in that respect, KIML supports two modes of operation with hierarchical graphs: the first hands over the complete graph to the layout algorithm, with all levels of hierarchy at once, requiring layout algorithms to support multiple levels of hierarchy; the second executes the layout algorithm for each level of hierarchy separately, going from the innermost subgraph outwards. The latter mode is the one supported by KLay Layered.

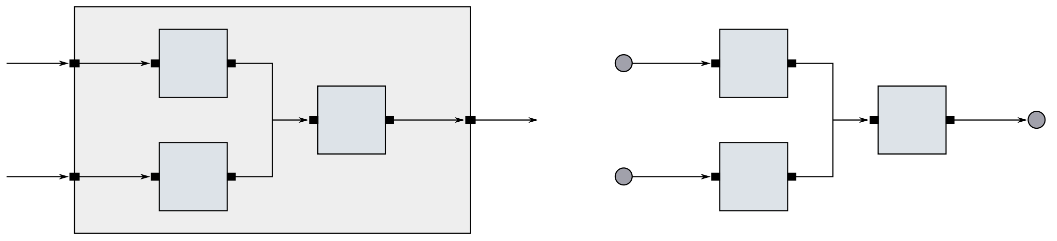
While a layout algorithm using the second mode does not need to support hierarchy as extensively as it does with the first mode, there is still one issue that usually needs special consideration to be well-supported. Let v be a hierarchical vertex with a nested data flow diagram, a western input port, and an eastern output port, similar to the vertex labeled `Embedded CT Model` in Figure 1.4. For the nested diagram to have any impact on the computation it has to receive or produce data and be connected to the outer hierarchy level in some way. With the natural way of data coming in to v via its western port and going out via its eastern port, its inner vertices can simply be connected to these to establish communication between them and the neighbors of v. Supporting these connections between ports of v and vertices of the diagram contained in v is what needs to be explicitly supported by the layout algorithm.

The easiest way to support this is to simply ignore such edges. KIML then takes care of drawing them as straight lines. However, with orthogonal edge routing and port constraints in place, that is not the way KLay Layered is supposed to handle this. In the rest of this section, we look at ways of how it does.

The Simple Case of Hierarchical Ports

Ports can be placed on any of the four sides of a hierarchical vertex, but to begin with we constrict the possibilities: ports with outgoing hierarchical edges are constrained to the western side, and ports with incoming hierarchical edges are constrained to the eastern side, thereby keeping the overall layout direction from left to right. Once a solution for this simple case is developed, it is extended for the general case in the next subsection.

From our algorithm's point of view, the task is to compute a layout for a hierarchical graph as defined in Definition 2.7: some ports are not assigned to vertices, but to the graph itself. In the spirit of reducing complex problems to simpler ones, our approach starts by introducing a *hierarchy dummy vertex* for each port to be able to treat hierarchical ports as vertices. Which layer the vertex is assigned to, how its coordinates are computed, and how its incident edges are treated differs with the port constraints set on the hierarchical vertex. After the algorithm's execution,



(a) A hierarchical vertex as KLay Layered might lay it out.

(b) The vertex as seen by KLay Layered.

Figure 3.14. In the simple case of hierarchical ports being restricted to either the western or the eastern side, hierarchy dummy vertices are created for the hierarchical ports and placed in separate layers.

the dummy vertices are removed again, setting the coordinates of the ports they represent accordingly.

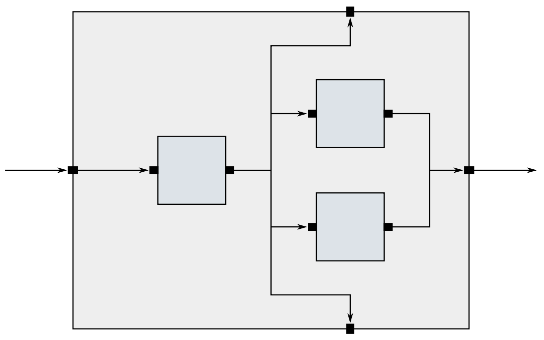
The case of port constraints set to Free is the simplest to handle, and is shown in Figure 3.14. Two new layers are created, one at the front and one at the back, and a hierarchy dummy vertex is created for each hierarchical port p. If $\text{flow}(p) > 0$ (that is, if it has more incoming edges than outgoing edges), the dummy vertex is placed in the new last layer; otherwise, it is placed in the new first layer. It is possible for vertices placed in the first layer to have incoming edges, which have to be reversed in order to keep the graph properly layered (of course, the same holds for vertices placed in the last layer that have outgoing edges). Since free port constraints do not constrain the positioning of ports, the crossing reduction and vertex placement phases are free to place the dummy vertices wherever they see fit.

The case of port constraints set to FixedSides is similar to the free case, with the exceptions that dummy vertices belonging to western ports are always placed in the new first layer, and dummy vertices belonging to eastern ports are always placed in the new last layer, regardless of their net flow. Edges are again reversed, if necessary.

The case of port constraints set to FixedOrder is similar to the case of fixed sides. With the order of ports—and thus, the order of dummy vertices—already fixed, however, the crossing reduction phase is not free to determine the vertex order any more. Vertex successor constraints are used to keep the order of dummy vertices unchanged.

The case of port constraints set to FixedRatio is again similar to the previous case. In addition to having to fix the order of dummy vertices, which affects crossing reduction, here the y coordinates of the dummy vertices are constrained as well, affecting vertex placement. The coordinates depend on the vertical size of the graph, which may change even after vertex placement. They thus have to be checked and possibly corrected before the edge routing phase, which may lead to additional edge bends.

The case of port constraints set to FixedPos is actually a little easier than the


(a) A hierarchical vertex with northern and southern hierarchical ports as KLay Layered might lay it out.

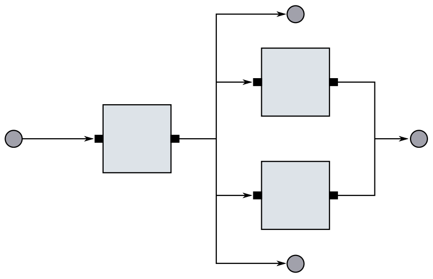

(b) The vertex as seen by KLay Layered.

Figure 3.15. The extended cases of hierarchical ports does not impose restrictions on port sides anymore: hierarchical ports can now be placed on the northern or southern side as well. In the case of FixedSides, a single hierarchy dummy vertex is created for each hierarchical port.

previous case. The y coordinates of the dummy vertices do not depend on the graph's vertical size, but can be fixed in the first place. The vertex placement algorithm either has to be adapted to support fixed positions, or the coordinates have to be checked and corrected afterwards.

Extending the Simple Case

Having first developed a solution for the simple case, we can now go on and remove the constraints, adapting the solution to the general case. Removing the constraints means that ports can now be placed on the northern or southern sides. Again, we work our way through the different kinds of port constraints.

The case of port constraints set to Free allows us to place hierarchical ports anywhere we want. Of course, we settle for the easy and straightforward solution already used in the simple case.

The problem becomes more difficult once port constraints are set to at least FixedSides. With ports placed on the western or eastern sides the approach developed for the simple case still works well. However, as shown in Figure 3.15a, they may now be placed on the northern or southern sides as well, making the problem more complex to solve. Figure 3.15b shows how KLay Layered handles this case: for these ports, dummy vertices are created with just one port on the western side that all edges incident to the hierarchical port are reconnected to. Instead of being placed in the first or last layer, the vertex takes part in layer assignment. After the edge routing phase, the dummy vertex is removed, the hierarchical port is assigned the vertex's x coordinate, and incident edges are reconnected to the hierarchical port, with a bend point inserted at the dummy vertex's former location.

The approach, as described so far, presents two problems. The first concerns the problem of regular vertices placed between a hierarchical dummy vertex and the

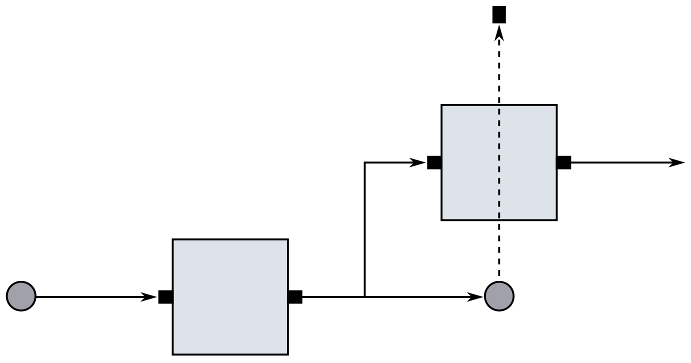


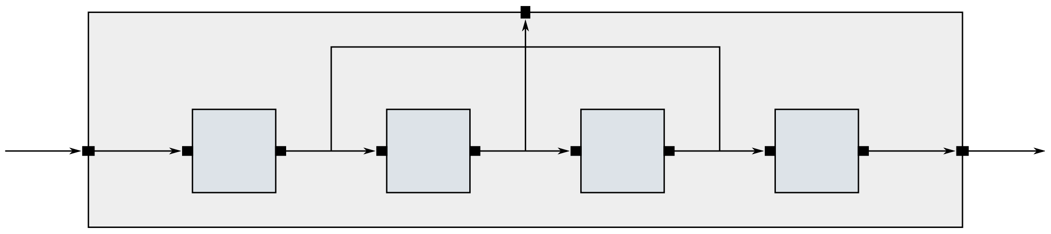
Figure 3.16. Without further consideration, regular vertices can be placed between a hierarchy dummy vertex and its hierarchical port, resulting in hierarchical edges crossing regular vertices. Our algorithm avoids this by reserving special areas at the top and bottom of each layer for hierarchy dummy vertices.

side its hierarchical port is assigned to. As Figure 3.16 shows, just reconnecting incident edges and adding a bend point would result in edges crossing the regular vertex, which is not desirable. We solve this problem by dividing layers into three regions, with the top and bottom regions reserved for hierarchical dummy vertices representing northern and southern hierarchical ports.

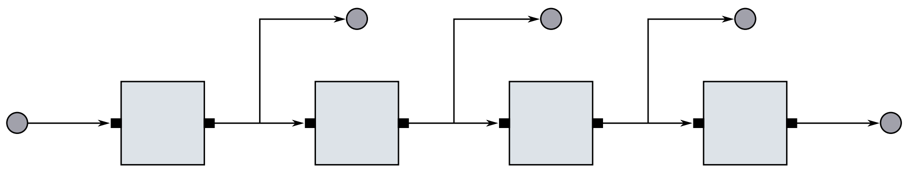
The second problem arises when a layer contains several dummy vertices representing hierarchical ports assigned to the same side. So far, the approach would make the edges and hierarchical ports overlap. We avoid this by adjusting the x coordinate of the dummy vertices just prior to the edge routing phase such that the edges are routed next to one another.

Handling the case of port constraints set to FixedOrder differs in two key points from the previous approach: the creation of dummy vertices, and the routing of hierarchical edges. Instead of creating just one dummy vertex for each hierarchical port, one dummy vertex is created for each layer containing vertices connected to the hierarchical port, and placed in the following layer as shown in Figure 3.17. Now, every hierarchical port may have several dummy vertices created for it, so its position cannot be derived directly anymore. Instead, a barycenter heuristic is used to set its position, with Forster's algorithm [10] used to ensure the proper port order. Simply reconnecting the edges and adding a bend point does not work anymore either: the hierarchical port will usually have an x coordinate that differs from the coordinate of its dummy vertices. Instead, a full orthogonal edge routing algorithm is executed to properly route the edges between hierarchical ports and their dummy vertices.

The cases of port constraints set to FixedRatio or FixedPos can be handled similarly to the previous case, with the exception of how the positions of hierarchical ports are set. With these constraints active, the port position may not be arbitrarily set anymore, but is fixed or at least directly derived from the graph's size. In these cases, the position is not calculated using a barycenter heuristic, but simply set to the required or calculated value.



(a) A hierarchical vertex with a northern hierarchical port and port constraints at least at FIXED-ORDER as KLay Layered might lay it out.



(b) The vertex as seen by KLay Layered.

Figure 3.17. In the case of port constraints set to at least FIXEDORDER, hierarchical ports do not have just one dummy vertex created for them anymore. Instead, each layer following one that has vertices connected to a hierarchical port contains a dummy vertex for that port, with edges between the dummy vertices and the hierarchical ports later routed using a full-blown orthogonal edge routing algorithm.

Further Thoughts About Hierarchy

In data flow diagrams, port constraints for hierarchical ports are usually set to FREE, giving the algorithm responsibility for crossing reduction. Our approach of always looking at just one level of hierarchy is limited to reduce crossings locally, computing the port order in a way that results in less crossings in the current level of hierarchy, but without taking the environment into account. Figure 3.18 illustrates that this can easily lead to non-optimal results. While solving this is out of the scope of this thesis, there is research based on work by Sugiyama and Misue that explores laying out compound graphs [28]. Current research at our group explores a solution that flattens the hierarchy to be able to take the environment into account.

▶ 3.7 Minimizing Edge Bends

The vertex placement algorithm proposed by Sander [24] and used in KLay Layered tries to keep the number of edge bends low. One way it achieves this concerns the treatment of long edges, or rather, the treatment of edge dummy vertices inserted to turn a standard layering into a proper layering. As mentioned in Section 2.2, the algorithm is based on the concept of linear segments: sets of vertices that are to be placed along a straight line. All edge dummy vertices inserted to split the same long edge e end up in the same linear segment, resulting in straight long edges.

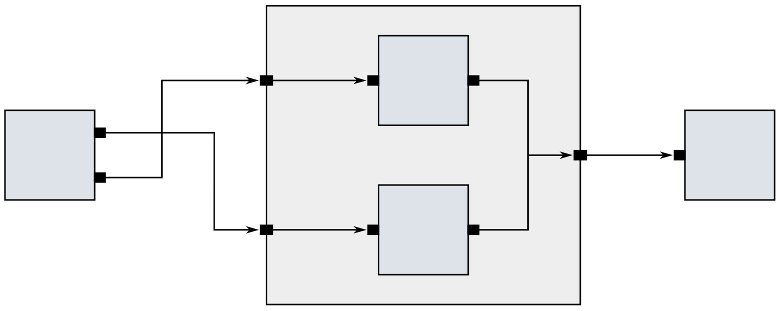


Figure 3.18. An algorithm that always lays out each level of hierarchy separately cannot take the environment into account during crossing reduction. Here, the hierarchical vertex was laid out first, with port constraints initially set to FREE. Afterwards, the outer level of hierarchy was laid out, with the hierarchical ports fixed to their previously computed positions. The crossing could have been avoided if the algorithm had taken the outer level of hierarchy into consideration in the first place.

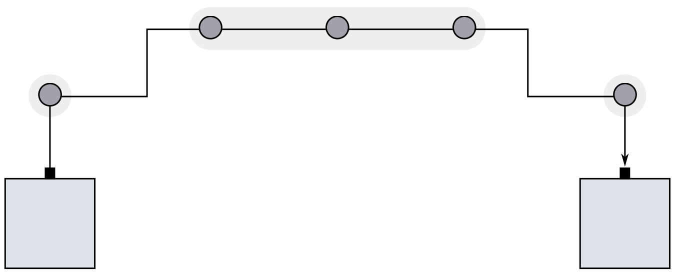
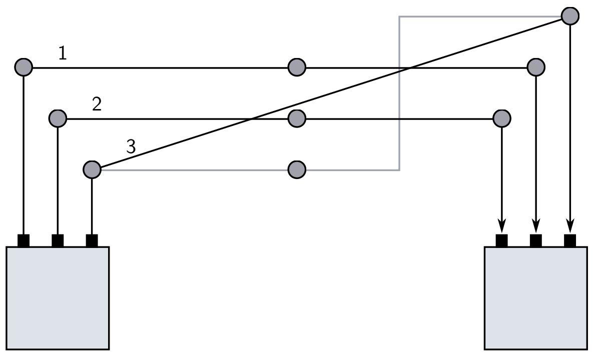


Figure 3.19. Our approach to handling northern and southern ports can introduce unnecessary bend points when the linear segments do not extend to bend dummy vertices. Even in this simple case, making them part of the same segment would get rid of four bend points.

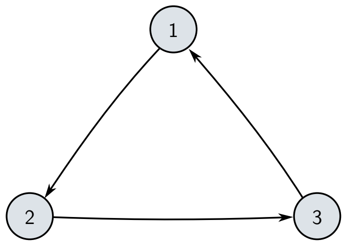
One of the disadvantages of how KLoDD handled inverted, northern and southern ports was the introduction of unnecessary bend points. While this is not a problem anymore with how KLay Layered handles inverted ports, Figure 3.19 illustrates how it is one with our way of handling northern and southern ports. Without further consideration, linear segments do not extend to inserted bend dummy vertices, and thus may cause unnecessary bend points—but changing this is not without problems.

Consider the diagram shown in Figure 3.20a: It is perfectly valid; however, the northern ports of the two vertices are connected in a way that not all three edges can be drawn as straight, orthogonal lines.

To calculate the coordinates of vertices, Sander's algorithm iterates through all vertices of all layers in graph G, building a *segment ordering graph* $O = (V_O, E_O)$ whose vertices represent the linear segments. If, in some layer in G, it finds a vertex w directly after a vertex v, the algorithm inserts an edge in the segment ordering graph from the linear segment of v to the linear segment of w. The edge can be interpreted as a dependency: since v appears above w in G, the respective linear segments must retain that order. Sander shows that when linear segments are created from dummy vertex sequences of long edges, the resulting segment ordering



(a) A diagram with ports connected so that not all three edges can be drawn with only two bend points.



(b) The linear segment ordering graph for the diagram.

Figure 3.20. If linear segments are simply extended to include bend dummy vertices, there are diagrams for which the segment ordering graph is not acyclic, and thus does not imply a single correct segment ordering anymore.

graph is free of cycles; this property is necessary to establish the vertical order of linear segments. With bend dummy vertices made part of linear segments, however, the ordering graph may well contain cycles, as Figure 3.20b shows.

To ensure that the segment ordering graph is acyclic, segments must be split, either during or after building the ordering graph. Splitting them afterwards is difficult, since the question of which segments to split cannot be answered easily. Even in the simple case of Figure 3.20, we could split either segment 3, or both, segments 1 and 2—all the time being unsure of where the segments should be split.

My approach is based on avoiding the introduction of cycles in the first place. The reason for a cycle in the dependency graph is that a pair of linear segments appear in a certain order in one layer, and in the reverse order in another layer. In other words, if when building the ordering graph we detect that a linear segment s_1 appears after some other segment s_2 when it previously appeared before it, we can split s_1, beginning a new one as soon as we detect this case. (The light edge in Figure 3.20a hints at where segment 3 would be split using this approach.) The algorithm to create the edges in the segment ordering graph is given in pseudo code as Algorithm 3.4.

▶ 3.8 Simplifying Hyperedges

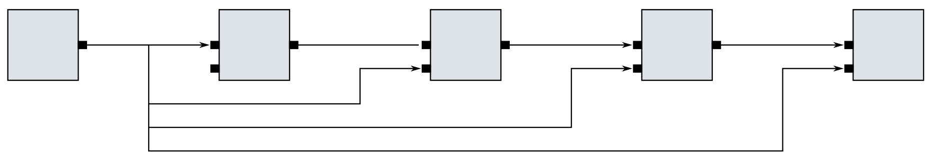
In graph theory, hyperedges are undirected edges that connect more than two ports, and are commonly represented by a set of ports rather than an ordered pair. For our purposes, we change this definition a little to mean sets of edges that share a common port. Using orthogonal edge routing, hyperedges can be routed such that they share some horizontal and vertical segments. Figure 3.21a shows a diagram with hyperedges that are routed just like any other regular edge, while Figure 3.21b

```
procedure CreateOrderingGraphEdges($G$, $O = (V_O, E_O)$)
    for $k \leftarrow 0, |L| - 1$ do                                        ▷ Iterate through the layers
        $lastSegment \leftarrow \perp$
        for $i \leftarrow 0, |L_k| - 1$ do                              ▷ Iterate through the layer's vertices
            $currSegment \leftarrow \text{segment}(L_k(i))$
            $cycleSegment \leftarrow$ FindCycleSegment($k, i, currSegment$)
            if $cycleSegment \neq \perp$ then                       ▷ Check if we would introduce a cycle
                $oldSegment \leftarrow currSegment$
                $currSegment \leftarrow$ AddNewSegment($V_O$)
                $\text{vertices}(currSegment) \leftarrow \{v \in \text{vertices}(oldSegment) \mid \exists j \geq i : v \in L_j\}$
                $\text{vertices}(oldSegment) \leftarrow \text{vertices}(oldSegment) \setminus \text{vertices}(currSegment)$
            end if

            if $lastSegment \neq \perp$ then                ▷ Possibly add an edge in the ordering graph
                $E_O \leftarrow E_O \cup \{(lastSegment, currSegment)\}$
            end if

            $\text{lastLayer}(currSegment) \leftarrow k$                                        ▷ Bookkeeping
            $\text{lastLayerIndex}(currSegment) \leftarrow i$
            $lastSegment \leftarrow currSegment$
        end for
    end for
end procedure

function FindCycleSegment($0 \leq k < |L|$, $0 \leq i < |L_k|$, $s \in V_O$)
    for $j \leftarrow i + 1, |L_k| - 1$ do                  ▷ Iterate through the remaining vertices in $L_k$
        $cycleSegment \leftarrow \text{segment}(L_k(j))$
        if $\text{lastLayer}(s) = \text{lastLayer}(cycleSegment)$ then
            if $\text{lastLayerIndex}(s) > \text{lastLayerIndex}(cycleSegment)$ then
                return $cycleSegment$                                 ▷ We would introduce a cycle
            end if
        end if
    end for
    return $\perp$                                                              ▷ No cycle found
end function
```

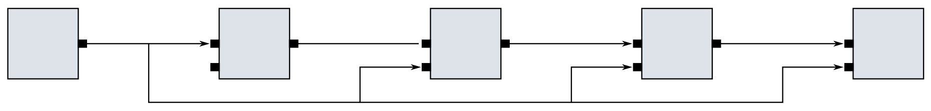
Algorithm 3.4. This algorithm adds the edges to the segment ordering graph, avoiding those that would introduce a cycle. The function segment : $V \to V_O$ returns the linear segment a given vertex is part of; vertices : $V_O \to \mathcal{P}(V)$ returns the vertices that are part of a given linear segment. The functions lastLayer : $V_O \to \mathbb{N}$ and lastLayerIndex : $V_O \to \mathbb{N}$ are used to keep track of the last layer a linear segment was encountered in, and its position in the layer. The function AddNewSegment(...) adds a new linear segment to V_O and returns it.

shows the same diagram with compacted hyperedges. Note how the second diagram is superior in terms of both readability and conciseness.

As Figure 3.21 suggests, the vertical segments of hyperedges are already compacted by the orthogonal edge routing algorithm used in KLay Layered, leaving us with having to compact the horizontal segments. Those only play a role if the hyperedges are long edges; the problem of compacting them is thus reduced to the problem of merging their edge dummy vertices. Our approach for this works by remembering the long edge's source and target port for each edge dummy vertex



(a) Hyperedges drawn like regular edges. The three bottom edges are long edges, each having its own edge dummy vertices.



(b) Hyperedges compacted to share a horizontal segment by merging their edge dummy vertices.

Figure 3.21. Drawing hyperedges separately adds to the visual clutter of a diagram, and increases its size. Readability is greatly enhanced by letting hyperedges share horizontal segments.

inserted to break up the edge, and by merging those dummy vertices that share a common source or target port. The algorithm is run just after crossing reduction, and is given in pseudo code as Algorithm 3.5. When two dummy vertices are merged, the corresponding long edge source and target port information needs to be updated. Since the long edges of the dummy vertices usually do not share both ports, we set the source port of the merged dummy vertex to $\perp$ if the source ports differ, and the target port to $\perp$ if the target ports differ.

▶ 3.9 Spacing and Margins

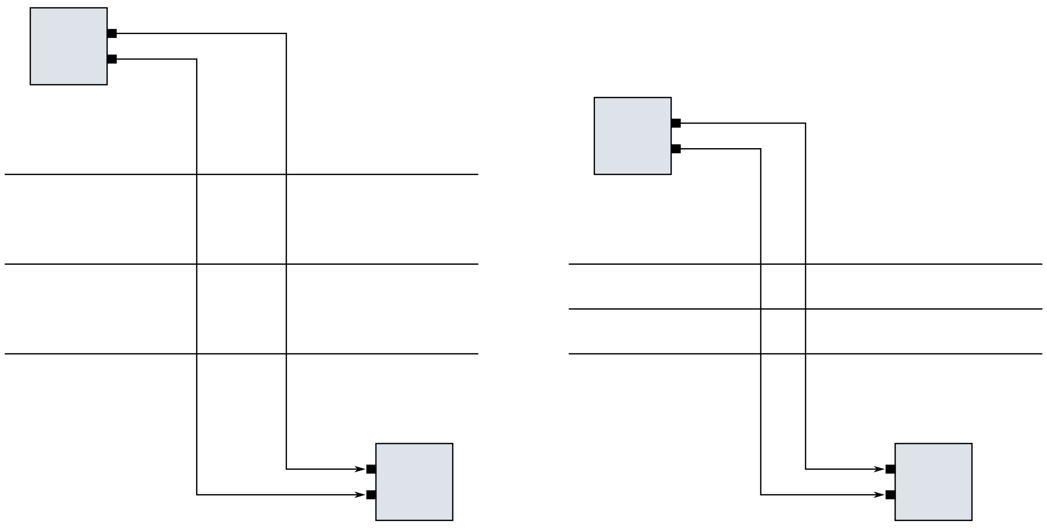
When drawing diagrams, the spacing between their different components is important. This is shown in Figure 3.22: too much spacing enlarges the diagram and makes it difficult to gain an overview, while too little spacing makes a diagram very crowded and hard to read. Our approach to spacing is to keep a certain distance d between regular vertices, and to keep edges closer together. To that end, we introduce an *edge spacing factor* d_e, with the distance between edges computed as $d \cdot d_e$.

The edge spacing factor needs to be taken into consideration during two phases. The vertex placement phase is affected when dummy vertices are present—dummy vertices are removed later, and thus must be treated as representatives of an edge. When setting the spacing between any two vertices $v, w \in V$, the spacing must be at least d if at least one of the two is a regular vertex; if both v and w are dummy vertices, the spacing must be at least $d \cdot d_e$. The edge routing phase must take care to add the proper spacings between vertices and vertical segments on the one hand, and among different vertical segments on the other hand.

In data flow diagrams, vertices are often identified by labels. This in itself would not pose a problem, if it weren't for the fact that these labels are usually not constricted to the rectangle the vertex is drawn as. Instead, vertex labels are often drawn underneath, and, to make matters worse, ports may also have labels. If this

```
procedure MergeDummyVertices($G$, leSource : $V \to P$, leTarget : $V \to P$)
    $oldVertices \subseteq V \leftarrow \emptyset$                          ▷ Set of vertices to remove later.

    for $k \leftarrow 0, |L| - 1$ do                                        ▷ Iterate through the layers.
        $lastV \leftarrow \bot$
        for $i \leftarrow 0, |L_k| - 1$ do                               ▷ Iterate through the layer's vertices.
            if leSource($lastV$) = leSource($L_k(i)$) $\vee$ leTarget($lastV$) = leTarget($L_k(i)$ then
                MergeInto($L_k(i)$, $lastV$)
                $oldVertices = oldVertices \cup \{L_k(i)\}$

                if leSource($lastV$) $\neq$ leSource($L_k(i)$) then
                    leSource($lastV$) $\leftarrow \bot$
                end if
                if leTarget($lastV$) $\neq$ leTarget($L_k(i)$) then
                    leTarget($lastV$) $\leftarrow \bot$
                end if
            end if
        end for
    end for

    RemoveVertices($oldVertices$)
end procedure
```

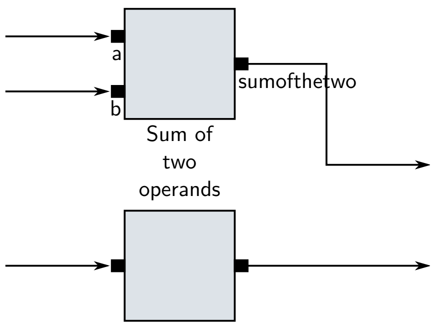
Algorithm 3.5. This algorithm iterates over a graph's edge dummy vertices and merges those belonging to the same hyperedge. The two functions leSource and leTarget assign a long edge's source and target port to its edge dummy vertices. This information is updated as the algorithm merges the dummy vertices.



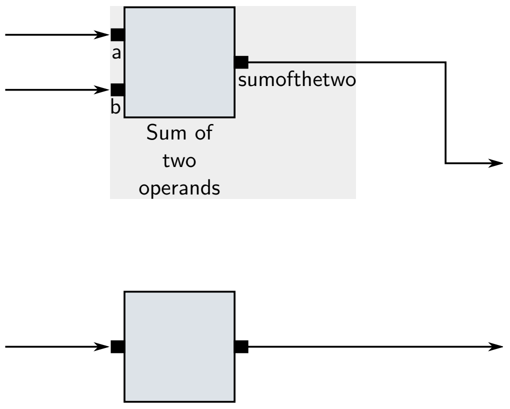
(a) With equal spacing between vertices and edges, a diagram can become very large, wasting much space.

(b) With less spacing between edges, diagrams become more compact without sacrificing readability.

Figure 3.22. The spacing between the different components of a diagram determine the diagram's size. Diagrams can be made much more compact by reducing those spacings whose reduction does not sacrifice readability.



(a) Without taking labels into consideration, vertices may appear to be too close to one another, edges may cross labels, and which component a label belongs to may become ambiguous.



(b) Using a vertex's bounding box (here shown in gray) during vertex placement and edge routing solves these problems.

Figure 3.23. Vertices and ports may have labels that must be kept clear of other vertices and edges. An ideal solution would place labels intelligently, but as this is hard we settle on an approach involving a vertex's bounding box.

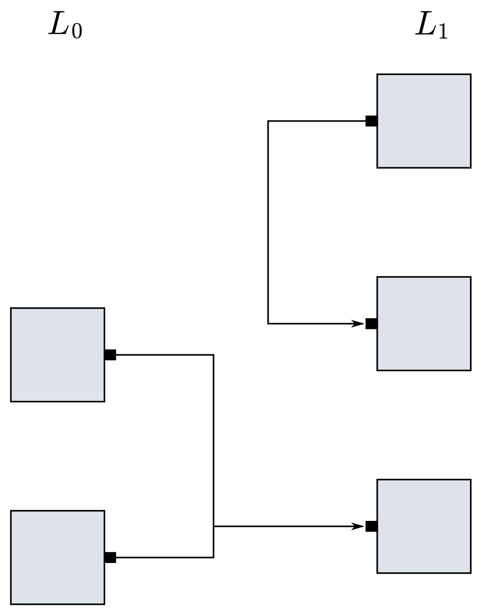
is not considered during vertex placement and edge routing, the spacing between vertices may appear too small in the final drawing, and edges may intersect labels, as shown in Figure 3.23a.

The optimal solution would be to compute proper positions for the labels, but this kind of explicit label management is hard and out of the scope of this work. So for the time being, we propose a simpler approach to at least keep the labels readable. This approach is based on two assumptions: first, the size of the labels does not change during layout; and second, the label positions relative to the vertices and ports stays the same. If these conditions are met, all we have to do is to compute a bounding box for each vertex that contains the vertex itself, its ports, and all the associated labels. Instead of using the size and position of the vertex during vertex placement and edge routing, we use the size and position of its bounding box. The result is shown in Figure 3.23b.

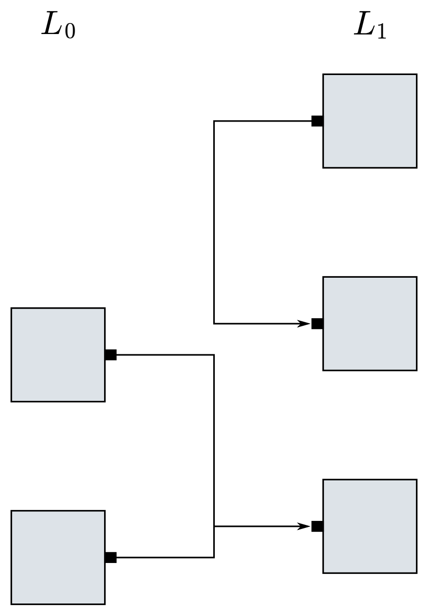
▶ 3.10 Reducing Layer Distances

The orthogonal edge routing algorithm implemented in KLay Layered is based on an algorithm developed by Sander [25]. It iterates over the graph's layers, routing the outgoing edges for each layer: first those edges originating from western ports, then those edges originating from eastern ports.

To do just that, the algorithm iterates over the western or eastern ports of a layer, assembling a list of *vertical segments*. Each such segment has a list of western and eastern ports connected to it, whose y coordinate determine the segment's vertical



(a) The vertical segments in this example could well have been assigned to the same segment slot.



(b) Here, the vertical segments are assigned to the same segment slot, saving some space between the two layers.

Figure 3.24. When each layer's outgoing edges are considered separately, this can lead to space being wasted. The solution is to consider all outgoing edges between pairs of layers.

extent. For each port p it finds that it has not visited yet, a new vertical segment is created and the port added to its list of western or eastern ports. The algorithm goes on by visiting all the ports reachable from p, adding them to the segment's list of ports and going on to their reachable ports. If a port is encountered for which a vertical segment has already been created, the two segments are merged.

Once all ports have been visited, we know the list of vertical segments and their vertical extent, but not their horizontal coordinate. The algorithm now creates horizontal segment slots and assigns the segments to these slots such that the segments do not overlap. The assignment is done in a way to minimize the number of resulting edge crossings (see Figure 2.3 for why this is important).

Doing this for the western and eastern ports of a layer separately can result in too many segment slots being created, and can thus lead to too much space being left between two layers. An example of such a situation is shown in Figure 3.24a. Here, the vertical segments created for western ports of layer L_1 could well have been placed in the same slot as the vertical segments created for eastern ports of layer L_0.

The algorithm was improved by changing the way it iterates over the layers. Instead of looking at each layer separately, the algorithm now looks at pairs of layers (L_i, L_{i+1}), with the western ports of the first layer and the eastern ports of the last layer considered separately. For each pair of layers, all the necessary vertical segments for the eastern ports of L_i and the western ports of L_{i+1} are created and assigned to segment slots, improving the example from Figure 3.24a as shown in Figure 3.24b.

▶ 3.11 Calculating Barycenters

The calculation of barycenters is based on looking at assigned rank values of the neighborhood of a vertex. In Section 3.1, where we talked about in-layer edges, we have already seen how this fails for in-layer edges; but from Section 3.4, which was about northern and southern ports, follows another problem with the usual barycenter approach: if a regular vertex $v \in V$ has northern or southern ports, its neighborhood changes. In particular, after bend dummy vertices have been created those vertices that were connected to it via northern or southern ports are then connected to the bend dummy vertices—and thus not part of the neighborhood of v anymore. This, in turn, leads to their rank values not being taken into account, resulting in a possibly undesired barycenter value for v.

To remedy this, we experimented with an approach that attaches a list of vertices N_v, called *barycenter associates*, to each vertex $v \in V$. The vertices in v are taken into account during the barycenter calculation of v, with a factor $w_n \in \mathbb{R}$ that determines their impact on the calculation. However, our experiments have shown—somewhat to our surprise—that this method does not significantly improve the number of crossings.

4

Integration into KLay Layered

Please note that we have added a consequence for failure. Any contact with the chamber floor will result in an unsatisfactory mark on your official testing record. Followed by death. Good luck!

— *GLaDOS, Portal*

With the theory in place, this chapter is all about the practical side of things. We start by explaining the new architecture of KLay Layered, and then spend the rest of the chapter looking at how the theoretical concepts are implemented, getting rather technical in some places.

▶ 4.1 A Dynamic Architecture for Layout Algorithms

The five-phase structure for layer-based layout algorithms as outlined in Section 2.2 lends itself well to laying out diagrams without ports. However, the last section has shown that handling ports well is a lot more complex, giving rise to the question if the five-phase architecture fits this application well. As it turns out, it does not.

KLoDD was structured around the five phases, with an additional phase added just before vertex placement for vertex-local edge routing. Parts of the algorithm were implemented as part of the data structure used to represent graphs instead of clearly separating data and logic. The complexity of handling ports and supporting different layout directions quickly brought this structure to its limits. For KLay Layered, a new architecture was to ensure that the algorithm would stay maintainable and would adapt well to changing requirements.

Intermediate Processors

The basic idea of our new architecture is to keep the five main phases around, but to keep them as simple as possible by factoring out everything that does not contribute

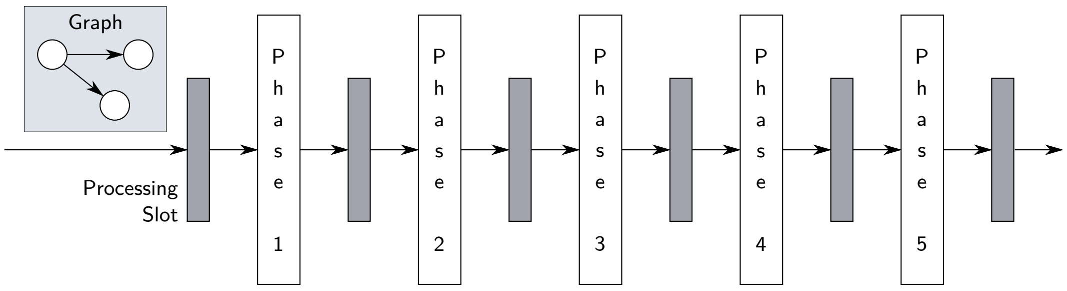


Figure 4.1. The architecture of KLay Layered distinguishes between the five phases and six processing slots, capable of housing an arbitrary number of intermediate processors.

directly to their respective goals. Furthermore, our data structure was to be kept free of the algorithm. To that end, we introduce the concept of *intermediate processors*: comparatively small modules that can be inserted before, after or between the main phases, doing the work that would otherwise make the main phases unnecessarily complex.

As a first example of an intermediate processor, let us look at the task of turning a layering into a proper layering. This would usually be done during the layer assignment phase, but there are two disadvantages to this: first, different implementations of layer assignment algorithms would have to include essentially duplicate code to turn their layering into a proper layering; and second, even if later phases do not actually require the layering to be proper, the layer assignment phase would happily go ahead and produce one anyway. This is solved by having only one intermediate processor execute right after the layer assignment phase, if at all necessary.

As Figure 4.1 illustrates, intermediate processors are placed in *processing slots* distributed around and between the five phases. Each slot can hold an arbitrary number of processors, but not more than one instance of each; different instances of the same processor can however be placed in different processing slots. When previously the graph was passed on from one phase to the next, it now has to pass through each processing slot, invoking each intermediate processor on it along the way.

To determine which intermediate processors are to be placed in the processing slots, implementations of the main phases are required to specify their *dependencies*: which processors they require in which slots. When KLay Layered is invoked, the user preferences are queried for the particular implementations to be used for the different phases, after which these implementations are queried for their dependencies, filling the processing slots accordingly. If, for instance, the crossing reduction algorithm requires the layering to be proper, it will depend on the appropriate intermediate processor to be inserted in the slot between phases two and three.

Having multiple intermediate processors in the same processing slot raises the question of which order to run them in. With the approach being as dynamic as it is, it seems probable that the order may influence the results. Indeed, it often does, potentially rendering the whole approach unusable if this problem is not solved well. The order can be deducted easily if we keep a map of dependencies between intermediate processors around: if processor A only gives correct results in the presence of processor B if B is run prior to A, A depends on B. Keeping up with these dependencies, however, quickly increases in complexity with the number of intermediate processors involved. Fortunately, an intermediate processor can usually only be sensibly placed in one or at most two processing slots, leaving us with having to manage the dependencies between only parts of the intermediate processors. If A and B will never be placed in the same processing slot anyway, there is no need to think about how they could interfere with one another. With the map of dependencies in place we can derive a proper order for the processors in each slot during their development.

Moving tasks from the five phases out to intermediate processors comes with a performance penalty. Each processor usually has to iterate over the whole graph, piling up iterations when many processors are used. While this is a concern for huge graphs, it usually is not for our applications; we have found KLay Layered to perform reasonably well and think that the benefits of the clear structure far outweigh the performance hit, on which Section 5.5 has more details.

Alternatives

The introduction of intermediate processors is by no means the only way to restructure the algorithm. In this subsection, we discuss some alternatives that come to mind.

Having divided the algorithm into smaller parts inserted dynamically into processing slots, it seems tempting to get rid of the distinction between processors and phases altogether, making the algorithm completely dynamic. For a given layout task, one would choose the necessary modules, letting KLay Layered taking care of satisfying dependencies and establishing the proper order. There are several reasons why we did not take the concept this far.

First, getting rid of the five phases actually weakens the algorithm's structure when the goal was to strengthen it. The five phases are well established, making the algorithm much easier accessible and easier to grasp than a set of loosely tied modules.

Second, with the structure of KLay Layered, users choose between different implementations of the five phases. Getting rid of them (the five phases, not the users) raises the problem of how the algorithm can be configured. One might present a selection of predefined configurations, but that is unlikely to cover all use cases. Allowing the user to build his own configurations from scratch would require him to know a lot about the algorithm when in fact he just wants his diagrams laid out. Whatever the solution, using the algorithm only gets harder for the user.

Finally, we have already seen that keeping track of the dependencies between intermediate processors gets increasingly harder with more processors. We have simplified the problem by reducing the number of processors that can appear in the same slot, but without this restriction, dependency management quickly becomes a nightmare.

An alternative going into the opposite direction is to concentrate on the five phases, supplying different implementations tailored to specific layout applications. For instance, laying out data flow diagrams with northern and southern ports would require a specific set of phase implementations supporting that kind of layout. The immediate disadvantage of this approach is its inflexibility: for each supported layout application, a complete set of phase implementations would have to be written, with only minor changes to accommodate the application's specific challenges.

▶ 4.2 An In-Depth Look at KLay Layered

Over the course of this thesis, 19 intermediate processors were developed for KLay Layered, listed in Table 4.1 together with the processing slots they can be placed in. Some of these simply extract functionality from one of the phases, while others implement the new concepts introduced in Chapter 3. This section describes each one of them in detail, complete with dependencies to other processors in the same slot, preconditions, and postconditions. The focus lies on what each processor contributes with respect to the concepts introduced in Chapter 3; the next section takes that view a little higher, concentrating on how the processors fit together to form a layout algorithm.

Edge and Layer Constraint Edge Reverser

Sometimes, vertices are required to be placed in the first or last layer, which restricts their edges to either be all outgoing or all incoming for a valid layering to be produced. Examples are hierarchical dummy vertices inserted as part of handling hierarchical ports, or vertices that represent data producers, which are sometimes required to be placed on the left side of a diagram. To this end, a *layer constraint* can be attached to vertices to make the algorithm aware of the required placement. This processor takes care of reversing the edges such that a valid layering can be produced when the layer constraint is satisfied. As a simple rule, all layer assignment implementations should include a dependency on this processor.

As its precondition, this processor expects an unlayered graph.

As its postcondition, vertices with layer constraints have their incident edges reversed accordingly.

This processor has no dependencies on other processors.

Hierarchical Port Constraint Processor

Handling hierarchical ports involves creating hierarchical dummy vertices. This processor takes care of some required preprocessing before crossing reduction, and sets necessary constraints.

Processing Slot	Intermediate Processors
Before phase 1	EdgeAndLayerConstraintEdgeReverser
Before phase 3	HierarchicalPortConstraintProcessor InvertedPortProcessor LayerConstraintProcessor LongEdgeSplitter NorthSouthPortPreprocessor PortListSorter PortSideProcessor SelfLoopProcessor
Before phase 4	HyperedgeDummyMerger InLayerConstraintProcessor PortListSorter PortPositionProcessor VertexMarginCalculator
Before phase 5	HierarchicalPortDummySizeProcessor HierarchicalPortPositionProcessor
After phase 5	HierarchicalPortOrthogonalEdgeRouter LongEdgeJoiner NorthSouthPortPostprocessor ReversedEdgeRestorer

Table 4.1. The 19 intermediate processors developed for KLay Layered, along with the processing slots they may be placed in. While most processors can only be sensibly placed in one slot, the PortOrderProcessor may appear both before phase 3 and before phase 4.

For eastern and western hierarchical ports, the order of their dummy vertices is fixed if port constraints are set to FixedOrder or higher. This processor sets the necessary vertex successor constraints to ensure that the order is kept.

For northern and southern hierarchical ports, the graph importer only creates one dummy vertex. As we have seen in Section 3.6, however, this may not be enough: with port constraints set to anything other than Free, there needs to be one dummy vertex per port and layer that contains vertices connected to the port. Since the layer assignment is not known yet when importing the graph, this processor replaces the single dummy vertex by the required number of additional dummy vertices. To avoid having to distinguish between the different port constraints, the dummy vertices are also replaced in the Free case.

This processor is tightly related to the HierarchicalPortDummySizeProcessor, the HierarchicalPortOrthogonalEdgeRouter, and the HierarchicalPortPositionProcessor. Phases either rely on all or none of them.

As its preconditions, this processor expects the graph to be layered, edge dummy vertices to not have been inserted yet, (because inserting additional hierarchy dummy vertices usually already removes some long edges) and layer constraints to be satisfied.

As its postconditions, constraints on hierarchy dummy vertices are set such that port constraints are respected.

Since layer constraints are expected to be satisfied, this processor depends on the LayerConstraintProcessor.

Inverted Port Processor

This processor iterates over the graph's vertices, looking for inverted ports. For each such port it finds, the offending edges are iterated over, creating edge dummy vertices as explained in Section 3.2. The processor simply appends the created dummy vertices to the layers, leaving the task of figuring out their order to the crossing reduction phase.

As its preconditions, this processor expects the graph to be layered, and port sides to be fixed.

As its postconditions, new edge dummy vertices may have been created, and the graph may now contain in-layer edges.

Since this processor relies on port side information, the PortSideProcessor is run prior to it.

Layer Constraint Processor

To restrict the layers a vertex can be placed in, a *layer constraint* can be attached to it that specifies the layer the vertex is to be placed in: either the first or last existing layer, or a newly created first or last layer. If layer assignment implementations do not support layer constraints, they depend on this processor to adhere to them anyway.

This processor iterates over the graph's vertices, looking for those whose layer constraint is not satisfied yet. If necessary, a new first and last layer is created. For the graph's layering to still be valid after running the processor, vertices placed in the first layers may only have outgoing edges, and vertices placed in the last layers may only have incoming edges. The EdgeAndLayerConstraintEdgeReverser can help ensure this.

As its preconditions, this processor expects the graph to be layered, and vertices to be placed in the first or last layers to have only outgoing or incoming edges.

As its postcondition, vertices may have been moved to different layers such that all layer constraints are satisfied.

The HierarchicalPortConstraintProcessor may set layer constraints and is thus run prior to this processor.

Long Edge Splitter

The layer assignment phase is not required to produce a layering that is proper. If later phases require it to be, they depend on this processor, which iterates over the graph's layers looking for long edges. If one is found, it is split by inserting an edge dummy vertex. To be able to merge hyperedges later, the dummy vertex is annotated with the original source and target ports of the long edge.

As its precondition, the processor expects the graph to be layered.

As its postcondition, the graph is properly layered.

Since the LayerConstraintProcessor modifies the layer assignment, it is run before this processor.

North South Port Preprocessor

Part of handling northern and southern ports is the creation of appropriate bend dummy vertices, which is what this processor does. It iterates over the graph's vertices, looking for those with port constraints set to at least FixedSides and with northern or southern ports. For any it finds, it creates bend dummy vertices as explained in Section 3.4, with proper vertex successor constraint and layout unit information attached. The vertices are simply added to the end of the layers, leaving the task of ordering them to the crossing reduction phase.

Self-loops involving northern or southern ports are handled as well, but it is expected that edges involving ports on two different sides either originate at a western port, a northern port, or, if the edge goes into an eastern port, at a northern or southern port. Ensuring this is part of the responsibility of the SelfLoopProcessor.

This processor only makes sense if the created bend dummy vertices are later removed again and edges are reconnected to their original ports. Thus, any phase that depends on this processor also depends on the NorthSouthPortPostprocessor.

As its preconditions, this processor expects the graph to be layered.

As its postconditions, no edges are connected to northern or southern ports anymore, and bend dummy vertices may have been added to the graph.

Since the port order is important and self-loops involving northern or southern ports require some preprocessing, this processor depends on the PortListSorter and on the SelfLoopProcessor.

Port List Sorter

Oftentimes, algorithms iterate over the ports of a vertex in a certain order that depends on the port's side and position. Instead of having each such algorithm sort the port lists itself, the lists are sorted by this processor. Sorting the ports only makes sense for vertices whose port constraints are set to at least FixedOrder. Since this may not be the case for all vertices before phase 3, this processor may also be placed before phase 4, by which time all vertices have fixed port orders.

As its precondition, this processor expects the graph to be layered.

As its postcondition, vertices with fixed port orders have their port lists sorted by port side and position.

This processor does not depend on other processors.

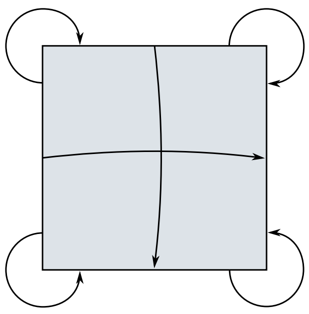


Figure 4.2. The SelfLoopProcessor ensures that self-loops involving ports on two different sides always go from left to right or top to bottom. This reduces the complexity of handling self-loops by constraining them to the cases shown here.

Port Side Processor

This processor iterates over the graph's vertices, looking for those with port constraints set to Free. If such a vertex is found, its ports are assigned to the western or eastern side depending on the number of incoming and outgoing edges: if there are more outgoing than incoming edges, a port is assigned to the eastern side; if there are at least as many incoming edges as outgoing edges, it is assigned to the western side.

As its precondition, this processor expects the graph to be layered.

As its post condition, all vertices have their port constraints set to at least FixedSides.

This processor does not depend on other processors.

Self-Loop Processor

The task of handling self-loops is split between this processor and the NorthSouthPortPreprocessor, which phases that depend on this processor also depend on. First, this processor ensures that self-loops involving two port sides always go from left to right or from top to bottom by reversing self-loop edges until they conform to the cases shown in Figure 4.2. It then looks for self-loop edges going from a western to an eastern port and splits them up by inserting an edge dummy vertex. Self-loops involving northern or southern ports are then left to the NorthSouthPortPreprocessor, while self-loops involving only eastern or only western ports are expected to be handled by the edge router.

As its precondition, this processor expects the graph to be layered.

As its postcondition, some edges may have been reversed, and edge dummy vertices may have been inserted.

Since the InvertedPortProcessor would try to process edges originating at a western port, it is run prior to this processor and expected to leave self-loops untouched.

Hyperedge Dummy Merger

This processor iterates over the graph's vertices, looking for pairs of edge dummy vertices placed right next to each other. If they belong to two long edges coming from the same source port or going into the same target port, the vertices are merged, and the edge is reconnected accordingly.

As its preconditions, this processor expects the graph to be layered, vertex orders in each layer to be fixed, and edge dummies belonging to a long edge to be annotated with that edge's original source and target ports.

As its postcondition, some edge dummy vertices may have been merged.

Since the vertex orders are expected to be fixed, the InLayerConstraintProcessor must be run prior to this processor.

In-Layer Constraint Processor

As described in Section 3.6, northern and southern hierarchical ports are handled by placing dummy vertices in the top and bottom regions of layers. *In-layer constraints* are attached to these vertices, specifying the region in a layer the vertices are to be placed in. Satisfying in-layer constraints is the responsibility of crossing reduction algorithms. However, our algorithm does not yet support them and thus depends on this processor to rearrange vertices until in-layer constraints are satisfied.

The processor iterates over the vertices in each layer, moving them to their respective regions, if necessary. The relative order of the vertices in each region remains unchanged.

As its preconditions, this processor expects the graph to be layered, and crossing reduction to have been applied.

As its postcondition, vertices are ordered such that in-layer constraints are satisfied.

This processor has no dependencies on other processors.

Port Position Processor

At some point, the exact coordinates of each port must be calculated, relative to its vertex. While this task was previously done during vertex placement, it is now performed by this processor. For each side of a vertex, the ports on that side are distributed evenly.

As its preconditions, this processor expects the graph to be layered, and vertices to have their port constraints set to at least FixedOrder.

As its postcondition, all ports have their positions calculated, and vertices have their port constraints set to FixedPos.

This processor does not depend on other processors.

Vertex Margin Calculator

To leave enough space around a vertex for its label, its ports, and its port labels not to be obstructed, a bounding box is calculated for each vertex, which can be done once its port positions are known. This processor iterates over the graph's vertices, calculating the bounding boxes and attaching the margin information to each vertex. The margin is later taken into account during vertex placement and edge routing.

As its preconditions, this processor expects the graph to be layered and port positions to be fixed.

As its postcondition, margin information are attached to each vertex to form a bounding box around the vertex, its label, its ports, and its port labels.

Since port positions must be fixed when this processor is run, it depends on the PortPositionProcessor.

Hierarchical Port Dummy Size Processor

If multiple hierarchical dummy vertices representing northern or southern hierarchical ports are placed in the same layer, their x coordinate needs to be adjusted to keep edges from overlapping later. This processor iterates over the graph's vertices looking for such vertices and adjusts their size accordingly. By doing that, we get differing x coordinates since the edge routing phase centers each vertex in its layer.

This processor is tightly related to the HierarchicalPortConstraintProcessor, the HierarchicalPortOrthogonalEdgeRouter, and the HierarchicalPortPositionProcessor. Phases either rely on all or none of them.

As its preconditions, this processor expects the graph to be layered, vertices to have their y coordinates set, and vertices to be ordered such that in-layer constraints are satisfied.

As its postcondition, hierarchical dummy vertices have their widths modified so that edges do not overlap when routing hierarchical edges using the HierarchicalPortOrthogonalEdgeRouter.

This processor has no dependencies on other processors.

Hierarchical Port Position Processor

The vertical position of western and eastern hierarchical dummy vertices cannot be arbitrarily set if port constraints are set to at least FixedRatio, thus restricting the vertex placement phase's freedom in assigning y coordinates. However, if the vertex placement phase does not support fixed coordinates, it depends on this processor to fix its results. The processor iterates over all vertices, looking for hierarchical dummy vertices that represent eastern or western hierarchical ports and whose coordinates are fixed, correcting them if necessary.

As its precondition, this processor expects the graph to be layered.

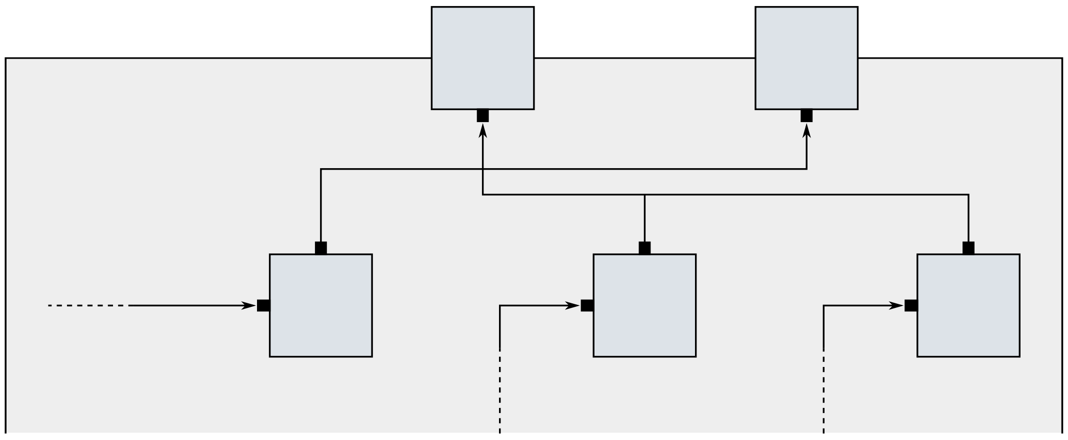


Figure 4.3. To route edges incident to northern hierarchical ports, two layers of vertices are created on which the orthogonal edge routing algorithm is run. In this example, the two upper vertices are dummy vertices originally created for hierarchical ports, while the three lower vertices are dummy vertices created in their stead.

As its postcondition, all hierarchical dummy vertices that represent eastern or western hierarchical ports have valid coordinates.

This processor has no dependencies on other processors.

Hierarchical Port Orthogonal Edge Router

To keep the edge routing phase as simple as possible, routing edges between hierarchical ports and their dummy vertices is done in a separate processor. Producing the routing takes five steps.

The first step restores the original hierarchical dummy vertices created for northern and southern ports while importing the graph. These vertices are viewed as representing the hierarchical ports, and the vertices created in their stead are connected to them. As Figure 4.3 shows, the vertices form two layers to route the edges between.

The second step calculates the coordinates for the original dummy vertices, thus determining where the hierarchical ports will later be placed. The y coordinates may not be final since the routing of edges to northern or southern hierarchical ports may change the graph's height.

The third step routes the edges between the original dummy vertices and the dummy vertices created in their stead, using the algorithm proposed by Sander that is also used by the orthogonal edge routing implementation [25].

The fourth step removes the dummy vertices that replaced the original dummy vertices, joining incident edges and adding appropriate bend points.

The fifth and final step does some final coordinate calculations. The routing of edges connected to northern and southern ports adds some height to the graph, thereby invalidating the previously calculated coordinates for hierarchical dummy vertices that represent eastern and western ports if port constraints are set to Fixed-

Ratio. This step performs necessary corrections, but usually introduces additional bend points in the process.

This processor is tightly related to the HierarchicalPortConstraintProcessor, the HierarchicalPortDummySizeProcessor, and the HierarchicalPortPositionProcessor. Phases either rely on all or none of them.

As its preconditions, this processor expects the graph to be layered, vertices to have x and y coordinates assigned, and bend points of all non-hierarchical edges to be set.

As its postconditions, all hierarchical dummy vertices now map onto actual hierarchical ports again, their coordinates specifying the final coordinates of the ports. Also, hierarchical edges have their bend points set.

This processor has no dependencies on other processors.

Long Edge Joiner

This processor removes edge dummy vertices inserted during the execution of the algorithm, joining their incident edges. Edge dummy vertices primarily stem from long edge splitting, but need not be. In fact, inverted port handling uses edge dummy vertices as part of the process because the way they are removed and their incident edges are joined is exactly what is required.

Any phase with a dependency on the LongEdgeSplitter also depends on the LongEdgeJoiner, as does any phase inserting edge dummy vertices or depending on another processor that inserts them.

As its preconditions, this processor expects the graph to be layered, the vertices to have proper x and y coordinates, and the edges to have all their bend points set.

As its postconditions, the graph does not contain any edge dummy vertices anymore, and the graph may not be properly layered anymore.

The HierarchicalPortOrthogonalEdgeRouter is run prior to this processor since it adds bend points to edges.

North South Port Postprocessor

This processor iterates over the graph's vertices, looking for bend dummy vertices created for northern or southern ports. If it finds one, its incident edges are reconnected to their original port, and a bend point is added at the port's x coordinate and the dummy vertex's y coordinate. This works best with orthogonal edge routing algorithms.

If a phase depends on the NorthSouthPortPreprocessor, it also depends on this processor.

As its preconditions, this processor expects the graph to be layered, vertices to have x and y coordinates, port positions to be fixed, and bend points of edges to be set.

Phase	Dependencies
Cycle removal	ReversedEdgeRestorer
Layer assignment	EdgeAndLayerConstraintEdgeReverser LayerConstraintProcessor
Crossing reduction	InLayerConstraintProcessor LongEdgeJoiner LongEdgeSplitter PortListSorter (before crossing reduction) PortSideProcessor
Vertex placement	PortPositionProcessor VertexMarginCalculator
Orthogonal edge routing	VertexMarginCalculator

Table 4.2. Basic list of dependencies of each phase on intermediate processors. The only edge routing implementation with dependencies is the orthogonal edge router. Depending on the graph's features, it may have additional dependencies shown in Table 4.3.

As its postconditions, the graph does not contain any bend dummy vertices anymore, and edges incident to northern and southern ports have their bend points properly set.

This processor does not depend on other processors.

Reversed Edge Restorer

The cycle removal phase removes cycles by reversing edges until the graph is acyclic, marking the edges as having been reversed. This processor iterates over all edges, restoring reversed edges to their original direction.

As its precondition, the processor expects the graph to be layered.

As its postcondition, the graph does not contain any reversed edges anymore. As a corollary, it may also not be acyclic anymore and may contain cycles again.

This processor has no dependencies on other processors.

▶ 4.3 A Problem-Oriented View

It is now time to take a step back from the details to look at how they work together to implement the concepts described in Chapter 3. We start by getting an overview of the algorithm at its most basic form. Table 4.2 lists all dependencies the different phases have on intermediate processors independent of the graph's features.

The Algorithm in Its Most Basic Form

The first step usually consists of the EdgeAndLayerConstraintEdgeReverser reversing edges such that a valid layering can be produced in the presence of layer constraints. Only then can the cycle removal phase make the graph acyclic: reversing edges may already have removed some of them.

Next up is the layer assignment phase, which produces a layering that may or may not adhere to layer constraints set on vertices. To take care of that, the LayerConstraintProcessor looks for violated constraints and moves affected vertices accordingly. Once the layer assignment is final, the LongEdgeSplitter inserts edge dummy vertices to turn the layering into a proper layering. The PortSideProcessor then makes sure that every vertex has its port constraints set to at least FixedSides, and the PortListSorter takes care of sorting port lists, which is important for the barycenter method of crossing reduction.

With port lists sorted, the crossing reduction phase can be run. Since our implementation does not respect in-layer constraints yet, the InLayerConstraintProcessor makes sure they are satisfied, followed by the PortPositionProcessor, which calculates final port positions, and the VertexMarginCalculator, which calculates the margin to be left around vertices.

The vertex placement phase then assigns y coordinates to all vertices, and the orthogonal edge routing phase calculates a routing for all edges as well as x coordinates for all vertices.

To wrap things up, the LongEdgeJoiner removes all edge dummy vertices and the ReversedEdgeRestorer restores reversed edges to their original direction.

Advanced Algorithm Configurations

Support for hyperedges, inverted ports, northern and southern ports, and hierarchy is largely a matter of edge routing. The orthogonal edge router implemented in KLay Layered supports all of these, but needs additional intermediate processors to do so, as does the vertex placement implementation to support hierarchy. For performance reasons, adding dependencies on these processors is subject to the graph actually requiring them. Table 4.3 lists the additional dependencies by graph features, and through the rest of this chapter we look at each one of them.

The existence of hyperedges depends on whether or not there are ports in the graph with multiple incident edges. If there are, the HyperedgeDummyMerger processor is inserted before phase four to merge edge dummy vertices.

Once vertices with port constraints set to something other than Free are present in the graph, it may contain inverted ports. However, there is one other condition that, if met, can lead to inverted ports even though all port constraints are set to Free. Let $p \in P$ be a port with one incident edge $e \in E$, attached to a vertex with port constraints set to Free. If e is incoming, p is placed on the western side; if e is outgoing, p is placed on the eastern side. If the cycle removal phase decides to reverse e, the placement of p changes accordingly, but does not become an inverted port. Now suppose that p has $n > 3$ incident edges $e_1, \ldots, e_n$, all of them incoming and thereby placing p on the western side. If the cycle removal phase decides to reverse one of the incoming edges, p is still placed on the western side, but has an outgoing edge now, making it an inverted port even though port

Graph Feature	Additional Dependencies
Ports with multiple incident edges	HyperedgeDummyMerger
Non-free port constraints or ports with multiple incident edges	InvertedPortProcessor
Northern / southern ports	NorthSouthPortPostprocessor NorthSouthPortPreprocessor
Hierarchy	HierarchicalPortConstraintProcessor HierarchicalPortDummySizeProcessor HierarchicalPortOrthogonalEdgeRouter HierarchicalPortPositionProcessor

Table 4.3. The algorithm supports port constraints and hierarchy, but needs the help of several intermediate processors to do so. Depending on whether a graph exhibits a certain feature or not, it specifies dependencies to additional intermediate processors.

constraints are set to Free. Thus, if there is a port with multiple incident edges, this may also lead to inverted ports. If at least one of these two conditions is met, the InvertedPortProcessor is inserted before the crossing reduction phase to add the required edge dummy vertices. The LongEdgeJoiner later removes the created edge dummy vertices.

Contrary to inverted ports, the presence of northern and southern ports requires vertices with port constraints set to at least FixedSides, and with ports already placed at the northern or southern side (the algorithm will never decide to place a port on something other than the eastern or western side if it is not already there). If a vertex with such ports is found, the NorthSouthPortPreprocessor is inserted before the crossing reduction phase and the NorthSouthPortPostprocessor is inserted after the edge routing phase. The preprocessor creates the necessary bend dummy vertices, allowing the crossing reduction phase to determine their order. After edges are properly routed, the postprocessor removes the dummy vertices and adds the required bend points to their incident edges.

With hierarchical ports present, a dummy vertex is created for each port when importing the graph. However, as we saw in Section 3.6, this may not be enough. The HierarchicalPortConstraintProcessor, inserted before the crossing reduction phase, takes care of setting appropriate vertex successor and in-layer constraints, and replaces the dummy vertices created for northern and southern hierarchical ports, if necessary. Once the vertex orders and y positions are set, the HierarchicalPortDummySizeProcessor, inserted before the edge routing phase, takes care of resizing hierarchical dummy vertices created for northern and southern ports to keep edges from overlapping later. With port constraints set to at least FixedRatio, the y coordinates of hierarchical dummy vertices that represent eastern or western hierarchical ports are fixed; the HierarchicalPortPositionProcessor adjusts the y coordinate of hierarchical dummy vertices if the vertex placement phase did not respect these fixed coordinates. After all of the non-hierarchical edge routing is finished, the HierarchicalPortOrthogonalEdgeRouter, inserted after the edge routing phase, takes care of routing the edges incident to hierarchical ports, and of calculating the final positions of hierarchical ports.

5

Evaluation

The Enrichment Center promises to always provide a safe testing environment. In dangerous testing environments, the Enrichment Center promises to always provide useful advice. For instance, the floor here will kill you. Try to avoid it.

— *GLaDOS, Portal*

In this chapter, we take a look at the quality of layouts produced by KLay Layered as well as at its performance. We start by getting a feel for how we can judge the quality of layouts and go on to describe the kinds of diagrams used for the evaluation. Since we make use of diagrams shipped with Ptolemy, we then see how these are imported into our own data format. The chapter closes with a description of the evaluation process and finally the evaluation results.

▶ 5.1 Judging the Quality of Layouts

Talking about the quality of a diagram's layout is inherently hard: opinions about what constitutes a good layout differ from one person to the next. When we talk about the quality of a layout then, the discussion is prone to drift into unscientific subjectivity, which is not how we can arrive at meaningful statements about layout algorithms. Rather, we have to try and strive for some objective *aesthetics criteria* which we can agree have some significance with regard to layout quality.

As it turns out, finding such criteria is not very hard, and indeed we have already been introduced to two of them very early on: the requirement of data flow diagrams to have the vast majority of edges pointing in the same direction, and the requirement of vertices to not overlap each other. We can easily think of further criteria that influence the readability of a diagram, such as the number of edge crossings or the number of edge bends. Di Battista et al. list the following aesthetics criteria [4]:

- The number of crossings should be minimized, since it gets progressively harder to follow an edge with more other edges crossing it. Ware et al. suspected the crossing angle to be of some significance, but could not find much evidence in support of this [31]; since our algorithm routes edges orthogonally, there is only one possible crossing angle anyway.
- The number of bends should be minimized as it is easier to follow a straight line than a bent line.
- The sum of edge lengths and the maximum edge length should be minimized as it is easier to see where edges go if they are short.
- The area of the drawing should be minimized to keep the diagram compact for the user to be able to see as much of it as possible at a glance. Some applications require diagrams to fit onto a single printed page, but that is outside the scope of this work.
- The smallest angle between two edges incident on the same vertex should be maximized to keep the user from confusing edges. This criterion does not apply to orthogonal layout algorithms such as KLay Layered.
- The aspect ratio of the drawing should approach popular screen aspect ratios. That, too, is outside the scope of this work.
- Symmetry in the graph should be reflected in the drawing. Defining symmetry and finding symmetrical areas in a graph is a very hard problem and has not yet been considered in KLay Layered.

A number of additional aesthetics criteria have been suggested that vary in their applicability to our domain, among them the following:

- Related vertices should be clustered to emphasize their relationship, as proposed for instance by Taylor and Rodgers [30]. This hardly applies to data flow diagrams because it is in direct conflict with the layered approach.
- As Purchase suggests, vertices should be placed along the intersections of a two-dimensional grid to align and separate them [21]. The layered approach already places vertices below one another, but does not necessarily place them along horizontal lines.

Bennett et al. give a comprehensive overview of further aesthetics criteria not relevant to our application [2].

Layout algorithms cannot place equal emphasis on all aesthetics criteria since they often contradict one another. For instance, keeping the number of edge bends low is easy if edges are allowed to cross vertices (just place all vertices next to each other along a horizontal line), and keeping the diagram's size small is easy if the number of edge crossings and edge bends is not important. Research looking into how much the different criteria contribute to the understanding of diagrams can help guide the choice of which criteria to emphasize. Such research is usually done in the form of studies in which test subjects are given different graphs drawn according to different aesthetics criteria. The test subjects are then asked to either rate how pleasing the layout is, or to do certain tasks as fast as possible. It can be argued that the results of these studies are not very significant because it is difficult to produce two drawings of the same graph that only differ in one or two aesthetics criteria, but for now it is the best data there is.

Purchase et al. have found edge crossings, edge bends, and symmetry to be the most important criteria [22]. However, they have also found the relative importance of different criteria to depend on the diagram's domain: the criteria important to the understanding of class diagrams differ from the criteria important to the understanding of data flow diagrams. There have been no studies specific to data flow diagrams yet.

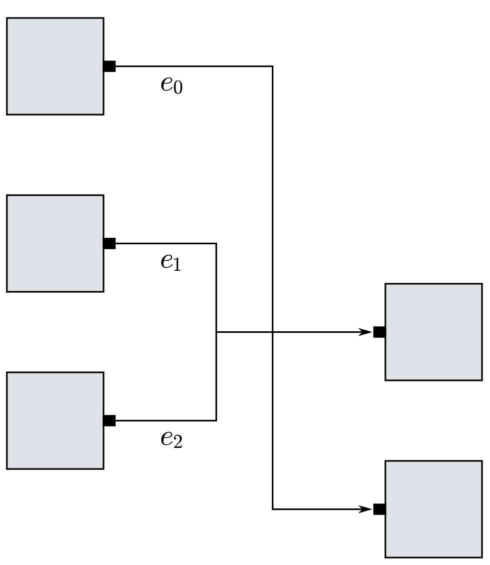
Layout Metrics

To use an aesthetics criterion for the evaluation of a layout algorithm, we cannot just look at diagrams and subjectively rate them with regard to the criterion; instead, we have to be able to measure it objectively. Furthermore, it has to be clear what exactly we are measuring. If, for instance, we talk about the number of edge bends, we have to decide beforehand if we mean the total number of bends, the average number of bends per edge, the minimum number of bends per edge, or the maximum number of bends per edge. A clearly defined measurement for an aesthetics criterion is called a *layout metric.*

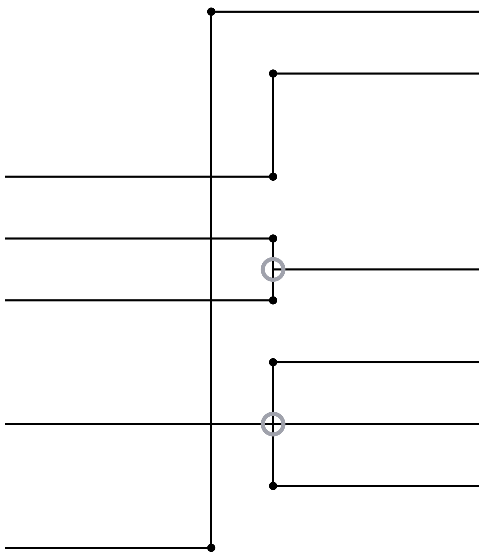
Helen Purchase has proposed a number of definitions for different layout metrics, all normalized to values between 0 and 1 [21]. However, this kind of scaling is not necessary for our evaluation. Taking the number of edge crossings as an example, a metric value of 1 would mean that the algorithm produced no crossings at all while a value of 0 would mean that the algorithm reached a calculated upper bound for crossings of the given graph. Such upper bounds though are only given for straight-line drawings, not for orthogonal drawings. Since we only want to compare the performance of KLay Layered and KLoDD, we avoid searching for new upper bounds and simply compare the absolute numbers.

For our evaluation, we finally settled on the following layout metrics, with a few hypotheses as to the results we expected:

- The number of edge crossings. While this metric is straightforward for diagrams whose edge segments do not overlap, there is one problem to be aware of with diagrams whose edges are routed orthogonally. Figure 5.1a has an example of a case where an edge crosses two other edges, causing two crossings. However, the two edges overlap, causing the viewer to only see one crossing. In our metric, we count the number of crossings as a viewer would see them.



(a) When counting crossings, a naive approach will usually count more crossings than a human viewer would see. In this case, e_0 crosses both e_1 and e_2, while when viewing the diagram we only see one crossing. Our metric would return one crossing for this diagram.



(b) When counting bend points, simply summing up all the bend points of all edges usually gives results that are too high. In this case, bend points that would be counted more than once are circled. The naive approach would count 12 bend points while our metric would count 10, as a viewer would.

Figure 5.1. Counting the number of crossings and bends poses problems when diagrams are drawn orthogonally.

Since we have put some emphasis on reducing the number of edge bends, we expected that this might come with a higher number of crossings. We thus expected KLay Layered to fair slightly worse in this metric than KLoDD.

- The number of edge bends. This metric poses a similar problem as the crossings metric, as Figure 5.1b shows. Here, too, we settle on the number of bends as perceived by a viewer as opposed to the sum of bends of all edges.

  Due to our work on linear segments, we expected KLay Layered to produce less edge bends than KLoDD.

- The area occupied by the diagram, defined as the product of its height and width.

  The edge spacing factor we introduced works towards making diagrams more compact. We thus expected KLay Layered to produce smaller diagrams than KLoDD.

- The diagram's aspect ratio, defined as its width divided by its height.

  We did not have any hypothesis for the results of this metric.

- The coverage of the diagram, defined as the ratio of the diagram's area that is covered by vertices and by areas where vertices cannot be placed. This metric gives an indication as to how well an algorithm makes use of available space.

  Since we expected KLay Layered to produce smaller diagrams than KLoDD, we also expected the coverage to be higher.

- The average and maximum edge length. Measuring this is straightforward.

  Since we expected KLay Layered to produce smaller diagrams than KLoDD, we also expected the average edge length to be smaller. We did not have any hypothesis concerning the maximum edge length.

- The number of feedback edges. Since our algorithms lay out data flow diagrams from left to right, the number of edges whose end point lies left of the start point is the number of feedback edges.

  We expected both algorithms to be similar concerning this metric.

- The number of layers. Defining this for different levels of hierarchy can be hard. We use a simple metric that returns the sum of the number of layers of each level of hierarchy.

  We expected both algorithms to be similar concerning this metric.

▶ 5.2 Finding Adequate Models

To evaluate the quality of the layouts generated by our algorithms, we of course need diagrams to evaluate them with. We make use of three sets of diagrams.

The first set consists of 540 randomly created diagrams with 10 to 50 vertices each and a maximum of between one and three outgoing edges per vertex. Port sides are chosen randomly: input ports are usually placed on the western side and output ports on the eastern side. This is switched around with a probability of 5%, and each port has a 25% chance of being assigned to the northern or southern side. The diagrams contain only one level of hierarchy.

For the remaining sets of diagrams, we focus on real-world diagrams. The second set therefore consists of a selection of 43 data flow diagrams which we obtained from an industry partner and were originally created with a function development tool for Electronic Control Units (ECUs). Each diagram contains one level of hierarchy and between 4 and 44 vertices. There are only a small amount of cycles, but almost every diagram has at least one vertex with northern or southern ports, usually more.

The third set consists of a selection of diagrams taken from the repository of examples that is shipped with the Ptolemy tool. We imported a selection of 141 models into our own data format, stripped of any annotation vertices because KLoDD does not support them. While the demo repository contains a lot more diagrams, we selected only those that imported correctly and that did not contain any but the simplest modal models. Contrary to the two other sets of diagrams, the Ptolemy

	Random	ECU	Ptolemy
Diagrams	540	43	141
Vertices	10–50 (30)	4–44 (15.67)	4–81 (16.28)
Compound vertices			1–10 (2.1)
Compound vertex children			1–43 (8.98)
Vertex degree	1–11 (3)	0–9 (1.77)	0–14 (1.66)
Edges	10–104 (45.01)	2–36 (14.42)	1–106 (15.39)
Self-loops			0–4 (0.09)
Connected components	1–5 (1.17)	1–11 (2.28)	2–19 (5.14)

Table 5.1. Properties of the three sets of diagrams used for the evaluation. Number ranges give the minimum and maximum, with the average value enclosed in parentheses.

diagrams usually contain multiple levels of hierarchy, with one hierarchical vertex averaging 8.98 child vertices, up to a maximum of 43. Importing the Ptolemy models was done using a transformation script the development of which is the topic of the next section.

Table 5.1 has some more details on the three sets of diagrams.

Both, KLoDD and KLay Layered, are primarily designed for the layout of data flow diagrams. Even though we have taken certain measures to generate the random diagrams to bear some similarity to data flow diagrams, they differ a lot from real-world diagrams. The analysis results obtained from random diagrams are thus less significant than the results obtained from the other two sets of diagrams.

The same reason also prevents us from using one of the popular graph libraries often used to evaluate layout algorithms, such as the Rome graph library [6] and the North or AT&T graph library [5].

For the performance evaluation, we use random diagrams generated at runtime. Section 5.4 contains details about this process.

▶ 5.3 Transforming Ptolemy Models

Ptolemy models are saved in the Modeling Markup Language (MoML) format, an XML-based format specifically designed for actor-based models. To use these diagrams for the quality evaluation, the KIELER tool has to be able to open them. Since there is no direct support for the MoML format yet, the Ptolemy models need to be imported into the actor-oriented model format supported by the KIELER tool, the KIELER Actor Oriented Modeling (KAOM) file format. This format is also based on XML, which makes the problem of importing MoML files one of transforming them from one XML-based format into another.

The KAOM file format is specified by a meta model that defines the kinds of objects a KAOM model can contain and how they can be put together. Figure 5.2 shows the different kinds of objects and their relationships: *entities* have *ports*, *links* connect linkable objects, and *relations* model hyperedges, as they do in Ptolemy. The meta model is specified using the Eclipse Modeling Framework (EMF), a popular

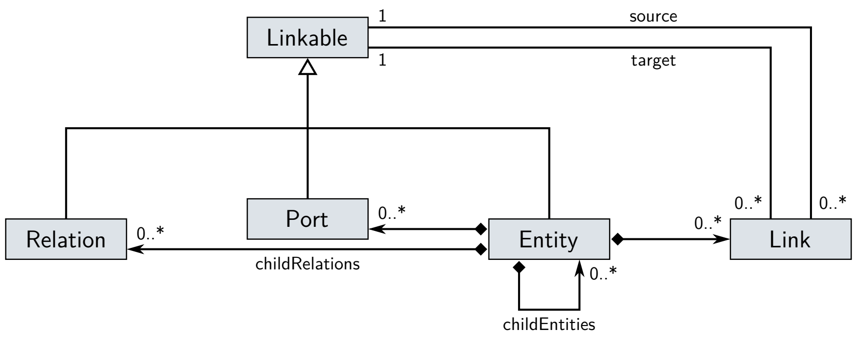


Figure 5.2. The KAOM meta model specifies the types of objects KAOM models may be composed of. *Entities* describe the actors in a model and may have *ports* while *relations* model hyperedges, as they do in Ptolemy. *Links* connect entities, ports, and relations.

Eclipse framework for specifying such meta models. EMF meta models can be used to automatically build graphical editors or to save models to and load them from XML-based files. Or, more important to the task at hand, to write transformations that turn a model based on another EMF meta model into one based on the KAOM meta model.

One framework for writing such transformations is called Xtend. It provides a functional language to traverse the object tree of a model, and to generate another model along the way.

Ptolemy is not based on the Eclipse platform, so the development team does not provide an EMF meta model for the MoML format. Luckily, the KIELER tool contains a basic MoML meta model imported from a Document Type Definition (DTD) document describing the MoML format. While the meta model is incomplete, it is sufficient for our requirements. A basic Xtend transformation was already available, but had to be improved and made easily available to the user.

Improving the Transformation

The most profound difference between MoML models and KAOM models is the way they represent edges. In KAOM models, edges are directed and always connect two ports. In MoML models, edges are undirected and connect a port with a *relation*, which is nothing more but a special object used to model hyperedges (all ports connected to a relation are interpreted as being connected directly to one another by the same hyperedge). Ports cannot be connected directly to each other: even if only two ports are to be connected, this still requires two edges and a relation, even though Ptolemy's graph editor will usually hide this from the user.

If edges are undirected in MoML models, but directed in KAOM models, we need some way to infer their direction during the transformation. We do this in two steps: the first step tries to annotate imported ports with information about whether they are input or output ports; the second step then iterates over all edges, trying to infer their direction.

To get our hands on port type information, we use the following method. When we encounter an actor, we ask a Ptolemy library included in the KIELER tool to instantiate it for us. However, the library only contains a subset of all available Ptolemy actors; even worse, models may define their own actors as compositions of existing actors, which the library may not be able to instantiate. If we are lucky, the library is successful and we can ask the instantiated actor for information about its ports. If not, we must do without explicit information.

To infer edge directions, we repeatedly iterate over the model. If we find an edge whose direction has not been inferred yet connected to a port with known port type, we set its direction accordingly. If we find an edge whose direction has already been inferred connected to a port of unknown type, we set its direction accordingly. If we do not find any of these but still have ports of unknown type left, we choose a port's type randomly and repeat the process. We are finished once all port types and edge directions have been set.

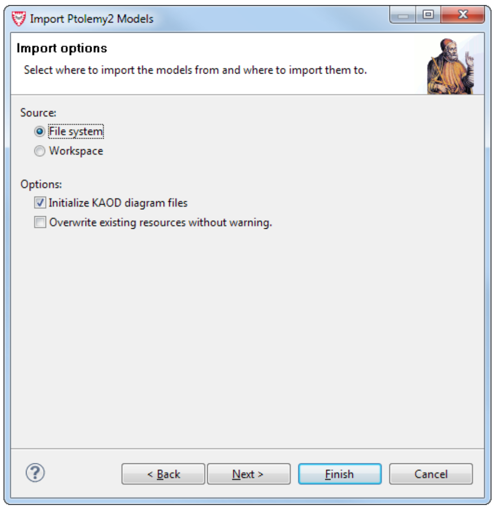
It is important to understand that this method is only a heuristic which may, and does, fail. However, the results are good enough for the imported models to be used to test our layout algorithms.

After all actors, ports, edges, and relations are transformed and edge directions inferred, we traverse the model again looking for relations that connect only two ports. These relations are replaced by single edges, since the KAOM meta model allows edges to directly connect two ports. Furthermore, each Ptolemy model contains a *director* that specifies the model of computation to be used. The director is not an actor, but rather specified as an annotation to the model. Since the KIELER tool does not display annotations in a diagram, we convert such director annotations to actors.

Finally, Ptolemy models can also contain comments that are usually used to explain what a model does and who developed it. These comments can contain formatting such as font sizes and colors, and are saved in the MoML model as embedded Scalable Vector Graphics (SVG) code. Since this is not part of the MoML meta model, the Xtend parser panics when it encounters SVG code and saves it in a list of unknown elements. We iterate over this list after the transformation itself is finished, looking for such SVG elements. If we find one that could define a comment, we add it to the KAOM model, stripped of all formatting which the KIELER tool does not support. Since the representation of comments may differ from model to model, this method does not always work. But again, it works well enough for our purposes.

Making the Transformation Available to the User

The Eclipse platform already specifies a mechanism for presenting import functionality to the user: *import wizards*, available under an "Import..." menu item. The import wizard created to import Ptolemy models consists of two steps, shown in Figure 5.3.



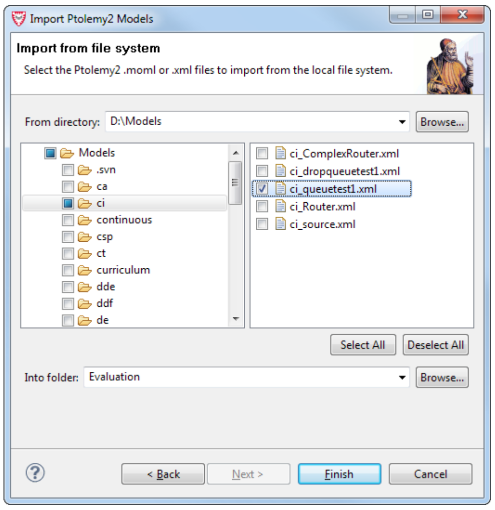
Import options

Select where to import the models from and where to import them to.

Source:
- (•) File system
- ( ) Workspace

Options:
- [✓] Initialize KAOD diagram files
- [ ] Overwrite existing resources without warning.

(a) During the first step, the user is asked for general information about the import process.



Import from file system

Select the Ptolemy2 .moml or .xml files to import from the local file system.

From directory: D:\Models

- Models
  - .svn
  - ca
  - ci
  - continuous
  - csp
  - ct
  - curriculum
  - dde
  - ddf
  - de

- [ ] ci_ComplexRouter.xml
- [ ] ci_dropqueuetest1.xml
- [✓] ci_queuetest1.xml
- [ ] ci_Router.xml
- [ ] ci_source.xml

Into folder: Evaluation

(b) During the second step, the user selects the models to import, and the location to import them to.

Figure 5.3. The Ptolemy import functionality is made available to the user as an import wizard, a standard mechanism specified by the Eclipse platform. The wizard has two steps.

The first step asks the user whether to import the model from the local workspace or from the file system, whether to overwrite existing models without further notice, and whether diagrams should automatically be created for the imported models.

The second step allows the user to specify the models to be imported, and the location in the workspace where the imported models are to be saved.

Any warnings or errors thrown during the import process are added to the error log, a standard facility provided by the Eclipse platform.

5.4 The Evaluation Process

We evaluate both, the quality of layouts generated by our algorithms, and their performance. In this section, we take a look at some of the details of the evaluation process before finally getting to the evaluation results in the next section.

Evaluating the Quality of Layouts

The quality evaluations are performed in the KIELER tool for which Martin Rieß has developed a graph analysis framework as part of his bachelor thesis [23]. All the metrics used for the evaluation are implemented as analyses for this framework. Since evaluating hundreds of models manually is not feasible, Rieß also developed a batch analysis tool to automate the process. The results are written into text files that can be easily parsed, processed, and turned into pretty diagrams with a spreadsheet program.

Setting	KLay Layered	KLoDD
Border spacing	20.0	20.0
Edge spacing factor	0.5	
Separate connected components	yes	
Spacing	20.0	20.0
Thoroughness	100	

Table 5.2. The settings used during the quality evaluation. Only the spacing and border spacing settings are provided by KLoDD.

Most of the metrics are straightforward to implement. However, the crossings and bends analyses have to be given some thought. To count crossings as described previously, we developed an algorithm that iterates over all edges, building a set of edge segments. All segments overlapping each other are then joined. Finally, the remaining segments are checked for crossings, which are only counted if the cross point does not happen to be and end point of one or more of the involved segments.

To count the number of bend points, we developed an algorithm that iterates over all edges, adding the discovered bend points to a hash set. The hash is computed from the coordinates of the bend points, thereby ensuring that multiple bend points at the same position are only added to the set once. The number of elements in the set is the number of bend points returned by the algorithm.

The analysis process has one disadvantage that comes into play with hierarchical diagrams. While both KLay Layered and KLoDD lay out each level of hierarchy separately, the analysis framework analyses all levels of hierarchy together. This means that when a hierarchical diagram is reported to have 80 vertices, the layout algorithms will usually only have been invoked on smaller subsets. This has to be taken account when interpreting the analysis results obtained from Ptolemy diagrams.

Both KLay Layered and KLoDD provide a number of settings to the user, all of which are left to their default values. The more important of these are given in Table 5.2. For KLay Layered, the network simplex algorithm is used in the layer assignment phase, and the orthogonal edge routing algorithm is used in the edge routing phase.

Evaluating the Performance of Algorithms

To evaluate the performance of our algorithms, we cannot simply press a layout button in the KIELER tool and measure the time it takes until the result is shown in the editor: a large part of that time is not due to the algorithm's performance, but rather has to be attributed to the task of showing the diagram to the user. We thus use a simple command-line program that measures only the time it takes for the algorithm to produce a result.

The tool generates random graphs for the algorithms to lay out. The graphs are generated in much the same way as the random graphs used for the quality evaluation: the number of outgoing edges is configurable, with the same probabilities of ports being inverted or placed on the northern or southern side. The diagrams could contain self-loops.

We use the tool for two different evaluations. For the first evaluation, we have the number of vertices fixed at 100 and make sure that each vertex has the same number of outgoing edges. We then run a number of tests with between 0 and 15 outgoing edges per vertex. For the second evaluation, we allow the number of outgoing edges per vertex to vary between 0 and 2. We then run a number of tests with between 10 and $10,000$ vertices.

For each of these tests, 10 different graphs are created for each graph size. The algorithms are run five times on each graph, with only the fastest run being recorded. The results are then averaged to arrive at a performance measurement for each graph size.

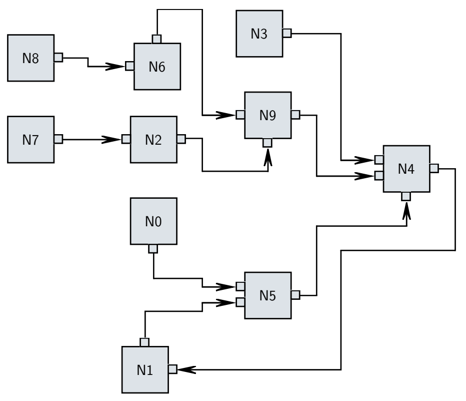
One thing to note is that the two algorithms are not run on the same graphs. Instead, new random graphs are created. This does not invalidate the results, however, due to the amount of graphs created per graph size and the averaged results.

The implementations of the different phases in KLay Layered are usually more expensive in terms of expected runtime performance than those in KLoDD. In particular, the network simplex algorithm for layer assignment used in KLay Layered is much more complex than the algorithm used in KLoDD. To make the two a little more comparable, we make KLay Layered use its longest path layer assignment algorithm. Other than that, all of the algorithm options are left to their default values.

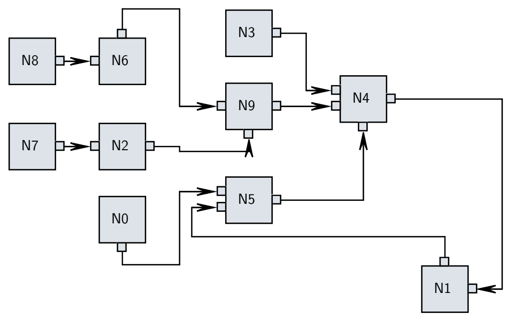
For the performance evaluation, we expected KLay Layered to be slower than KLoDD. The architecture results in a much higher number of iterations over the graph, and the algorithms implemented are more complex. In addition, KLay Layered uses much more dummy vertices, making iterations over the graph more time-consuming.

▶ 5.5 Evaluation Results

Before diving into the numbers, we take a look at two example diagrams. Figure 5.4 shows the layouts KLoDD and KLay Layered produce for one of the random diagrams. Both algorithms get away without any crossings, but while KLoDD produces 25 bends, KLay Layered only produces 17. For the cost of an additional layer, KLay Layered manages to keep the maximum number of vertices per layer at 3, one less than KLoDD. This makes the layout more compact, but a little wider. The diagram also shows some of the shortcomings still present with KLay Layered. First, looking at Figure 5.4b, `N1` and `N4` could well be placed in the same layer since the edge connecting the two connects two ports on the same side. Second, the edge between `N2` and `N9` has two obviously unnecessary bend points that could have been avoided by placing `N7` and `N2` a little lower. This is a problem of the vertex placement phase, the improvement of which was not part of this thesis. And third, while KLoDD reverses the edge between `N4` and `N1`, KLay Layered reverses the edge between `N1` and `N5`,



(a) The layout produced by KLoDD.



(b) The layout produced by KLay Layered.

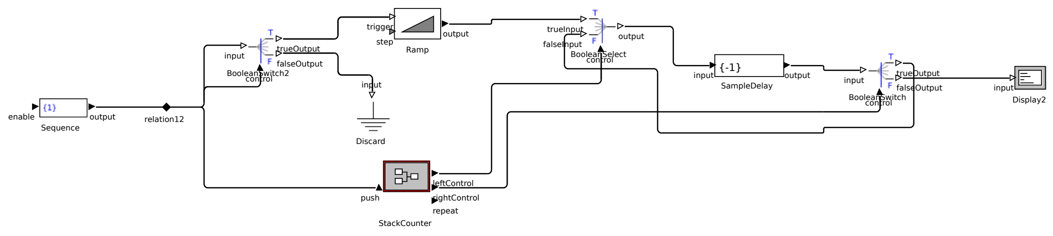
Figure 5.4. An example of the random diagrams, laid out with both KLoDD and KLay Layered.

requiring the edge to be routed around `N5`. Preferring to reverse feedback edges that go into eastern ports would make routing the edge around its target vertex—and thereby introducing further bend points—unnecessary.

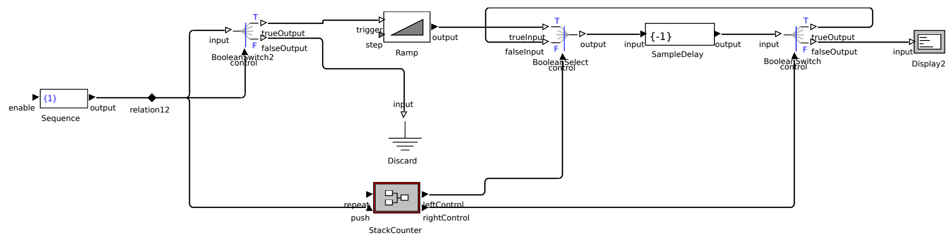
Figure 5.5 shows the layouts KLoDD and KLay Layered produce for one of the Ptolemy diagrams. Again, KLay Layered produces only 15 bend points while KLoDD produces 29. Due to the port configuration of the `BooleanSelect` operator, crossings cannot be avoided. KLay Layered fairs slightly better in this regard, producing only 1 compared to KLoDD's 3. Also, the layout produced by KLay Layered is slightly less wide than the one produced by KLoDD, at the cost of being a little higher. As noted before, readability is a term that is very hard to define; nevertheless, comparing the two diagrams we can certainly say that KLay Layered managed to produce a layout that is a lot easier to read. Probably, this is mainly due to the clear layout of the feedback loop, and to the low number of bend points, especially when it comes to the edges going into southern ports. Still, the layout produced by KLay Layered could have been even better. By placing `BooleanSwitch2` a little higher, the edge that connects it with `Ramp` could have been drawn as a straight line. Even worse, the edge that connects `StackCounter` with `BooleanSelect` would only have to have one bend point.

Now that we have seen two examples, it is time to move on to the evaluation's results.

During the development of KLay Layered, some emphasis was put on keeping the number of edge bends low. This is especially evident in the inclusion of bend dummy vertices into linear segments during vertex placement. We thus expected that the number of crossings produced by KLay Layered would be slightly higher than the number of crossings produced by KLoDD. As Figure 5.6 shows, this is only true for the random diagrams and the ECU diagrams—and even here KLay Layered fairs slightly better then KLoDD for very small diagrams. However, KLay Layered produces almost consistently better results for the Ptolemy set of diagrams. As noted previously, this may be because larger Ptolemy diagrams are hierarchically



(a) The layout produced by KLoDD.



(b) The layout produced by KLay Layered.

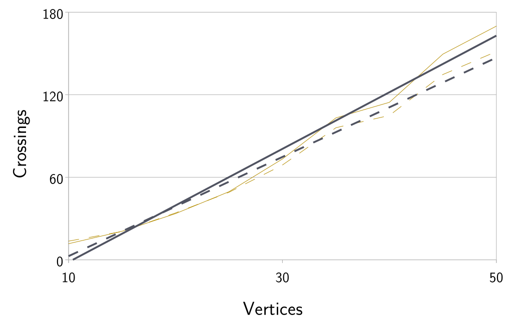
Figure 5.5. An example of the Ptolemy diagrams, laid out with both KLoDD and KLay Layered.

composed of smaller diagrams, which would support the results obtained from the random diagrams and ECU diagrams.

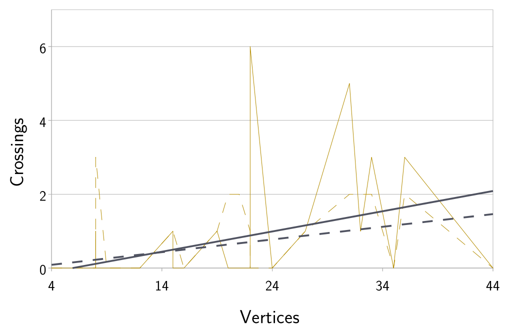
The analysis results concerning the number of bends was less mixed, with KLay Layered almost always producing better results than KLoDD. This is not surprising: the impact of including bend dummy vertices into linear segments has already been mentioned, and dropping the KLoDD method of vertex-local edge routing also removes a lot of unnecessary bend points.

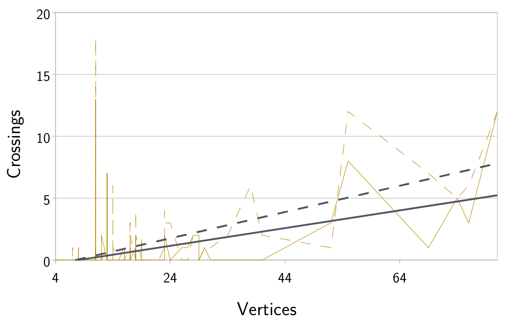
As far as the average edge lengths are concerned, we expected the work done to reduce the spacing between edges and to optimize the edge routing between layers to reduce them somewhat. However, we did not expect the difference to be as large as the results show them to be. As Figure 5.8 shows, KLay Layered produces considerably shorter edges than KLoDD. As we will see, this also has an impact on both, diagram size and aspect ratio.

Further analysis results are summarized in Table 5.3. For most of these, no significant differences can be made out between the two algorithms, which is as expected for the most part. The maximum edge length produced is usually lower in diagrams laid out using KLay Layered, with the notable exception of the ECU diagrams. It is not clear why that is the case, but could be due to the fact that the average number of layers is higher for these diagrams as well. As already mentioned, KLay Layered produces significantly smaller diagrams, except for the Ptolemy set of diagrams. This might be due to the fact that KLay Layered uses more spacing than KLoDD when it comes to hierarchical vertices. Finally, the aspect ratio analysis shows that KLay Layered produces narrower diagrams than KLoDD. We attribute that to the already mentioned work done to reduce the edge spacing and the edge routing between layers.



(a) Random diagrams.





(b) ECU diagrams.

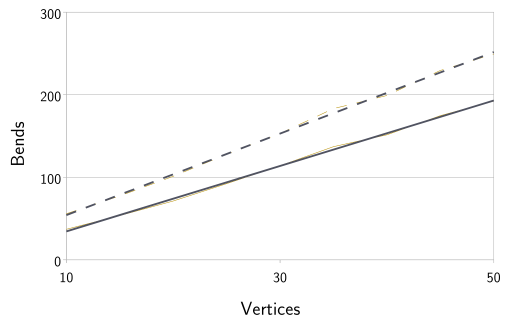
(c) Ptolemy diagrams.

Figure 5.6. The number of edge crossings produced. Solid lines show the results for KLay Layered, dashed lines show the results for KLoDD. Thick lines are trend lines while thin lines show the original data.

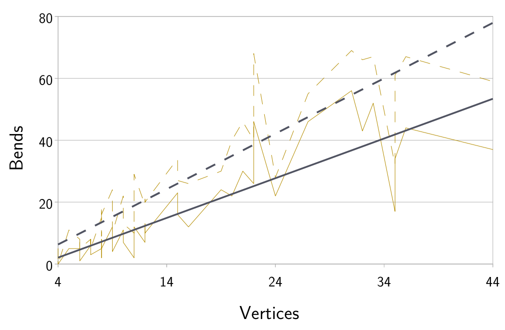
	Random	ECU	Ptolemy
Max. edge length	1405.57 / 1760.49	289.6 / 136.6	639.12 / 651.51
Feedback edges	5.1 / 5.18	0.35 / 0.35	0.56 / 0.55
Layers	11.38 / 11.3	7.02 / 5.98	10.13 / 9.58
Area	1,013,411 / 1,506,124	209,620 / 222,508	782,124 / 682,582
Aspect ratio	1.9 / 2.31	2.66 / 2.99	2.4 / 3
Coverage	0.11 / 0.08	0.21 / 0.2	0.28 / 0.33

Table 5.3. Further results of the evaluation. The numbers are averaged, with the first number giving the result for KLay Layered and the second number giving the result for KLoDD.

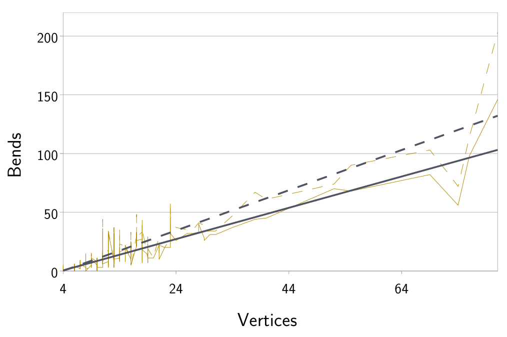
As already hinted at in Section 3.11, including barycenter associates during crossing reduction did not have much of an impact. Besides not including them, we ran experiments with a barycenter associate factor of 0.5 and 1.0. For ECU and Ptolemy diagrams, the difference in edge crossings produced lay between 0 and 3. Differences were only significant for the set of random diagrams, with between 0 and 170 crossings. However, with all sets of diagrams the factor that produced the best results differed: what would produce good results for one diagram did not necessarily produce good results for another diagram.



(a) Random diagrams.



(b) ECU diagrams.

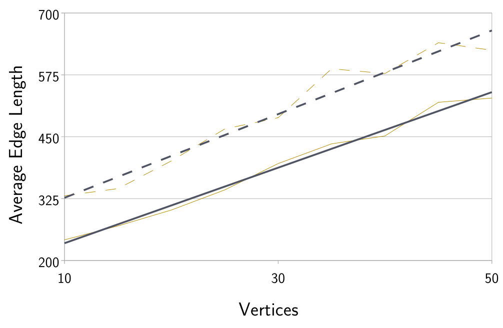


(c) Ptolemy diagrams.

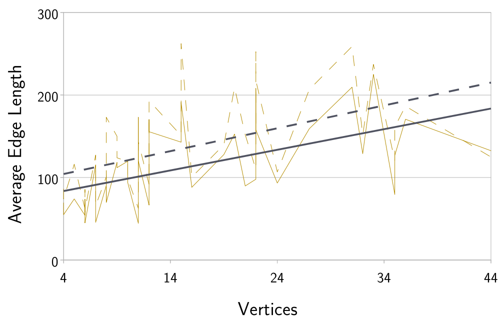
Figure 5.7. The number of edge bends produced. Solid lines show the results for KLay Layered, dashed lines show the results for KLoDD. Thick lines are trend lines while thin lines show the original data.

The results of the performance analysis were somewhat surprising. While we expected KLay Layered to perform worse in this experiment than KLoDD, the experimental results prove this hypothesis at least partly wrong. Figure 5.9a shows the results for a fixed number of vertices and a variable number of outgoing edges per vertex. Up to about three outgoing vertices, KLay Layered is about as fast as KLoDD. For more than three, the performance of KLay Layered skyrockets out of the realms of usability. However, the results for a fixed range of outgoing edges per vertex and a variable number of vertices, as shown in Figure 5.9b, tells another story. Here, KLay Layered still performs slightly worse than KLoDD for diagrams up to about $2,000$ vertices, but is a lot faster for larger diagrams.

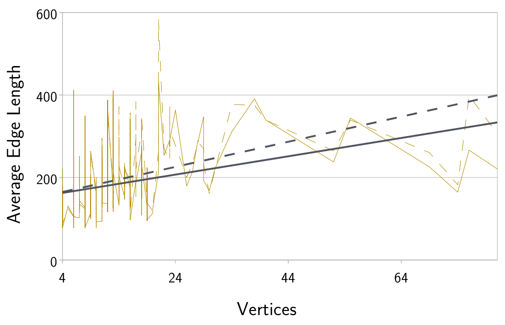
As Table 5.1 indicates, reasonable diagrams usually stay below 100 vertices, and average less than 2 incident edges per vertex. For these, KLay Layered is indeed slower than KLoDD, but not as much as we feared. Also, KLay Layered seems to be much more sensitive to higher numbers of edges than KLoDD, while the reverse is true for higher numbers of vertices. This may be due to the fact that a higher number of edges in our test sets also means a higher number of inverted, northern, and southern ports. While KLoDD handles these cases without introducing dummy vertices, KLay Layered makes heavy use of them, which may explain its sensitivity to the number of edges.



(a) Random diagrams.

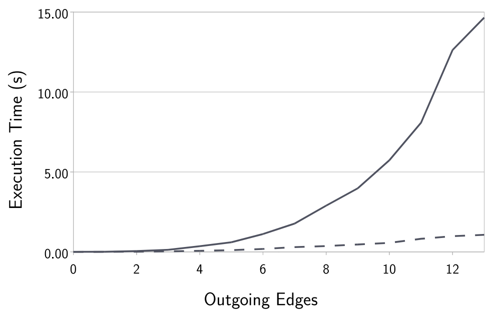


(b) ECU diagrams.

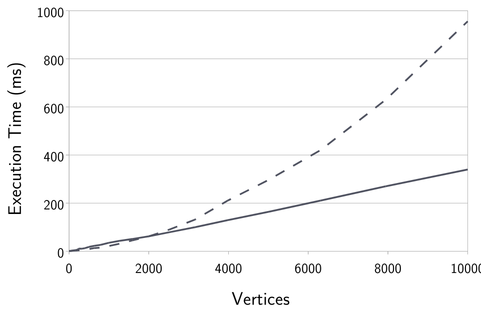


(c) Ptolemy diagrams.

Figure 5.8. The average edge lengths produced. Solid lines show the results for KLay Layered, dashed lines show the results for KLoDD. Thick lines are trend lines while thin lines show the original data.



(a) Fixed number of vertices, variable number of outgoing edges per vertex.



(b) Fixed range of outgoing edges per vertex, variable number of vertices.

Figure 5.9. Results of the performance evaluation. The solid line shows the results for KLay Layered, the dashed line shows the results for KLoDD.

6

Conclusion

This was a triumph. I'm making a note here, "Huge Success!"
— *GLaDOS, Portal*

In the final chapter we take a last summarizing look at the concepts introduced in this thesis. It closes with the mandatory look at what joyous tasks the future has in store for us.

▶ 6.1 Summary

In this thesis, the layer-based KLay Layered layout algorithm was extended to support port constraints, hierarchy, and self-loops, and improved in several further areas. A new architecture was developed for layout algorithms to adapt themselves flexibly to the layout tasks they are faced with. Compared to its predecessor, KLoDD, KLay Layered now produces more compact layouts with lower numbers of bends and, for small real-world diagrams, less crossings. Using KLay Layered on real-world diagrams has shown the layouts to be understandable and visually pleasing for the most part.

To achieve this, we extended existing concepts and introduced new ones. To handle inverted ports, the concept of a proper layering with in-layer edges was introduced and phases extended to support it. Handling northern and southern ports was achieved by introducing bend dummy vertices and making sure that bend dummy vertices created for different regular vertices could not be mixed. To keep the number of edge bends low, we made bend dummy vertices part of linear segments, introducing a safe way to split linear segments if necessary. We introduced a comprehensive method to route self-loops, and developed ways of handling hierarchical ports and their incident edges. Several smaller improvements also add to the quality of layouts generated by KLay Layered: the simplification of hyperedges, making sure that there is enough space to keep labels readable, and reducing distances to compact diagrams. Finally, we kept KLay Layered flexible by introducing the concept of intermediate processors, inserted as required.

There are two large applications for automatic layout algorithms: saving users the time and trouble of moving vertices around without actually doing any productive work at the semantic level, and serving as an enabling technology to higher-level technologies.

The first application hinges on users actually using automatic layout instead of doing the layout themselves. Users will only use it when it produces layouts that are good enough for users to be pleased. If the layout produced by a click on a button is almost as good or even better than what a user can produce by moving vertices around for half an hour, automatic layout will be used. While there are a lot of examples of diagrams where KLay Layered produces very pleasing results, there are on the other hand still a lot of examples of diagrams KLay Layered does not work as well with.

Laying out data flow diagrams, as laying out diagrams in general, is a very hard problem. The usual divide-and-conquer approach so popular in computer science does help, but leaves us with several more very hard problems. Being as hard as it is, solutions to this problem still leave a lot of room for improvement.

The second application can be less demanding. Generating diagrams of textually specified models on the fly, for instance, can benefit greatly from automatic layout. Being a supporting feature, the requirements are lower than in the first application, and automatic layout is already being successfully used in this area—much to the benefit of users.

All in all, automatic layout algorithms are an important enough technology to justify further research, and existing algorithms, good as they may be, still offer much room for improvement.

▶ 6.2 Future Work

The problem of laying out diagrams in a way that humans can understand them and like the layout enough to actually make use of layout algorithms remains hard. In this section, we take a short look at work that is still to be done.

Crossing reduction is a very hard problem to solve well. Even though we have heuristics that give reasonably good results, there is a lot of room for improvement. In particular, it would be interesting to investigate if orthogonal edge routing calls for specialized heuristics. Also, our way of respecting constraints is built on top of finished crossing reduction algorithms. It would be interesting to see if better results could be achieved if the crossing reduction algorithm itself already took constraints into account.

Another hard problem is that of vertex placement. Diagrams laid out by KLay Layered sometimes suffer from bend points that can easily be removed by moving vertices around slightly. Often, algorithms try to minimize the length of vertical edge segments, as seen in Figure 5.4b. To the user, however, such small displacements look worse than larger ones. Small displacements more often than not suggest easy ways of getting rid of the displacement in the first place. Doing that usually involves moving whole parts of the diagram around, though, as the edge between `BooleanSwitch2` and `Ramp` in Figure 5.5b shows. Such cases are very obvious to users, but hard to find algorithmically. Further research should be invested to find ways of improving this. Furthermore, a lot of the perceived beauty of a diagram's layout stems from the placement of vertices relative to each other. It should be investigated whether such considerations could be reasonably made during vertex placement.

As we have seen in Section 5.1, one important aesthetics criterion is symmetry. The diagram in Figure 1.1 is readily understandable largely to its symmetric layout. However, finding symmetry in a model is a very hard problem, and so far has not been considered by KLay Layered. Future research could investigate how symmetry detection can be incorporated into layer-based layout algorithms.

In Section 3.9 we have seen how we can prevent edges from crossing labels by calculating margins for vertices. What we have not done was to calculate explicit coordinates for labels—we simply assumed them to be fixed. Further research should explore the topic of label management: placing labels where they make the most sense, and even intelligently shortening labels that are too long.

The Ptolemy tool uses special vertices called *relation vertices* to model hyperedges. Currently, KLay Layered treats them as regular vertices, assigning them to a layer as usual. Layouts could be made more readable, however, if relation vertices were placed between layers, at points where their incident edges branch off. How this can be integrated well with the layered approach is currently unclear and needs further investigation.

Finally, different kinds of data flow diagrams may introduce further kinds of constraints. For instance, the LabVIEW tool developed by National Instruments introduces pairs of hierarchical ports with one port placed on the western side and the other placed on the eastern side. The vertical coordinate of both ports is required to be synchronized. Some research could be devoted to finding more examples of constraints not yet considered, and to developing ways to satisfy these constraints.

Bibliography

[1] Wilhelm Barth, Michael Jünger, and Petra Mutzel. Simple and efficient bilayer cross counting. In Michael Goodrich and Stephen Kobourov, editors, *Graph Drawing*, volume 2528 of *Lecture Notes in Computer Science*, pages 331–360. Springer Berlin / Heidelberg, 2002.

[2] Chris Bennett, Jody Ryall, Leo Spalteholz, and Amy Gooch. The aesthetics of graph visualization. 2007.

[3] Markus Chimani, Philipp Hungerländer, Michael Jünger, and Petra Mutzel. An SDP approach to multi-level crossing minimization. In *Proceedings of the Thirteenth Workshop on Algorithm Engineering and Experiments (ALENEX'11)*, pages 116–126, 2011.

[4] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs.* Prentice Hall, 1999.

[5] Giuseppe Di Battista, Ashim Garg, Giuseppe Liotta, Armando Parise, Roberto Tamassia, Emanuele Tassinari, Francesco Vargiu, and Luca Vismara. Drawing directed acyclic graphs: An experimental study. In *Graph Drawing*, volume 1190 of *LNCS*, pages 76–91. Springer, 1997.

[6] Giuseppe Di Battista, Ashim Garg, Giuseppe Liotta, Roberto Tamassia, Emanuele Tassinari, and Francesco Vargiu. An experimental comparison of four graph drawing algorithms. *Computational Geometry*, 7(5-6):303–325, 1997.

[7] 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.

[8] Peter Eades and Kozo Sugiyama. How to draw a directed graph. *Journal of Information Processing*, 13(4):424–437, 1990.

[9] 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.

[10] 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.

[11] Hauke Fuhrmann and Reinhard von Hanxleden. Taming graphical modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6394 of *LNCS*, pages 196–210. Springer, October 2010.

[12] 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.

[13] Michael R. Garey and David S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness.* W. H. Freeman & Co, New York, 1979.

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

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

[16] Gilles Kahn. The semantics of a simple language for parallel programming. In Jack L. Rosenfeld, editor, *Information Processing 74: Proceedings of the IFIP Congress 74*, pages 471–475. IFIP, North-Holland Publishing Co., August 1974.

[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 and Christian Dziobek. Effizientes Erstellen von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus. In *Tagungsband Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, pages 115–126, 2011.

[19] Edward A. Lee and David G. Messerschmitt. Synchronous data flow. In *Proceedings of the IEEE*, volume 75, pages 1235–1245. IEEE Computer Society Press, September 1987.

[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] Helen C. Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages and Computing*, 13(5):501–516, 2002.

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

[23] Martin Rieß. A graph editor for algorithm engineering. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2010.

[24] Georg Sander. A fast heuristic for hierarchical Manhattan layout. In *Proceedings of the Symposium on Graph Drawing (GD'95)*, volume 1027 of *LNCS*, pages 447–458. Springer, 1996.

[25] 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.

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] Martyn Taylor and Peter Rodgers. Applying graphical design techniques to graph visualization. In *Proceedings of the Ninth International Conference on Information Visualization (InfoVIS'05)*, pages 651–656, July 2005.

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

A

Technical Documentation Produced for KLay Layered

The guy who knows about computers is the last person you want to have creating documentation for people who don't understand computers.

— *Adam Osborne*

The documentation reproduced here was written for the group's wiki as part of this thesis. It provides a short introduction to the structure of KLay Layered and then goes on to describe each phase and each intermediate processor. The documentation was meant to be concise and precise. Concepts are not explained in great detail.

▶ A.1 Architecture

To get an idea of how KLay Layered is structured, take a look at the diagram shown in Figure A.1. The algorithm basically consists of just three components: layout phases, intermediate processors and an interface to the outside world. Let's briefly look at what each component does before delving into the gritty details.

The backbone of KLay Layered are its five layout phases, of which each is performing a specific part of the work necessary to layout a graph. The five phases go back to a paper by someone whose name I'm currently too lazy to look up. They are widely used as the basis for layout algorithms, and can be found in loads of papers on the topic. A detailled description of what each layout phase does can be found below.

Intermediate processors are less prevalent. In fact, it's one of our contributions to the world of layout algorithms. The idea here is that we want KLay Layered to be as generic as possible, supporting different kinds of diagrams, laid out in different kinds of ways. (as long as the layout is based on layers) Thus, we are well motivated to keep

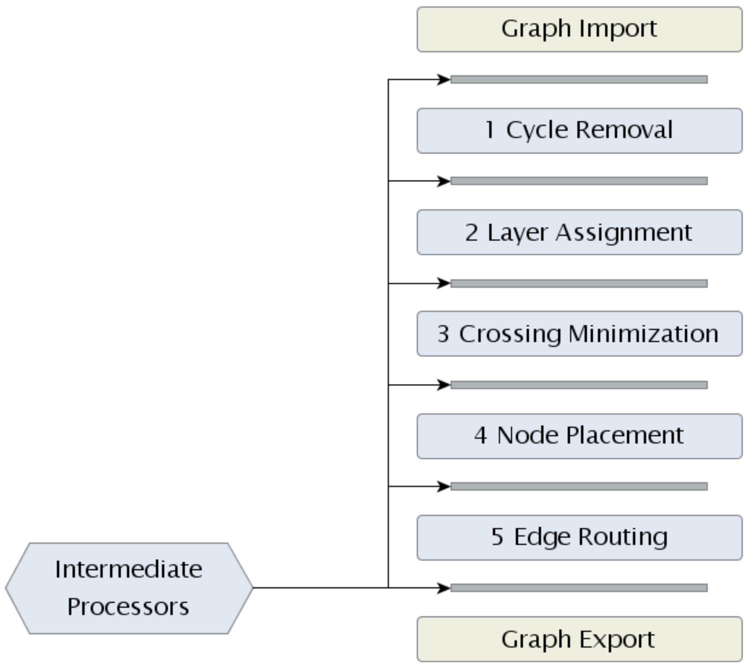


Figure A.1. The five phases of KLay Layered.

the layout phases as simple as possible. To adapt the algorithm to different needs, we then introduced small processors between the main layout phases. (the space between two layout phases is called a *slot*) One processor can appear in different slots, and one slot can be occupied by more than one processor. Processors usually modify the graph to be laid out in ways that allow the main phases to solve problems they wouldn't solve otherwise. That's an abstract enough explanation for it to mean anything and nothing at once, so let's take a look at a short example.

As will be seen below, the task of phase 2 is to produce a layering of the graph. The result is that each node is assigned to a layer in a way that edges always point to a node in a higher layer. However, later phases may require the layering to be *proper*. (a layering is said to be proper if two nodes being connected by an edge are assigned to neighbouring layers) Instead of modifying the layerer to check if a proper layering is needed, we introduced an intermediate processors that turns a layering into a proper layering. Phases that need a proper layering can then just indicate that they want that processor to be placed in one of the slots.

The interface to the outside world finally allows us to plug our algorithm into the programs wanting to use it. While not strictly part of the actual algorithm, that interface allows us to lay out graphs people throw at us regardless of the data structures used to represent those graphs. Internally, KLay Layered uses a data structure called `LGraph` to represent graphs. (an `LGraph` can be thought of as a lightweight version of `KGraph`, with the concept of layers added) All a potential user has to do is write an import and export module to make his graph structure work

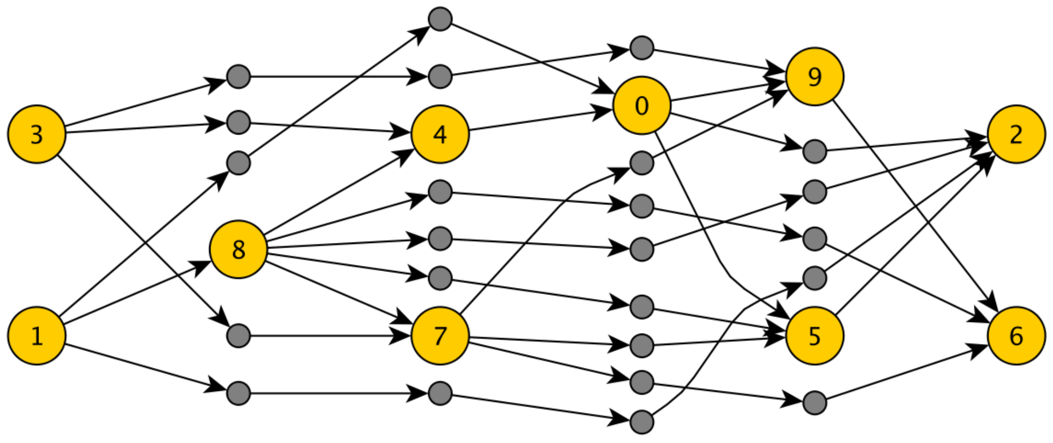


Figure A.2. A properly layered graph, with dummy nodes inserted.

with KLay Layered. Currently, the only module available is the `KGraphImporter`, which makes KLay Layered work with KIML.

Dummy Nodes

Before getting down to it, one other thing is worthy of our attention: dummy nodes. Dummy nodes are nodes inserted into the graph during the layout process. They were not in the original graph that is to be laid out, and are removed prior to the layout being applied to the original graph structure. So why then do we need them?

The different layout phases often have very specific requirements concerning the graph's structure. Real-world graphs usually don't meet these requirements. We could of course respond to that by enabling the phases to cope with these kinds of adverse conditions. But it's much simpler to just insert a few dummy nodes to make the graph fit the requirements.

See the graph in Figure A.2 for an example. The orange nodes were layered, but the layering was by no means a proper layering. Thus, gray dummy nodes were added.

In KLay Layered, we make extensive use of dummy nodes to reduce complex and very specific problems such that we can solve them using our general phases. One example is how we have implemented support for ports on the northern or southern side of a node.

▶ A.2 Phases

This section describes the algorithm's five main phases and the available implementations. You can find a list and descriptions of the intermediate processors below.

Phase 1: Cycle Removal

The first phase makes sure that the graph doesn't contain any cycles. In some papers, this phase is an implicated part of the layering. This is due to the supporting function cycle removal has for layering: without cycles, we can find a topological ordering of the graph's nodes, which greatly simplifies layering.

An important part to note is that cycles may not be broken by removing one of their edges. If we did that, the edge would not be routed later, not to speak of other complications that would ensue. Instead, cycles must be broken by reversing one of their edges. Since the problem of finding the minimal set of edges to reverse to make a graph cycle-free is NP-hard, (or NP-complete? I don't remember from the top of my head) cycle removers will implement some heuristic to do their work.

The reversed edges have to be restored at some point. There's a processor for that, called ReversedEdgeRestorer. All implementations of phase one must include a dependency on that processor, to be included after phase 5.

Precondition

- No node is assigned to a layer yet.

Postcondition

- The graph is now cycle-free. Still, no node is assigned to a layer yet.

Remarks

- All implementations of phase one must include a dependency on the ReversedEdgeRestorer, to be included after phase 5.

Current Implementations

- GreedyCycleBreaker. Uses a greedy approach to cycle-breaking.

Phase 2: Layering

The second phase assigns nodes to layers. (also called *ranks* in some papers) Nodes in the same layer are assigned the same x coordinate. (give or take) The problem to solve here is to assign each node x a layer i such that each successor of x is in a layer $j > i$. The only exception are self-loops, that may or may not be supported by later phases.

It must be differentiated between a *layering* and a *proper layering*. In a layering, the above condition holds. (well, and self-loops are allowed) In a proper layering, each successor of x is required to be assigned to layer $i + 1$. This is possible only for the simplest cases, but may be required by later phases. In that case, later phases use the LongEdgeSplitter processor to turn a layering into a proper layering by inserting dummy nodes as necessary.

Note that nodes can have a property associated with them that constraints the layers they can be placed in.

Precondition

- The graph is cycle-free.
- The nodes have not been layered yet.

Postcondition

- The graph has a layering.

Remarks

- Implementations should usually include a dependency on the LayerConstraintHandler, unless they already adhere to layer constraints themselves.

Current Implementations

- LongestPathLayerer. Layers nodes according to the longest paths between them. Very simple, but doesn't usually give the best results.
- NetworkSimplexLayerer. A way more sophisticated algorithm whose results are usually very good.

Phase 3: Crossing Reduction

The objective of phase 3 is to determine how the nodes in each layer should be ordered. The order determines the number of edge crossings, and thus is a critical step towards readable diagrams. Unfortunately, the problem is NP-hard even for only two layers. Did I just hear you saying "heuristic"? The usual approach is to sweep through the pairs of layers from left to right and back, along the way applying some heuristic to minimize crossings between each pair of layers. The two most prominent and well-studied kinds of heuristics used here are the barycenter method and the median method. We have currently implemented the former.

Our crossing reduction implementations may or may not support the concepts of node successor constraints and layout groups. The former allows a node x to specify a node $y \neq x$ that may only appear after x. Layout groups are groups of nodes. Nodes belonging to different layout groups are not to be interleaved.

Precondition

- The graph has a proper layering. (except for self-loops)
- An implementation may allow in-layer connections.
- Usually, all Nodes are required to have at least fixed port sides.

Postcondition

- The order of nodes in each layer is fixed.
- All nodes have a fixed port order.

Remarks

- If fixed port sides are required, the PortPositionProcessor may be of use.
- Support for in-layer connections may be required to be able to handle certain problems. (odd port sides, for instance)

Current Implementations

- LayerSweepCrossingMinimizer. Does several sweeps across the layers, minimizing the crossings between each pair of layers using a barycenter heuristic. Supports node successor constraints and layout groups. Node successor constraints require one node to appear before another node. Layout groups specify sets of nodes whose nodes must not be interleaved.

Phase 4: Node Placement

So far, the coordinates of the nodes have not been touched. That's about to change in phase 4, which determines the y coordinate. While phase 3 has an impact on the number of edge crossings, phase 4 has an influence on the number of edge bends. Usually, some kind of heuristic is employed to yield a good y coordinate.

Our node placers may or may not support node margins. Node margins define the space occupied by ports, labels and such. The idea is to keep that space free from edges and other nodes.

Precondition

- The graph has a proper layering. (except for self-loops)
- Node orders are fixed.
- Port positions are fixed.
- An implementation may allow in-layer connections.
- An implementation may require node margins to be set.

Postcondition

- Each node is assigned a y coordinate such that no two nodes overlap.
- The height of each layer is set.
- The height of the graph is set to the maximal layer height.

Remarks

- Support for in-layer connections may be required to be able to handle certain problems. (odd port sides, for instance)
- If node margins are supported, the NodeMarginCalculator can compute them.
- Port positions can be fixed by using the PortPositionProcessor.

Current Implementations

- LinearSegmentsNodePlacer. Builds linear segments of nodes that should have the same y coordinate and tries to respect those linear segments. Linear segments are placed according to a barycenter heuristic.

Phase 5: Edge Routing

In the last phase, it's time to determine x coordinates for all nodes and route the edges. The routing may support very different kinds of features, such as support for odd port sides, (input ports that are on the node's right side) orthogonal edges, spline edges etc. Often times, the set of features supported by an edge router largely determines the intermediate processors used during the layout process.

Precondition

- The graph has a proper layering. (except for self-loops)
- Nodes are assigned y coordinates.
- Layer heights are correctly set.
- An implementation may allow in-layer connections.

Postcondition

- Nodes are assigned x coordinates.
- Layer widths are set.
- The graph's width is set.
- The bend points of all edges are set.

Remarks

- None.

Current Implementations

- ComplexSplineRouter.
- OrthogonalEdgeRouter. Routes edges orthogonally. Supports routing edges going into an eastern port around a node. Tries to minimize the width of the space between each pair of layers used for edge routing.
- PolylineEdgeRouter.
- SimpleSplineEdgeRouter.

▶ A.3 Intermediate Processors

This section describes the intermediate processors that are available. For a description of what intermediate processors actually are, see Section A.1.

Each intermediate processor is described by its required preconditions, its postconditions, the slot where it should be placed in and dependencies to intermediate processors in the same slot. The descriptions are kept very brief, since layout processors are usually well documented. Programmers using layout processors need not worry about dependencies. However, when adding a new processor, dependencies matter. For more information, see the documentation of `IntermediateLayoutProcessor`.

Table A.1 provides an overview of all available layout processors and the slots they can be placed in. Note that a processor may appear in more than one slot.

Edge And Layer Constraint Edge Reverser

Edge constraints affect if a node may have only incoming or only outgoing edges. This processor reverses edges if necessary to respect the edge constraints.

Layer constraints can be seen as implicit edge constraints. If a node should be placed in the first layer, it may have only outgoing edges. Similar for the last layer.

Preconditions

- The graph is not layered yet.

Postconditions

- Nodes with edge or layer constraints have only incoming or only outgoing edges, as appropriate.

Slot

- Before phase 1.

Slot	Processor
Before phase 1	Edge And Layer Constraint Edge Reverser
Before phase 2	None.
Before phase 3	Hierarchical Port Constraint Processor Inverted Port Processor Layer Constraint Processor Long Edge Splitter North South Port Preprocessor Port List Sorter Port Side Processor SelfLoopProcessor
Before phase 4	Hyperedge Dummy Merger In-Layer Constraint Processor Node Margin Calculator Port Position Processor Port Side And Order Processor
Before phase 5	Hierarchical Port Dummy Size Processor Hierarchical Port Position Processor
After phase 5	Hierarchical Port Orthogonal Edge Router Long Edge Joiner North South Port Postprocessor Reversed Edge Restorer

Table A.1. Intermediate processors and where they can be placed.

Dependencies

- None.

Remarks

- Layerers should usually include a dependency on this processor.

Hierarchical Port Constraint Processor

This processor is concerned with hierarchical ports.

For eastern and western ports, the order of the nodes is fixed if the port constraints are at least at FixedOrder. This processor inserts the necessary in-layer successor constraints to ensure that this order is respected during crossing reduction.

For northern and southern hierarchical ports, we just need one hierarchical port dummy per hierarchical port in the simple cases. With port constraints at least at FixedOrder, however, we need to modify that approach a little. For each node connected to a hierarchical port on the northern or southern side, we insert a hierarchical port dummy in the following layer. We then remove the original hierarchical port dummy representing the port, setting the new dummy's `ORIGIN` property to that original dummy node. The old dummy nodes are inserted again by the HIERARCHICALPORTORTHOGONALEDGEROUTER. For all other port constraints, the original port dummy nodes are replaced by a single new dummy node, in a similar way as described above. This greatly simplifies hierarchical port dummy handling later on during edge routing.

Preconditions

- A layered graph.
- Long edge dummies have not yet been inserted.
- Layer constraints are satisfied.

Postconditions

- For graphs with port constraints at least at FIXEDORDER, northern and southern hierarchical port dummies are handled.

Slot

- Before phase 3.

Dependencies

- LayerConstraintProcessor

Remarks

- This processor is necessary to ensure proper functioning of the HIERARCHICALPORTORTHOGONALEDGEROUTER.

Hierarchical Port Dummy Size Processor

Sets the width of hierarchical port dummy nodes.

To see why this is necessary, let's step back for a minute and imagine three hierarchical northern port dummy nodes in the same layer. With the default hierarchical port edge router, what will happen is the following. Each each going into one of the nodes is routed upwards, the bend point being placed at the dummy node's input port. If all dummy nodes have the same width, these ports will have the same x coordinate - the edges incident to all three nodes will be routed on top of each other. To make the task of avoiding this easier, this processor sets the width in a way ensuring that the x coordinates of the three ports are as far apart as the edge spacing dictates.

Preconditions

- A layered graph.
- Nodes are assigned y coordinates.
- Nodes are not assigned x coordinates yet.
- Bend points for edges have not yet been set.
- Nodes are ordered such that in-layer constraints are respected.

Postconditions

- Hierarchical port dummies are assigned appropriate widths.

Slot

- Before phase 5.

Dependencies

- None.

Remarks

- This processor is required for HierarchicalPortOrthogonalEdgeRouter to function properly.

Hierarchical Port Orthogonal Edge Router

After edge routing, edges have only been routed inside the graph. What remains to be done is to route the edges to the hierarchical ports. This processor does just that, using an orthogonal edge routing approach. During that process, hierarchical port dummy nodes that map onto hierarchical port are assigned the coordinates of the hierarchical port, relative to the graph's content area and already corrected for the offset. The necessary bend points are added to the edges connected to hierarchical ports. Hierarchical port dummy nodes that don't map onto a hierarchical port are removed, their incident edges connected to the appropriate hierarchical port dummy node representing a hierarchical port.

This is the default edge router for edges incident to hierarchical ports. Other edge routers are free to come with an own implementation for routing hierarchical edges.

Preconditions

- A layered graph.
- Nodes are assigned y coordinates.
- The bend points of all internal edges are set.

Postconditions

- All hierarchical port dummy nodes left map onto an actual hierarchical port.
- The coordinates of hierarchical port dummy nodes specify the coordinates of their respective hierarchical port.
- All hierarchical port dummy nodes have a size of $(0, 0)$.
- Edges connected to hierarchical ports have their bend points set.

Slot

- After phase 5.

Dependencies

- None.

Remarks

- For anything other than free port constraints, this processor requires that ConstrainedHierarchicalPortProcessor has executed.
- This processor requires that HierarchicalPortDummySizeProcessor has executed.

Hierarchical Port Position Processor

If port constraints are set to at least FixedRatio, the node placement phase is not free to position external port dummies at will. If the node placement algorithm doesn't support fixed positions, including a dependency on this processor fixes the y positions of external port dummies representing western or eastern ports.

Preconditions

- A layered graph.
- Nodes are assigned y coordinates.
- External port dummies for western and eastern ports are placed in the first and last layer, respectively.

Postconditions

- The y coordinates of external port dummies are set as needed in the FIXED-RATIO and FIXEDPOS port constraint cases.

Slot

- Before phase 5.

Dependencies

- None.

Remarks

- If northern or southern external edge routing modifies the height of the diagram, the dummy node positions become invalid in the FIXEDRATIO case. They are then recomputed by the HIERARCHICALPORTORTHOGONALEDGER-OUTER.

Hyperedge Dummy Merger

Merges long edge dummy nodes with edges originally coming from the same port or going into the same port. The idea is to reduce the amount of edges in the diagram as much as possible.

Preconditions

- The graph is layered.
- Node orders are fixed.
- For long edge dummies to be joined, their `LONG_EDGE_SOURCE` and `LONG_EDGE_TARGET` properties must be set.

Postconditions

- Some long edge dummy nodes may have been merged.

Slot

- Before phase 4.

Dependencies

- INLAYERCONSTRAINTPROCESSOR

Remarks

- This processor only makes sense if the LongEdgeSplitter was used before.

In-Layer Constraint Processor

Makes sure that in-layer constraints are respected. This processor is only necessary for crossing minimizers that don't respect in-layer constraints.

Preconditions

- The graph is layered.
- Crossing minimization is finished.

Postconditions

- Nodes may have been reordered to match in-layer constraints.

Slot

- Before phase 4.

Dependencies

- None.

Remarks

- Crossing minimizers that don't support in-layer constraints must include a dependency on this processor. Other crossing minimizers should not depend on it.

Inverted Port Processor

Inserts odd port side dummy nodes to cope with odd port sides. Odd port sides are the eastern side for input ports and the western side for output ports. In both cases, the incoming or outgoing edges have to be routed around the node.

Preconditions

- The graph is layered.

Postconditions

- Odd port side dummy nodes are inserted for odd ports.
- The graph may contain new in-layer connections.

Slot

- Before phase 3.

Dependencies

- PortSideProcessor

Remarks

- The following phases must support in-layer connections for this to work.

Layer Constraint Processor

Nodes can have a property associated with them that restricts the layers they can be placed in. They can be forced in the first or the last existing layer. They can also be forced into a newly created first or last layer, along with all other nodes with the appropriate property set. While they may not be treated differently by the layerer, this processor moves them into the layer they should be placed in. A node placed in the first layer must have only outgoing edges; similarly, nodes placed in the last layer must have only incoming edges. This processor assumes that as a precondition.

Preconditions

- The graph is layered.
- Nodes to be placed in the first layer only have outgoing edges.
- Nodes to be placed in the last layer only have incoming edges.

Postconditions

- Nodes with layer constraints have been placed in the appropriate layers.

Slot

- Before phase 3.

Dependencies

- ConstrainedHierarchicalPortProcessor

Remarks

- Layerers should usually include a dependency on this processor, unless they already adhere to layer constraints themselves.

- The LayerConstraintEdgeReverser ensures that this processor's preconditions are met. Thus, layerers should also include a dependency on that processor.

Long Edge Joiner

Removes all long edge dummy nodes, joining their edges together.

Preconditions

- The graph is layered.
- Nodes are assigned x and y coordinates.
- The bend points of all edges are set.

Postconditions

- There are no long edge dummy nodes anymore.
- The graph may not be properly layered anymore.

Slot

- After phase 5.

Dependencies

- HierarchicalPortOrthogonalEdgeRouter

Remarks

- Since there are multiple processors that generate long edge dummies, this processor doesn't only make sense if the LongEdgeSplitter was used before.

Long Edge Splitter

Turns a layered graph into a properly layered graph by inserting long edge dummies where appropriate..

Preconditions

- The graph is layered.

Postconditions

- The graph is properly layered.

Slot

- Before phase 3.

Dependencies

- LayerConstraintProcessor

Remarks

- None.

Node Margin Calculator

Calculates node margins based on port positions and sizes and label positions and sizes.

Preconditions

- The graph is layered.
- Port positions are fixed.

Postconditions

- Node margins are properly set to form a bounding box around the node and its ports and labels.

Slot

- Before phase 4.

Dependencies

- PortPositionProcessor

Remarks

- None.

North South Port Postprocessor

Removes north / south port dummy nodes and routes the edges properly.

Preconditions

- The graph is layered.
- Nodes are assigned y coordinates.
- The bend points of all edges are set.
- Port positions are fixed.

Postconditions

- North / south port dummy nodes are removed, their edges being properly routed and connected.

Slot

- After phase 5.

Dependencies

- None.

Remarks

- This processor only makes sense if the NorthSouthPortPreprocessor was used before.

North South Port Preprocessor

Inserts dummy nodes to cope with northern and southern ports. Dummy nodes are assigned to layout groups identified by the node whose ports they were created from. Also, node successor constraints are set to keep north / south port dummy nodes in a certain order. This processor is capable of processing self-loops connecting two northern or two southern ports. For other kinds of self-loops, the SelfLoopProcessor may be required.

Preconditions

- The graph is layered.
- Port sides are fixed.

Postconditions

- North / south port dummy nodes have been inserted.
- No edge is connected to a northern or southern port any more.

Slot

- Before phase 3.

Dependencies

- PortListSorter
- SelfLoopProcessor

Remarks

- The dummy nodes must later be postprocessed by NorthSouthPortPostprocessor.
- A crossing minimizer must support layout groups and node successor constraints.

Port List Sorter

If a node already has a fixed port order, its port lists are sorted accordingly.

Preconditions

- The graph is layered.

Postconditions

- Nodes with fixed port orders have their port lists sorted accordingly.

Slot

- Before phase 3.
- Before phase 4.

Dependencies

- None.

Remarks

- It may make sense to use this processor in multiple slots. Once to ensure fixed port sides, once more to sort the port lists again for nodes that have just had their port orders fixed.

Port Position Processor

Calculates the exact coordinates a ports.

Preconditions

- The graph is layered.
- All nodes have a fixed port order.

Postconditions

- All nodes have fixed port positions.

Slot

- Before phase 4.

Dependencies

- None.

Remarks

- None.

Port Side Processor

Ensures that nodes have at least fixed port sides.

Preconditions

- The graph is layered.

Postconditions

- All nodes have at least fixed port sides.

Slot

- Before phase 3.

Dependencies

- None.

Remarks

- None.

Reversed Edge Restorer

Restores the direction of reversed edges.

Preconditions

- The graph is layered.

Postconditions

- Reversed edges are restored to their original direction.

Slot

- After phase 5.

Dependencies

- None.

Remarks

- Note that this processor doesn't have any dependencies. Let's take a long edge that was reversed during phase 1 and then split into multiple segments by the LongEdgeSplitter. All the edges generated by that processor inherit the `REVERSED` property of the original long edge. Thus, it doesn't make any difference if we reverse all those edges before joining them to the original long edge, or if we join them first and reverse the original long edge afterwards.

Self Loop Processor

Does some work that enables the other processors and phases to handle self-loops. To handle them well, the NorthSouthPortPreprocessor may be required.

Preconditions

- The graph is layered.

Postconditions

- Self-loop edges going into a western port have been reversed.
- For west-east self-loops, a dummy node has been inserted.

Slot

- After phase 3.

Dependencies

- InvertedPortProcessor

Remarks

- None.

Detailed Content

List of Figures

List of Tables

List of Algorithms