Christian-Albrechts-Universität zu Kiel

Diploma Thesis

On the Automatic Layout of Data Flow Diagrams

Miro Spönemann

2009-03-19

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Prof. Dr. Reinhard von Hanxleden
Dipl.-Inf. Hauke Fuhrmann

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,

Abstract

Data flow diagrams have been successfully applied in the area of model-based design of complex embedded systems. However, their creation and maintenance can be very time-consuming, because their layout must be managed by the user. Automatic diagram layout can significantly speed up development processes; numerous methods already exist for this task, with very differing outputs. Usually the underlying structure is represented by graphs, but in the case of data flow diagrams port constraints must be added for correct layout.

This thesis aims at extending two approaches of graph drawing, hierarchical layout and orthogonal layout, for the requirements of data flow diagrams. For this purpose the concepts and algorithms for both methods are presented, and it is shown how port constraints can be integrated into these algorithms. Experimental results show that the adapted hierarchical layout algorithm is able to yield well-readable drawings, whereas the approach used for orthogonal layout still has subproblems that need further research.

Contents

List of Figures

List of Abbreviations

API	Application Programming Interface
BFS	Breadth First Search
CSV	Comma Separated Values
DFD	Data Flow Diagram (from the area of structured software analysis)
DFS	Depth First Search
EMF	Eclipse Modeling Framework
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
GPL	GNU General Public License
ILP	Integer Linear Program
JNI	Java Native Interface
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout for the Eclipse RichClientPlatform
KIML	KIELER Infrastructure for Meta Layout
LGPL	GNU Lesser General Public License
OGDF	Open Graph Drawing Framework
UML	Unified Modeling Language
VLSI	Very Large Scale Integration
WYSIWYG	What-You-See-Is-What-You-Get
XML	eXtensible Markup Language

1 Introduction

Graphical modeling languages have evolved to very appealing and convenient instruments for the development and documentation of systems, both in hardware and in software. The advantages relative to textual formats are apparent: structure and relationships of system components are caught more easily if they are represented using two dimensions instead of only one. There are various examples for graphical modeling frameworks that have become an important part of modern development processes. *Unified Modeling Language* (UML) is a collection of standards for specification and visualization of object-oriented software. *Statecharts*, originally presented by Harel [35], are extensions of state machines used to describe system behavior, and are now available with numerous dialects and semantics. The concept of *data flow* is based on compound systems that consume and produce data, which is passed through interface ports.

The majority of models used in model-based design can be mapped to *graphs* with additional data, such as transition triggers and actions for the edges of Statecharts (see Figure 1.1), or field and operation data for the vertices of class diagrams (see Appendix A). Graphs in turn have a natural graphical representation by drawing vertices as points or shapes in the plane and edges as lines connecting them. If the position of each object in the plane is fixed, drawing a specific diagram is just a matter of creating appropriate graphical representations of these objects. Usually the objects are placed manually using a What-You-See-Is-What-You-Get (WYSIWYG) editor. The hard part is to find *good* placements automatically; this is commonly referred to as the *graph drawing* problem. To avoid creating a layout manually each time a graphical model is built, and adjusting the layout each time the model is changed, methods for automatic diagram layout are needed.

In contrast to layout of one-dimensional textual models, the problem of automatic layout of graphical models is very complex. For some types of models, such as class diagrams, this topic has been intensively addressed [17, 30], while for others, such as data flow diagrams, it seems to be largely an open problem.

This thesis aims at building a bridge between the areas of graphical modeling using data flow concepts and graph drawing by discussing possible methods of automatic diagram layout. Two very different approaches are covered: the *hierarchical* layout method reveals to be very extensible and suitable for the constraints introduced for data flow diagrams, whereas the extension of the *orthogonal* layout method is more complex, and a lot more work is required to develop and implement it. We will call the former our *main approach*, and the latter our *alternative approach.*

Most concepts of the two layout approaches are taken from previous work in the area of graph drawing. My main contributions are the formal definition of drawing

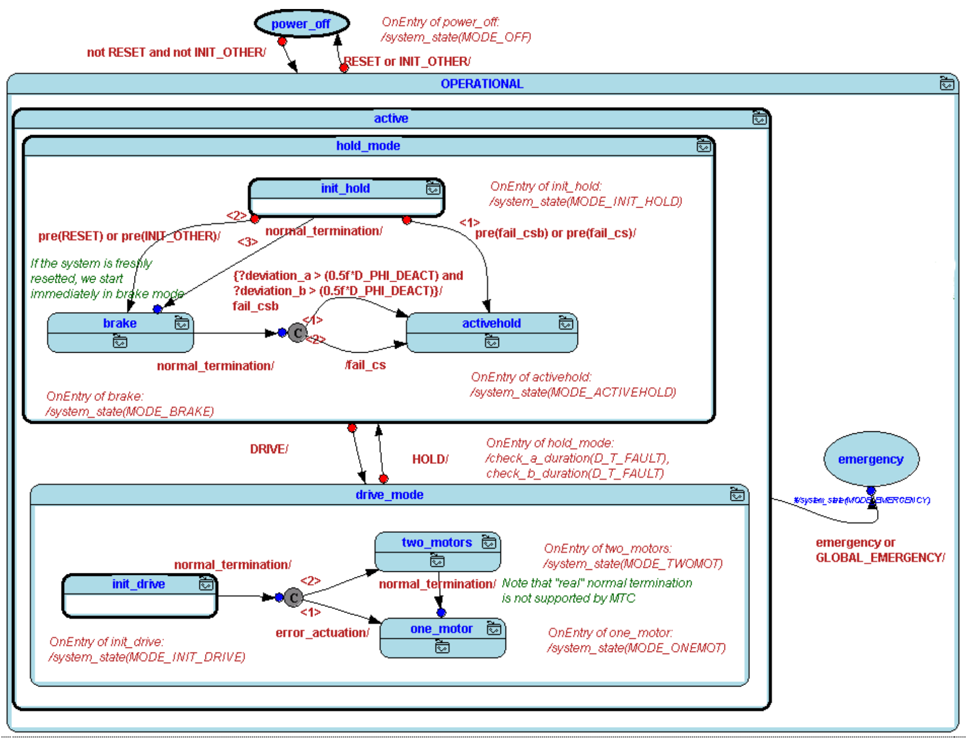


Figure 1.1: Modeling system behavior with Statecharts, here an avionics application [24]

with port constraints in Section 4.3, and specific algorithms to handle these constraints in the hierarchical layout method in Section 5.2. These algorithms include methods for crossing reduction under different scenarios, handling of hyperedges that connect multiple ports, and a new approach for displaying hierarchy of models. The drawings given in Section 7.3.1 demonstrate the usefulness of the new methods by comparing outputs of automatic layout with hand-made diagrams. Furthermore, it is shown how to apply previous work on constraints for planarization to the scenarios of port constraints in data flow diagrams.

We will proceed as follows. Chapter 2 introduces data flow diagrams, general drawing aesthetics, and special constraints imposed on drawings of data flow. Chapter 3 includes an overview of methods for graph drawing and a related topic, circuit and schematic design; moreover it covers existing approaches and solutions for drawing with port constraints. Chapter 4 provides exact definitions of graphs, drawings, and constraints. The main and alternative approaches are discussed in Chapters 5 and 6, respectively. An overview of the implementation and some results are covered in Chapter 7, and Chapter 8 will conclude and depict further work to be done.

2 Data Flow Diagrams

This chapter will illustrate the term *data flow* by introducing its basic concepts and prevalent applications. Furthermore, we will examine aesthetics criteria as well as key points that distinguish layout of data flow diagrams from the general problem of graph layout.

2.1 System Modeling with Data Flow Languages

The term *data flow diagram* as used in this work refers to graphical representations of *data flow models*. Such models are usually constructed using a *data flow language*, that is a programming language based on the exchange of data entities between *operators* [47], which can also be seen as *computations*, *functions*, *actors*, or *nodes*. Operators consist of an external interface, which defines the inputs and outputs of data, and an internal specification of behavior. The operators together with the fixed paths of data flow form a directed graph (see Section 4.1); for this reason each data flow model naturally yields a graphical representation [10], see Figure 2.1 for an example. There are different semantics which can be applied to the data flow paradigm [40, 47].

Applications of data flow diagrams can be found in modern software and hardware development tools; some of these, such as Simulink (The MathWorks, Inc.), LabVIEW (National Instruments Corporation), and ASCET (ETAS Inc.), are mainly used for model-based design and simulation of embedded systems and digital or analog hardware, while others, such as SCADE (Esterel Technologies, Inc.), are optimized for automatic code generation from high-level system models. The Ptolemy project [46] features data flow diagrams for *actor-oriented design*, where *actors* exchange

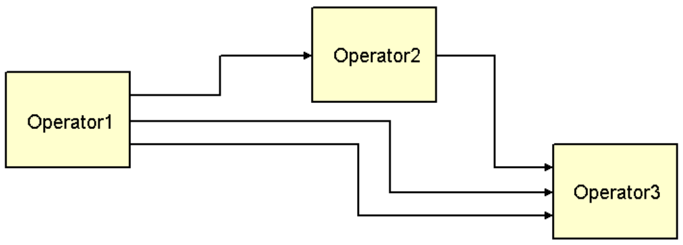


Figure 2.1: Example for the graphical representation of a data flow model: Operator1 has three outputs, Operator2 has one input and one output, and Operator3 has three inputs

data and process it under different *models of computation.* All these examples feature a graphical editor for data flow diagrams, so that users can create diagrams in WYSIWYG manner. Example diagrams are presented in Figures 2.2 and 2.3.

It is not surprising that most fields of application are related to embedded systems, because the concepts of data flow can be closely matched with those of block diagrams for hardware and distributed systems. In fact, the layout methods presented in this thesis are not restricted to diagrams for the data flow paradigm, but can be applied more generally to any diagram that is submitted to the same rules (see Section 2.3), *e.g.* drawings of hardware circuitry.

2.2 Aesthetics of Diagrams

Aside from general restrictions and drawing conventions, algorithms for automatic layout are subject to the goal of optimizing a set of *aesthetics criteria* [51, 6]. The most important to mention are the following:

Crossings Minimize the total number of crossings between edges.

Direction Maximize the number of edges pointing to a specific direction, *e.g.* to the right.

Bends Minimize the total number of bends along the edges.

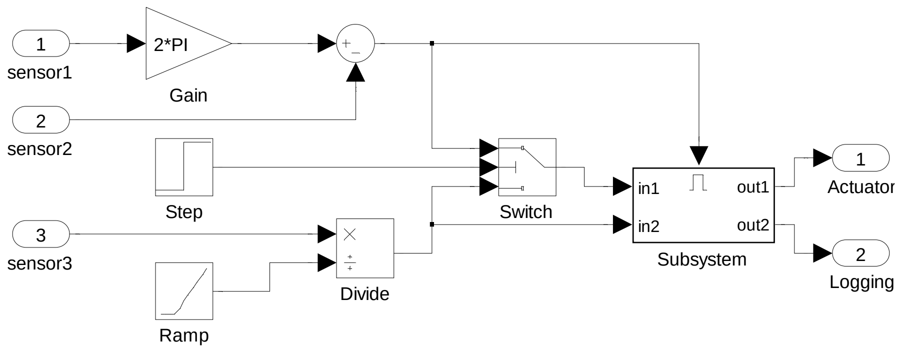
Area Minimize the total area of the drawing while preserving a minimal distance between all objects.

AspectRatio Keep the aspect ratio low, that is the width of the drawing divided by its height for landscape format drawings, and the inverse for portrait format. A good aspect ratio should be similar to that of a standard screen, which is about 1.33, or approximate the *golden ratio* of $(1 + \sqrt{5})/2 \approx 1.62$.

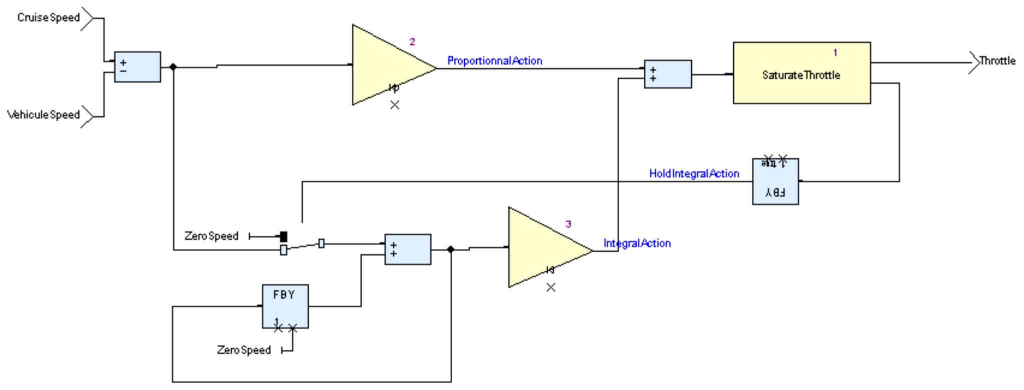
The optimization problems related with the first three criteria are NP-hard [28, 27, 29], and in some cases they can even contradict each other (see Figure 2.4). For these reasons it is impossible to optimize all aesthetics criteria at once, and all drawing algorithms are forced to use heuristics to address the criteria one by one. The order in which the criteria are treated determines a precedence among them: the criterion which is handled by the first heuristic has the highest priority. Therefore the choice of drawing algorithm has a great impact on the resulting drawing and should depend on the priority of aesthetics criteria for the specific class of diagrams, or even on the preference of each user.

2.3 Specialties of Data Flow Diagrams

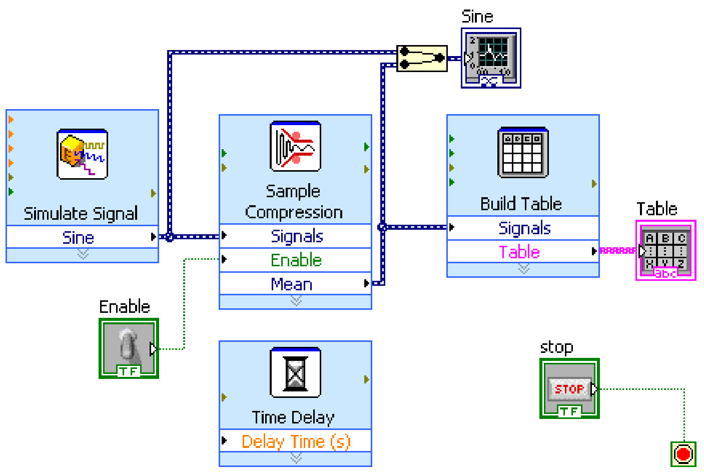
Although the actual representation of vertices and edges is different for all three diagrams shown in Figure 2.2, they have some common properties which can be expressed as drawing conventions for the corresponding graphs:



(a) Simulink



(b) SCADE



(c) LabVIEW

Figure 2.2: Example diagrams from graphical modeling tools

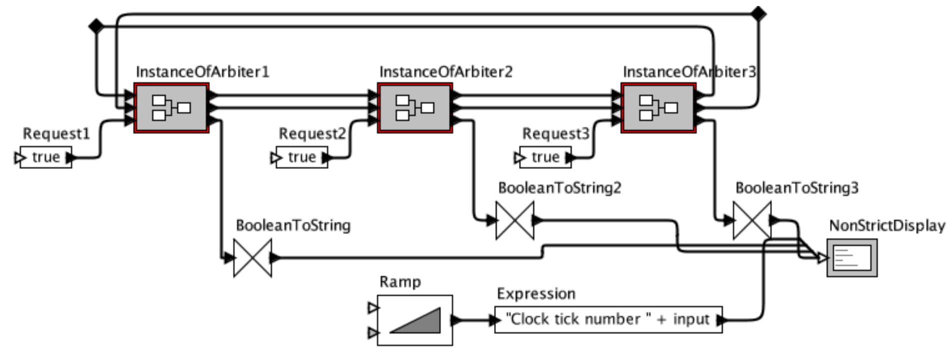


Figure 2.3: Example diagram from Ptolemy

ORTHOGONALITY Edges are drawn only with horizontal or vertical line segments, thus all angles formed by edge bends are multiples of 90°. This drawing constraint is also called *rectilinear* edge routing.

PORTS Edges do not connect to their endpoint vertices at arbitrary positions, but only at prescribed *ports*.

HYPEREDGES Some line segments of edges that are incident at the same port of a vertex may be superposed (see Figure 2.5). These superposed edges can also be seen as one *hyperedge* which connects more than two ports.

The ORTHOGONALITY and HYPEREDGES conventions do not have any semantic meaning, but rather reflect the usual way in which block diagrams are drawn. The junction points of superposed edges are often drawn as small circles to distinguish them from crossings of edges that do not belong together.

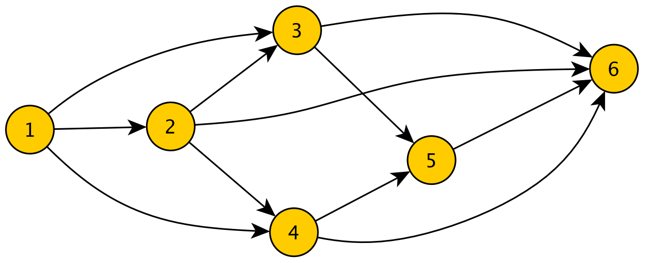
In contrast, the PORTS convention is important to keep the data flow diagrams unambiguous, for an edge does not only have to specify which operators it connects, but also which output port is responsible to feed the corresponding flow of data, and which input ports read from that flow. Four main scenarios are conceivable for the positions of ports around a vertex:

FREEPORTS All ports may be drawn at arbitrary positions on the border of their corresponding vertex.

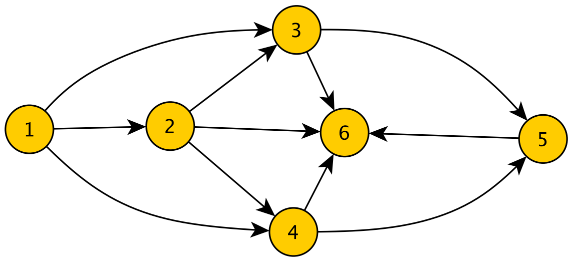
FIXEDSIDES The side of the vertex is prescribed for each port (*e.g.* the top, bottom, left, or right border for rectangle-shaped vertices), but the order of ports is free on each side.

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

FIXEDPORTS The exact position is fixed for each port.



(a) Drawing that respects direction of flow



(b) Planar drawing

Figure 2.4: Conflict between the Crossings and the Direction criteria

Mixed-case scenarios, in which some ports have fixed positions and others are free, are not covered in this thesis, because they require very complex handling and are probably not needed in most applications. Furthermore, only vertices with rectangular shape are considered to simplify matters; this can be generalized to other types of shapes by replacing them with their bounding box, *i. e.* the smallest rectangle by which they are completely covered.

2.4 Displaying Hierarchy

In most graphical modeling tools the internal behavior of each operator of a data flow diagram is hidden. This means that to display the internals of an operator user action is required, *e. g.* a double-click on the operator. If the operator itself is described by a data flow model, the result of this action is that a new window holding the nested diagram is created. The inputs and outputs of the containing vertex are then displayed as special vertices, as can be seen in the example in Figure 2.6(a).

An alternative for displaying such hierarchical structures is to draw the nested diagram directly inside the containing vertex, as in Figure 2.6(b). Let for instance A be an operator whose internal behavior is defined using nested operators $a_1, \ldots, a_n$, and let $E \subseteq \{a_1, \ldots, a_n\}$ be the subset of operators which are connected with inputs or outputs of A. By drawing an edge from each $a \in E$ to the corresponding ports of A, the relationship between the external use and the internal behavior of A becomes much more perceivable. In Figure 2.6(b), data emitted by Box1 flows into Box2,

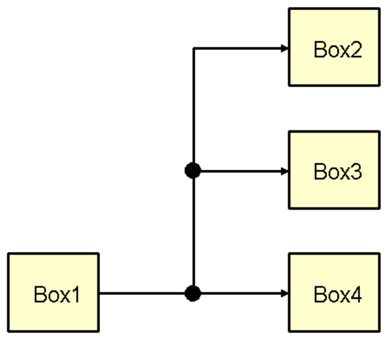


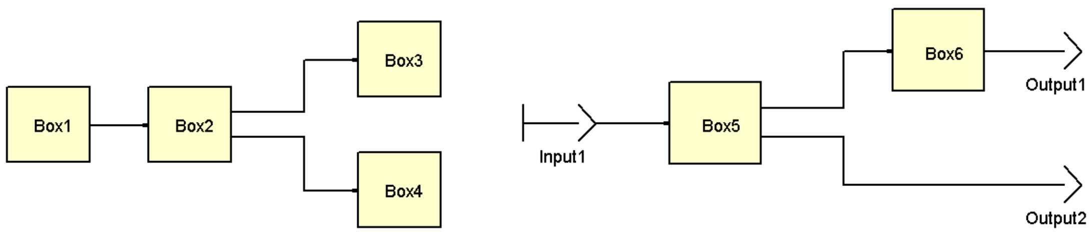
Figure 2.5: A hyperedge connecting one box with three others

where it is received by Box5. Box5 and Box6 then output their data tokens, which are sent from Box2 to Box3 and Box4.

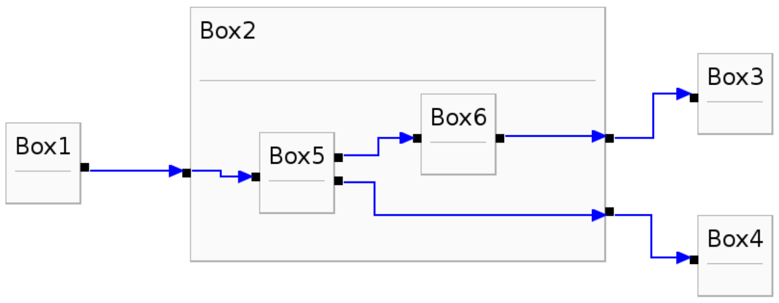
These concepts can be summarized to the following drawing requirements:

Hierarchy A vertex may contain other vertices to describe its internal behavior.

ExternalPorts The input and output ports of a vertex may be connected with its internal vertices.



(a) Hidden internal behavior (left), with further specification of Box2 (right)



(b) Directly expressed hierarchy: Box2 is expanded to reveal its internal behavior

Figure 2.6: Two ways of displaying hierarchy

3 Related Work

Besides the context of data flow languages and system modeling introduced in Chapter 2, the term *data flow diagram* and its abbreviation DFD are used in the area of structured software analysis [7]. In this sense DFDs are used for software requirements specification and modeling of the interaction between processes and data. Layout of DFDs has been covered by Tamassia *et al.* [5] and Doorley *et al.* [13]. The kind of diagrams discussed in this thesis is very different, therefore the DFDs of structured analysis will not be mentioned again, and by *data flow diagram* we will always refer to the meaning explained in Chapter 2.

This chapter will give an overview of methods that can be used for automatic layout and identify previous work about layout with port constraints in literature and in software.

3.1 Methods of Graph Drawing

There are several approaches to the general problem of graph drawing [6, 42, 39, 59, 11], of which a selection is presented in this section.

Layered approach: The *layered* or *hierarchical* layout method is for directed graphs and emphasizes DIRECTION [56]. It first eliminates cycles in the graph, then determines a layering of vertices and optimizes this layering with respect to vertex positions. The hierarchical method was selected as the main approach for this thesis and is therefore covered with more detail in Chapter 5.

Force-directed approach: This approach creates a model of physical forces and minimizes the energy of the model [14]. One variant consists in assigning springs with appropriate forces to each pair of vertices; such methods are called *spring embedders*. This very natural approach is widely used and especially suited to display symmetries of the graph. Edges are mostly drawn straight-line; see Figure 3.1 for an example.

As planarity of graphs is a topic which is well studied in graph theory, many drawing methods expect a planar embedding as input. If the graph to be drawn is not planar, it is first processed in a *planarization* phase (see Sections 4.2 and 6.1). By doing so, the highest priority is put on the CROSSINGS criterion. Methods which build on planarization are the following.

Topology-shape-metrics approach: This method puts second highest priority on BENDS by computing a bend-minimal orthogonal drawing, so the ORTHOGONALITY convention is always satisfied; more details are found in Chapter 6.

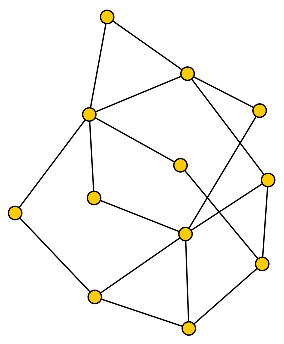


Figure 3.1: A drawing produced by a force directed method

There are more approaches to orthogonal graph drawing, of which an overview is given by Eiglsperger, Fekete, and Klau [42].

Visibility approach: A *visibility representation* is constructed, which maps vertices and edges to horizontal and vertical segments; these are in turn replaced by drawings of their corresponding elements [60, 6], see Figure 3.2. There are different variants for the actual shape of vertices and edges, some allowing rectilinear edge style. Visibility representations may be seen as a form of layering, which means that the DIRECTION criterion can also be taken into account.

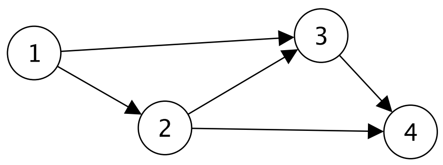
Augmentation approach: The graph is augmented by vertices, edges, or both, to get a graph with specific properties, *e.g.* one in which all faces have exactly three edges [54, 23], or a biconnected graph [8]. In their basic variants, these algorithms usually yield a straight-line drawing (see Figure 3.3).

Mixed model approach: This approach extends methods of straight-line drawing from the augmentation approach to construct orthogonal or quasi-orthogonal drawings [41, 32] (see Figure 3.4).

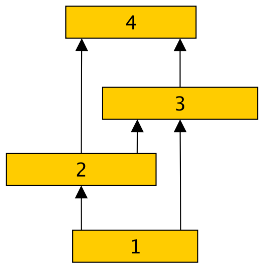
The main specialty that makes layout of data flow diagrams more difficult than layout of general graphs is that edges are always connected to *ports* of a vertex (see Section 2.3). Up to now, port constraints have received little attention in the area of graph drawing:

- Gansner *et al.* proposed a method to add port constraints to hierarchical layout as a *displacement* in one dimension [25], which means that ports may only be attached to the vertex side that lies towards the next layer (for outgoing edges) or the previous layer (for incoming edges). Port constraints with the FIXEDPORTS scenario are not possible with this approach.

- Eiglsperger *et al.* include port constraints in their general approach for constraints in the orthogonalization phase of orthogonal layout [18]. However, it is sufficient to handle side constraints in the orthogonalization phase (see Section 6.2.2).

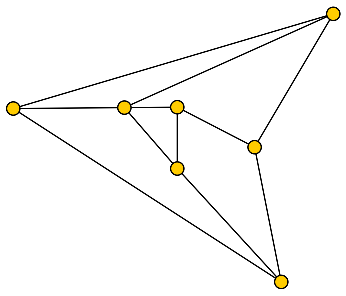


(a) An acyclic directed graph

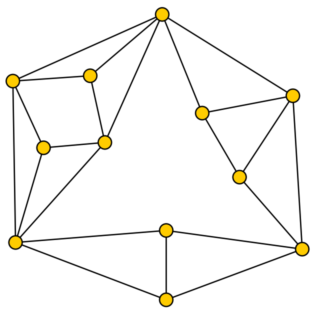


(b) Visibility drawing of the graph in (a)

Figure 3.2: Example for a visibility drawing deduced from a directed graph: two vertices are *visible* if they can be connected with a straight vertical line without crossing other objects.



(a) Drawing constructed by triangulation



(b) Drawing constructed by analyzing connectivity of the graph

Figure 3.3: Examples of techniques following the augmentation approach

- Gutwenger *et al.* introduced a general structure for port constraints in planarization [31]; this very useful approach facilitates realization of different scenarios for port constraints and is presented in Section 6.1.3.

An overview of graph drawing methods that support constraints is given by Tamassia [58], but ports are not mentioned in that paper.

3.2 Circuit and Schematic Design

Some aspects of the graph drawing problem are related to similar problems in design of integrated circuits or hardware schematics. It therefore seems natural to look into these areas, especially for the layout of data flow diagrams, which are very similar to schematics.

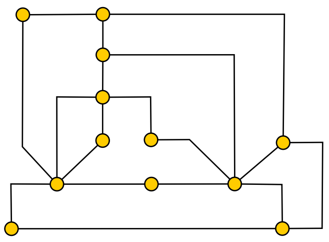


Figure 3.4: Quasi-orthogonal drawing constructed with a mixed-model approach

The general approach of circuit and Very Large Scale Integration (VLSI) design is the *place and route* method: first the modules of the circuit are placed, then wires are routed to connect the modules electrically [3, 61]. There are numerous different placement methods, of which some are analytical techniques, while others are based on physical laws or biological phenomena [55]. Routing is usually done by partitioning the available area into rectangular *channels*, and each channel is assigned a set of wires which may use the corresponding area. Wires may only cross if they lie on different *layers* to avoid unwanted electrical contact. The number of available layers may vary depending on the type of circuit and the application. If there is only one layer, no wire crossings are allowed at all. The *Manhattan* routing model uses two layers, one to route channels horizontally and the other to route channels vertically. An overview of routing methods is given by Hu and Sapatnekar [37].

While methods of graph drawing aim at readability and clarity for human users, the problem of circuit design is a technical one, and its methods are therefore optimized to meet the requirements imposed by the respective technical environment. Prevalent optimization goals are minimization of the number of routing layers, minimization of the area and minimization of wire lengths. As the time needed to find such a layout is not a critical aspect, computations may take hours or even days, which is unacceptable in a user interface environment, where the user should not be forced to wait for a result for more than few seconds. For these reasons it seems unlikely that methods from circuit design can practically be applied to the graph drawing problem.

A more promising area is that of automatic generation of schematic diagrams [1, 45], which are used for modeling and documentation of hardware systems. Here the focus is on human readability, so the aesthetics criteria are equivalent to those used in graph drawing. In terms of schematic design, the basic structure which is layouted consists of *modules* and *connections*. The layout problem is decomposed into four steps:

1. Logical module placement
2. Logical connection routing
3. Geometrical module placement

4. Geometrical connection routing

The place and route method used here follows the basic concepts of circuit design. During the logical phase modules are placed into a grid, and connections are wired through channels. The geometrical phase determines exact coordinates from these logical positions.

The two methods that were chosen for further examination, hierarchical layout and orthogonal layout, are both from the area of graph drawing, which grants them the advantage of a large research community with solid theoretical backgrounds. The methods of hardware design outlined in this section are a lot more specific to certain applications, hence their application to graphs would probably require more work to be done. While graph drawing methods must be *specialized* for the requirements of data flow diagrams, hardware design methods must be *generalized* to lift them from their original field of application.

3.3 Graph Drawing Libraries

There are many tools and software libraries that can be used for graph drawing [39]. This section will give a brief overview of existing solutions.

aiSee (AbsInt Angewandte Informatik GmbH) is a commercial graph layout tool which is optimized for huge graphs. Input is expected in a special textual format called GDL.

CCVisu (Dr. Dirk Beyer, Simon Fraser University) is a free graph layout tool for force-directed layout. Input is expected in the textual format RSF. The tool is available under the GNU Lesser General Public License (LGPL).

GoDiagram (Northwoods Software Corporation) is a commercial library for creation, visualization, and layout of diagrams. Supported platforms are Java, .NET, and MFC.

GoVisual (oreas GmbH) is a commercial graph layout library featuring various layout algorithms. Supported platforms are Java (via JNI), C++, .NET, and COM.

GDToolkit (Graph Drawing group of the Univesità Roma Tre, Graph Drawing and Visualization group of the Università di Perugia) is a partly free C++ library for handling and layout of graphs, evolved from long-term research projects. The academic license is limited to 6 months.

Graphviz (`http://www.graphviz.org/`) is an open source graph layout tool [26, 25] which is widely used in different applications. Input is expected in the textual format DOT.

ILOG JViews Diagrammer (ILOG, an IBM company) is a commercial Java library for creation, visualization, and layout of diagrams.

JGraph Layout Pro (JGraph Ltd.) is a commercial Java library for graph layout.

OGDF (Algorithm Engineering group of the Dortmund University of Technology, chair of Prof. Dr. M. Jünger of the University of Cologne, oreas GmbH) is a free C++ graph drawing library used for academic research which contains very sophisticated algorithms.

PIGALE (H. de Fraysseix, P. Ossona de Mendez) is a free C++ graph layout library intended for research on planar graphs and is available under the GNU General Public License (GPL).

Tom Sawyer Layout (Tom Sawyer Software) is a commercial graph layout library. Supported platforms are Java, ActiveX, ASP.NET, JSP, and MFC.

uDraw(Graph) (Chair of Prof. Dr. B. Krieg-Brückner, University of Bremen) is a free graph layout library.

yFiles (yWorks) is a commercial library for layout and visualization of graphs. Supported platforms are Java, .NET, AJAX, and FLEX.

Zest (Eclipse GEF) is a sub-project of the Eclipse Graphical Editing Framework (GEF) for layout and visualization of graphs.

The only libraries whose authors clearly state to support port constraints are the commercial products ILOG JViews Diagrammer, Tom Sawyer Layout, and yFiles. The free tool Graphviz has only limited support of ports, as the only ways to declare ports as attachment point of edges are to either give a side of the corresponding vertex, *e.g. north* or *north-west*, or to create a structured vertex or label. In the latter case the vertex or its label must be assigned an internal structure, so that edges can be directly connected with particular elements of the internal structure. This is used for layout of data structures, which may contain *pointers* that reference other objects.

Unfortunately, none of the open libraries has direct support of port constraints. This affirms the assumption that the topic of graph drawing with port constraints has received comparatively little attention in research so far.

4 Definitions

In this chapter we will introduce the mathematical notation which is needed to handle graphs and their drawings and which is used throughout this thesis. Most notation is taken from Di Battista *et al.* [6], with some adaptions and additions. Undirected edges are written as $\{u, v\}$ to distinguish them from directed edges (u, v). A functional notation is used to declare objects related with an element of the graph, *e.g.* $v_t(e)$ for the target vertex of e, or $E_o(v)$ for the outgoing edges of v.

Based on the essential definitions, we formally introduce port constraints to identify the central problem of data flow layout.

4.1 Graphs

An *undirected graph* is a pair $G = (V, E)$, where V is a finite set and E is a multiset of multisets $e \subseteq V$ with $|e| = 2$. The elements of V are called *vertices* or *nodes*, and the elements of E are called *edges* or *connections*. An edge $e \in E$ with $e = \{v, v\}$ is called a *self-loop*. An edge whose multiplicity in E is greater than one is called a *multiple edge*. A *simple undirected graph* is an undirected graph with no self-loops and no multiple edges, while a general graph is also called *multigraph*. The elements of an edge $e = \{u, v\}$ are called its *endpoints*. If there exists an edge $e = \{u, v\} \in E$, we call u and v *adjacent* to each other and e *incident* to u and v. The *neighbors* of a vertex v are its adjacent vertices. The *degree* of v is the number of edges which are incident to v. A *subgraph* of $G = (V, E)$ is a graph $G' = (V', E')$ for which $V' \subseteq V$ and $E' \subseteq \{\{u, v\} \in E : u, v \in V'\}$. For $S \subset V$ we define $G \setminus S := (V \setminus S, E \setminus \{\{u, v\} : u \in S \vee v \in S\})$, and for $v \in V$ we write $G \setminus v := G \setminus \{v\}$.

A *directed graph* $G = (V, E)$ consists of a finite set of vertices V and a multiset $E \subseteq V \times V$. An edge $e = (u, v) \in E$ is an *outgoing edge* of u and an *incoming edge* of v. $v_s(e) := u$ is called the *source* of e, and $v_t(e) := v$ is called the *target* of e. The *indegree* of a vertex v is the number $|E_i(v)|$ of its incoming edges $E_i(v)$, and its *outdegree* is the number $|E_o(v)|$ of outgoing edges $E_o(v)$. A vertex with no outgoing edges is called a *sink* of the graph, and a vertex with no incoming edges is called a *source* of the graph. The terms and properties defined above for undirected graphs apply analogously for directed graphs.

A *path* of a graph is a sequence $(v_1, \ldots, v_k)$ of vertices such that $\{v_i, v_{i+1}\} \in E$ (respectively $(v_i, v_{i+1}) \in E$ for a *directed path*) for $i \in \{1, \ldots, k - 1\}$. A path $p = (v_1, \ldots, v_k)$ is called *simple* if $v_i \neq v_j$ for all $i \neq j$. p is a *cycle* if $v_1 = v_k$. A cycle $(v_1, \ldots, v_k)$ is called *simple* if $(v_1, \ldots, v_{k-1})$ is a simple path. A graph G is *acyclic* if it contains no cycles. It is *connected* if for each pair (u, v) of vertices there is a path between u and v in G. The *connected components* of G are the maximal connected subgraphs of G.

The notion of connectivity can be extended to stronger versions: A vertex v in a connected graph G is a *cutvertex* if $G \setminus v$ is not connected. G is called *biconnected* if it contains no cutvertices. A pair (u, v) of vertices in a biconnected graph G' is a *separation pair* if $G' \setminus \{u, v\}$ is not connected. G' is called *triconnected* if it contains no separation pairs. The maximal subgraphs of an arbitrary graph which are biconnected or triconnected are respectively called *biconnected* or *triconnected* components.

A *topological numbering* of a directed graph $G = (V, E)$ is a map $t : V \to \mathbb{N}$ such that $t(u) < t(v)$ for all edges $e = (u, v)$. If t is injective, we call it a *topological sort.*

A connected and acyclic graph is called a *tree.* A *rooted tree* is a tree $T = (V, E)$ with a specific *root* $r \in V$. All vertices which only have one neighbor are called *leaves* of T, except the root; the remaining vertices are the *inner vertices* of T. The unique path $(v_1, \ldots, v_k)$ from the root $r = v_1$ to any leaf v_k determines a parent-child relationship between each pair of vertices (v_i, v_{i+1}), $i < k$. An *ordered tree* is a rooted tree with a given order of children for each parent vertex.

4.2 Drawings and Planarity

A set $M \subset \mathbb{R}^2$ is *convex* if $\{x + t(y - x) : t \in [0, 1]\} \subseteq M$ for each two points $x, y \in M$. M is *bounded* if there are $(l_1, l_2), (u_1, u_2) \in \mathbb{R}^2$ such that $l_1 \leq x_1 \leq u_1$ and $l_2 \leq x_2 \leq u_2$ for all $(x_1, x_2) \in M$. Here we define a *curve* in $\mathbb{R}^2$ as the image of a continuous mapping $\gamma : [0, 1] \to \mathbb{R}^2$. The *power set* $\mathcal{P}(M)$ of a set M is the set of all subsets of M.

A *drawing* of a graph $G = (V, E)$ is a function $\Gamma : V \cup E \to \mathcal{P}(\mathbb{R}^2)$, where for each $v \in V$ the set $\Gamma(v) \neq \emptyset$ is convex, bounded, and closed, and for each $e = \{u, v\} \in E$ the set $\Gamma(e)$ is a curve in $\mathbb{R}^2$ such that

$$|\Gamma(e) \cap \Gamma(u)| = |\Gamma(e) \cap \Gamma(v)| = \begin{cases} 2 & \text{if } u = v \\ 1 & \text{otherwise} \end{cases} .$$

For a proper drawing the following constraints are usually added to Γ.

DisjointNodes: $\Gamma(u) \cap \Gamma(v) = \emptyset$ for all $u \neq v \in V$.

DisjointEdges: $\Gamma(e_1) \cap \Gamma(e_2)$ is finite for all $e_1 \neq e_2 \in E$.

UnambiguousEdges: $\Gamma(e) \cap \Gamma(u) = \emptyset$ for all $e = \{v_1, v_2\} \in E$ and $u \in V$ with $u \notin e$.

Let $v \in V$ and $X_v := \{x' : (x', y') \in \Gamma(v)\}$, $Y_v := \{y' : (x', y') \in \Gamma(v)\}$. The *position* of v is the point $\text{pos}(v) := (\min X_v, \min Y_v)$, and its *size* is the vector $\text{size}(v) := (\max X_v, \max Y_v) - \text{pos}(v)$. In the following the terms *vertex* v and *drawing* of the vertex $\Gamma(v)$ are used interchangeably, as well as the terms *edge* and *drawing* of the edge.

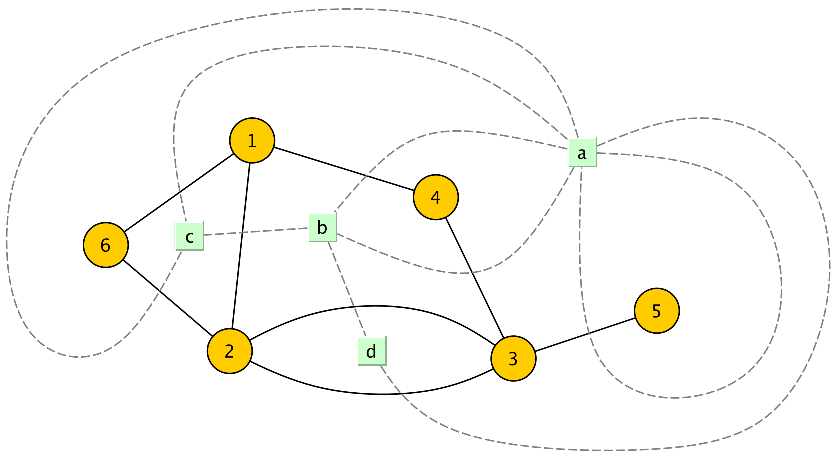


Figure 4.1: A planar drawing of a graph (circular vertices) and its dual graph (square vertices)

A drawing Γ is called *polyline*, if for each edge e its image $\Gamma(e)$ can be decomposed into a sequence of straight lines $l_1, \ldots, l_h$ for a minimal h, such that $|l_i \cap l_{i+1}| = 1$ for all $i \in \{1, \ldots, h-1\}$. Then the points $\{b_i\} := l_i \cap l_{i+1}$ are the *bend points* or *bends* of e. A polyline drawing is *straight-line* if $h = 1$ for all edges, and it is *orthogonal*, or *rectilinear*, if all line segments are aligned horizontally or vertically (see the ORTHOGONALITY drawing convention in Section 2.3). A *quasi-orthogonal* drawing is an orthogonal drawing for which the first and the last line segment of each edge may be aligned non-orthogonally.

An *edge crossing* is a point $x \in \mathbb{R}^2$ such that there exist distinct edges $e_1 = (u_1, v_1), e_2 = (u_2, v_2) \in E$ satisfying $x \in \Gamma(e_1) \cap \Gamma(e_2)$ and $x \notin \Gamma(u_1) \cup \Gamma(v_1) \cup \Gamma(u_2) \cup \Gamma(v_2)$. A drawing of a graph is *planar* if there are no edge crossings. A graph is called *planar* if it admits a planar drawing. A planar drawing partitions the plane $\mathbb{R}^2$ into topologically connected regions called *faces*. There is always exactly one unbounded face, which is called the *external face*. Furthermore, a drawing determines a circular order on the incident edges of each vertex according to their clockwise sequence; this is called an *embedding* of the graph. An *embedded graph* is a graph with a specific embedding. The *dual graph* G^* of an embedded planar graph G consists of a vertex f^* for each face f in G, and an edge $\{f^*, g^*\}$ for each edge in G which is shared by the two faces f and g. An example is shown in Figure 4.1. Each planar embedding of a graph is uniquely determined by the corresponding dual graph if we additionally define the external face.

A drawing of a directed graph is called *upward* if for each edge e the set $\Gamma(e)$ is a curve which monotonically increases in the y-direction, hence all edges are upward oriented. A graph that admits a drawing which is both planar and upward is called *upward planar* [38]. An example for a planar graph which is not upward planar is shown in Figure 2.4.

4.3 Port Constraints

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

A drawing of a port based graph is a function $\Gamma : V \cup E \cup P \rightarrow \mathcal{P}(\mathbb{R}^2)$ such that Γ restricted on $V \cup E$ yields a proper drawing of the graph G, and

(a) $|\Gamma(p)| = 1$ for all $p \in P$,

(b) $\Gamma(p)$ is on the boundary of $\Gamma(v)$ for all $p \in P(v)$,

(c) $\mathrm{pos}(p_1) \neq \mathrm{pos}(\mathrm{p}_2)$ for ports $p_1 \neq p_2$, and

(d) $\Gamma(p_s(e)), \Gamma(p_t(e)) \subset \Gamma(e)$ for all $e \in E$.

We write $\mathrm{pos}(p) := x$ for all ports p with $\Gamma(p) = \{x\}$. For the HYPEREDGES convention, one of the drawing constraints must be adjusted.

DISJOINTEDGES: $\Gamma(e_1) \cap \Gamma(e_2)$ is finite for all $e_1, e_2 \in E$ for which $\{p_s(e_1), p_t(e_1)\} \cap \{p_s(e_2), p_t(e_2)\} = \emptyset$.

Let $v \in V$ and the midpoint of its drawing be given as $m := \mathrm{pos}(v) + \frac{1}{2}\mathrm{size}(v)$. For each port $p \in P(v)$ we define the *port angle* $\alpha_v(p)$ as the positive angle between the vectors $\mathrm{pos}(p) - m$ and $\mathrm{pos}(v) - m$ (see Figure 4.2). Let the ports of v be ordered as $P(v) = \{p_1, \ldots, p_k\}$. We say that Γ *respects* the port order of v if $\alpha_v(p_i) < \alpha_v(p_j)$ for all $i < j \leq k$. We write $\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}$ for the top, right, bottom, and left side of a vertex, respectively, and add an order N < E < S < W. The side of a port $p \in P(v)$ is defined as

$$S_v(p) = \begin{cases} \mathrm{N} & \text{if } 0^\circ \leq \alpha_v(p) < 90^\circ \\ \mathrm{E} & \text{if } 90^\circ \leq \alpha_v(p) < 180^\circ \\ \mathrm{S} & \text{if } 180^\circ \leq \alpha_v(p) < 270^\circ \\ \mathrm{W} & \text{if } 270^\circ \leq \alpha_v(p) < 360^\circ \end{cases} .$$

Port positions are restricted depending on the chosen scenario.

FREEPORTS: No restrictions are imposed on port positions.

FIXEDSIDES: Each port $p \in P(v)$ has a prescribed side $S_v(p)$.

FIXEDPORTORDER: Γ must respect a prescribed port order of v, and the side of each port is fixed.

FIXEDPORTS: The exact position $\mathrm{pos}(p)$ is prescribed for each port $p \in P(v)$. This is only meaningful if the shape of $\Gamma(v)$ is fixed. If the area of $\Gamma(v)$ is not zero, these port positions induce an ordering on $P(v)$.

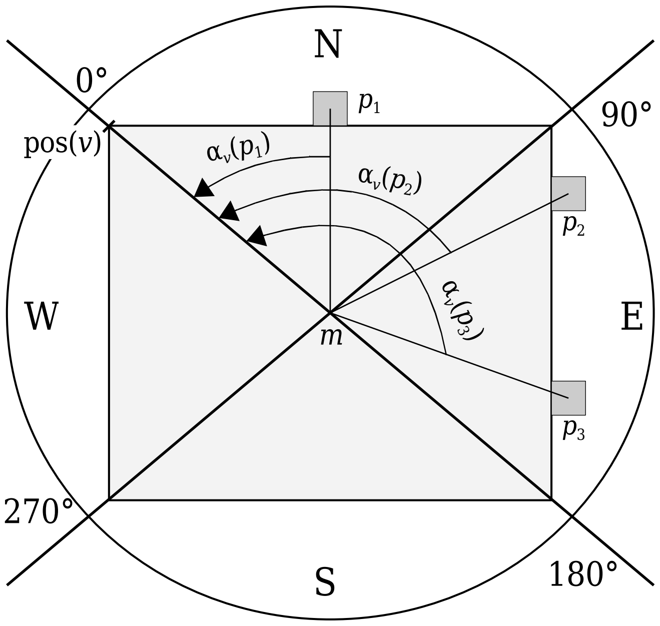


Figure 4.2: Ports $p_1, p_2, p_3 \in P(v)$ with their corresponding angles $\alpha_v(p_1)$, $\alpha_v(p_2)$, and $\alpha_v(p_3)$

5 Main Approach: Hierarchical Layout

The hierarchical layout method works only for directed graphs and aims at emphasizing the direction of flow, thus expressing the hierarchy of vertices in the graph. It was proposed by Sugiyama, Tagawa and Toda [56] and is often called *Sugiyama* layout. The term *hierarchy* here means a precedence among the vertices of the graph as induced by the directed edges. This is completely different from the notion of hierarchy introduced in Section 2.4.

5.1 Main Phases

If the input is an arbitrary directed graph, the main phases of the algorithm are the following.

1. **Cycle removal:** Break directed cycles by reversing some edges, while keeping the number of reversed edges as low as possible. In the final drawing the reversed edges are restored again, so that they point against the predominant direction of flow.

2. **Layer assignment:** Create a minimal set of *layers* $L_1, \ldots, L_k$ and assign a layer to each vertex such that for all edges (u, v) the assigned layers L_i of u and L_j of v satisfy $i < j$. This is possible because after the first phase the graph is acyclic. Another way of expressing this problem is that of finding a topological numbering t for which $\max t(V)$ is minimal.

3. **Crossing reduction:** Find an ordering of the vertices of each layer that minimizes the number of edge crossings.

4. **Node placement:** Determine exact positions of all vertices inside their corresponding layers. The vertices must not overlap each other, the ordering from the previous phase must be respected and the position of each vertex must be well-balanced with respect to its neighbors. We will call this *crosswise* placement.

5. **Edge routing:** Determine bend points for each edge and the exact distance between subsequent layers, which we will call *lengthwise* placement.

Now we will look more closely at some basic methods and heuristics which can be used for each phase. There are many alternative algorithms which are covered in the literature [6, 42, 15, 25, 20], but their complete discussion would exceed the scope of this thesis.

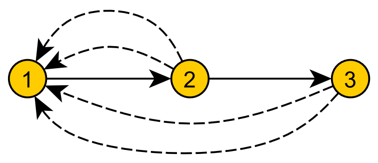


Figure 5.1: Starting a DFS on vertex 1 will reverse the four dashed edges, while starting on vertex 2 will reverse only one.

5.1.1 Cycle Removal

The goal of this phase is to find a minimal set of edges for which the graph obtained by reversing these edges is acyclic. This problem is equivalent to the *feedback arc set problem*, which is NP-complete [27]. A very simple and fast heuristic is to perform a Depth First Search (DFS) and reverse all back edges, but the results highly depend on the choice of starting vertex and can be very bad in some cases (see Figure 5.1). A better heuristic is Algorithm 5.1, which is a variant of *Greedy-Cycle-Removal* from Di Battista *et al.* [6]. The algorithm determines an ordering $v_1, \ldots, v_n$ of the vertices in G. By reversing all edges (v_i, v_j) for which $i > j$, all cycles are eliminated [6]. The worst case asymptotical running time is $\mathcal{O}(|V|^2)$ because of the implicit loop in line 26. However, in most graphs that loop is executed less often, because it only applies when a cycle is found.

Algorithm 5.1: GreedyCycleRemoval

```
1  procedure removeCycles($G$: directed graph)
2      foreach node $v$ in $G$ do
3          indeg($v$) := indegree of $v$
4          outdeg($v$) := outdegree of $v$
5          rank($v$) := 0
6      $L_i$ := list of sinks in $G$
7      $L_o$ := list of remaining sources in $G$      // Sources which are also sinks are already
8                                                    // processed in line 6
9      $r$ := $-1$                                   // Next index for the right group
10     $l$ := 1                                      // Next index for the left group

12     // All nodes are put either to the left group or to the right group, while keeping
13     // the number of edges going from right to left low.
14     while $\exists\, v \in V$ : rank($v$)=0 do
15         while $L_i$ is not empty do
16             remove a node $v_s$ from $L_i$
17             rank($v_s$) := $r$
18             $r$ := $r - 1$
19             updateNeighbors($v_s$)
20         while $L_o$ is not empty do
21             remove a node $v_s$ from $L_o$
22             rank($v_s$) := $l$
```

```
            $l := l + 1$
            updateNeighbors($v_s$)
        if $\exists\ v \in V$ : rank($v$)=0 then
            find $v$ for which rank($v$)=0 and outdeg($v$) $-$ indeg($v$) is maximal
            rank($v$) := $l$
            $l := l + 1$
            updateNeighbors($v$)

    foreach node $v$, rank($v$) $< 0$, do
        rank($v$) := rank($v$) + $|V| + 1$

    reverse all edges $e = (u, v)$ for which rank($u$) $>$ rank($v$)
end

// Update indeg and outdeg values and the lists of sources and sinks as if v was removed
procedure updateNeighbors($v$: node)
    foreach edge $e = (v, u)$, rank($u$) $= 0$, do
        indeg($u$) := indeg($u$) $-$ 1
        if indeg($u$) $= 0$ and outdeg($u$) $\neq 0$ then add $u$ to $L_o$
    foreach edge $e = (u, v)$, rank($u$) $= 0$, do
        outdeg($u$) := outdeg($u$) $-$ 1
        if outdeg($u$) $= 0$ and indeg($u$) $\neq 0$ then add $u$ to $L_i$
end
```

5.1.2 Layer Assignment

In this step we want to find layers $L_1, \ldots, L_k$ for the vertices of the acyclic graph G. A layering is called *proper* if all edges e connect only vertices from subsequent layers. A proper layering is constructed from a general layering by splitting *long edges*: given an edge $e = (v_i, v_j)$, $v_i \in L_i$, $v_j \in L_j$, for which $j - i > 1$, we add new dummy vertices $v_{i+1}, \ldots, v_{j-1}$ to the layers $L_{i+1}, \ldots, L_{j-1}$ and split e into a series of edges $e_i, \ldots, e_{j-1}$ such that $e_h = (v_h, v_{h+1})$ for all $h \in \{i, \ldots, j-1\}$ (see Figure 5.2).

The *height* of a layering $L_1, \ldots, L_k$ is k. There are other optimization goals for layer assignment besides minimizing height:

- Minimize the *width* of the layering, that is the number of vertices in a layer L for which $|L| = \max\{|L_i| : 1 \leq i \leq k\}$. Unfortunately, the problem of finding a layering with minimal width and height is NP-complete [15].
- Minimize the number of dummy vertices introduced by long edges.

A simple and linear running time heuristic consists in determining the longest path to a sink, see Algorithm 5.2. In this very common algorithm all sinks s are put into the last layer ($h(s) = 1$), and all other vertices are assigned a height $h(v)$ equal to the number of edges on a longest path to a sink plus one. This guarantees that the height of the resulting layering is minimal, but the width and the number of dummy vertices can become unnecessarily high, as seen in Figure 5.3(a).

Algorithm 5.2: LongestPathLayering

```
procedure layering($G$: directed graph)
    foreach $v$ in $G$ do
        visit($v$)
    $k := \max\{h(v) : v \in V\}$
    put each $v$ into layer $L_{k-h(v)+1}$
end

procedure visit($v$: node)
    if $h(v) \neq \perp$ then
        $h_m := 1$
        foreach edge $e = (v, u)$, $u \neq v$, do      // Self−loops are ignored during
            visit($u$)                                  // layer assignment
            $h_m := \max\{h_m, h(u)\}$
        $h(v) := h_m$
end
```

To improve this I propose moving some vertices to preceding layers, as in Algorithm 5.3. The balanced layering algorithm is greedy in that it decides locally for each vertex whether moving it to a preceding layer could improve the layering, thus greedily computing a local optimum. The algorithm runs in time $\mathcal{O}(|V|+|E|+|D|)$, where D is the set of dummy vertices implied by the output of the longest path layering. An example output of balanced layering is presented in Figure 5.3(b).

5.1.3 Crossing Reduction

The problem of crossing reduction for layered graphs, which consists in setting an order of vertices for each layer, is NP-complete, even if there are only two layers [28]. Nevertheless it is easier to find heuristics to set the order of vertices for two layers than to optimize the whole graph at once. For this reason this phase is usually solved with a *layer-by-layer sweep*: choose an arbitrary order for layer L_1, then for

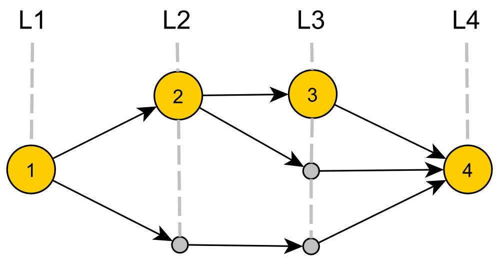
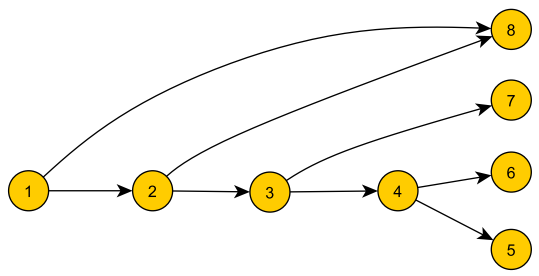
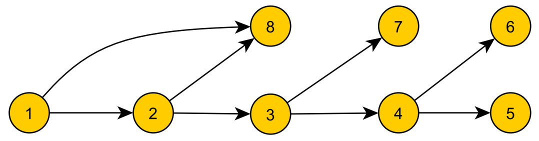


Figure 5.2: A layered graph with two dummy vertices for the long edge (1,4) and one for the edge (2,4)



(a) A longest path layering of a directed graph



(b) A balanced layering of the graph

Figure 5.3: Example for which balanced layering is clearly better than longest path layering

Algorithm 5.3: BalancedLayering

```
procedure balanceLayering($G$: directed graph)
    determine layers $L_1, \ldots, L_k$ for $G$ using longest path layering
    foreach layer $L_j$, $j \geq 3$, do
        foreach $v \in L_j$, indegree of $v \geq$ outdegree of $v$, do
            $r := \max\{i : (u, v) \in E, u \in L_i\} + 1$
            foreach layer $L_i$, $r \leq i < j$ with increasing $i$, until a fitting layer is found, do
                if $|L_i| \leq |L_j|$ then
                    move $v$ to the fitting layer $L_i$
end
```

each $i \in \{1, \ldots, k-1\}$ optimize the order for layer L_{i+1} while keeping the vertices of layer L_i fixed. Afterwards the same proceeding is applied backwards, and it can then be repeated for a specified number of iterations.

In the resulting *two-layer crossing problem* we have a fixed layer L_1 and a variable layer L_2. For a given set of vertices $\{v_1, \ldots, v_n\}$ we define the unique *rank* of each vertex v_j to be j. The goal is to assign ranks $r_2(v) \in \{1, \ldots, |L_2|\}$ to each $v \in L_2$ depending on the ranks $r_1(v') \in \{1, \ldots, |L_1|\}$ of the vertices $v' \in L_1$. These ranks determine the order of vertices in each layer. The most commonly used methods, *barycenter* and *median*, achieve ranking by first calculating values $a(v) \in \mathbb{R}$ for each $v \in L_2$ and then sorting the vertices in L_2 according to these values. The following functions are used:

Barycenter:

$$a_b(v) := \frac{1}{|E_i(v)|} \sum_{(u,v) \in E_i(v)} r_1(u)$$

Median:

$$a_m(v) := r_1(\bar{u})$$

for $\{(u_1, v), \ldots, (u_h, v)\} = E_i(v)$, $r_1(u_i) \leq r_1(u_{i+1})$ for all $i \in \{1, \ldots, h-1\}$, $\bar{u} := u_{\lfloor h/2 \rfloor}$

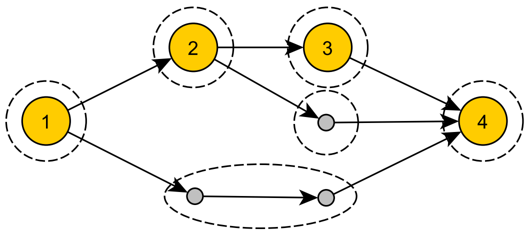
For isolated vertices v_s we set $a_b(v_s) = a_m(v_s) = 0$.

5.1.4 Node Placement

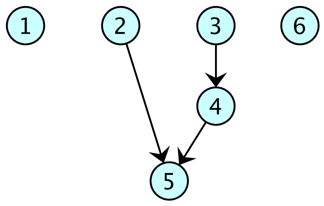
From this step on it is relevant whether *horizontal* or *vertical* overall layout direction is used: horizontal layout places the layers from left to right, while vertical layout places them top down. Only horizontal layout is discussed here, but of course the concepts for vertical layout are analogous. We assume functions r_i for each layer L_i which map each vertex to its rank as output from the crossings reduction phase.

For crosswise vertex placement in horizontal layout the vertices of each layer are arranged vertically. Sander proposes a two-phase method [52]: determine a correct initial placement, then balance vertex positions. For this purpose the concept of *linear segments* is introduced; here a linear segment is a set which contains either a single regular vertex or all dummy vertices introduced to split a single long edge (see Figure 5.4). It is important to put multiple dummy vertices of a linear segment at the same vertical position, so that the associated long edge does not receive too many bend points. For each vertex v we write $S(v)$ for the linear segment for which $v \in S(v)$.

The *segment ordering graph* describes the required order of linear segments. It contains an edge (S_1, S_2) if and only if the linear segments S_1 and S_2 contain vertices $v_1 \in S_1$ and $v_2 \in S_2$ which are located in the same layer L_i and are ordered subsequently, thus their ranks satisfy $r_i(v_2) = r_i(v_1) + 1$. Sander's algorithm (see



(a) A layered graph with its linear segments



(b) The segment ordering graph with a possible numbering and topmost alignment

Figure 5.4: Linear segments and their ordering graph

Algorithm 5.4) sets the vertical position of all vertices by performing a topological sort on the segment ordering graph G_S, which is possible because G_S is acyclic, and then finding the topmost position of each linear segment. This is done by keeping the current size $s(L)$ of each layer L as the topmost position for each new vertex in that layer. A fixed value d is also added, which is expected as input of the algorithm and holds the minimal distance between the drawings of vertices.

Algorithm 5.4: InitialNodePlacement

```
procedure initialPlacement($G$: directed graph, $L_1, \ldots, L_k$: layers, $d$: real)
    define the segment ordering graph $G_S = (V_S, E_S)$ by
        $V_S$ := set of linear segments in $G$
        $E_S := \emptyset$
    foreach layer $L_i$ do
        foreach $u, v \in L_i$, $r_i(v) = r_i(u) + 1$, do
            $E_S := E_S \cup \{(S(u), S(v))\}$
    determine a topological sort $t$ for $G_S$

    $s(L) := 0$ for all layers $L$
    foreach $S \in V_S$ in increasing order of $t(S)$ do
        $p := 0$                              // Topmost position, depending on
        foreach $v \in S$ do                  // current layer sizes
            get the layer $L_i$ of $v$
            $p := \max\{p, s(L_i) + d\}$
        foreach $v \in S$ do
            set vertical position $y(v)$ of $v$ to $p$
            get the layer $L_i$ of $v$
            $s(L_i) := p$ + height of node $v$    // All nodes are assumed to have fixed
end                                           // width and height
```

Because after the initial vertex placement all vertices are at their topmost vertical position, Sander proposes a *pendulum* method [52] to balance the drawing. Here a modified version is used, which is illustrated in Algorithm 5.5. First a reference layer is chosen, whose vertices are kept at their initial positions. All other layers are arranged according to the edges which connect the layers using a formula which is similar to the barycenter method for crossing reduction. These requested position movements must be validated so that

- the order of vertices is preserved,
- the minimal distance between vertices is preserved, and
- the size of the overall drawing does not change.

Such a validation is performed twice for each layer L_i to ensure that linear segments which span over multiple layers are handled correctly.

Algorithm 5.5: BalancedNodePlacement

```
procedure balancedPlacement($G$: directed graph, $L_1, \dots, L_k$: layers, $d$: real)
    initialPlacement($G$, $L_1, \dots, L_k$, $d$)
    find a reference layer $L_r$ for which $s(L_r) = \max\{s(L_i) : i \in \{1, \dots, k\}\}$
    $\Delta(S(v)) := 0$ for all $v \in L_r$                    // The reference layer is not moved
    foreach layer $L_i$, $i > r$ with increasing $i$, do
        createDeltas($L_i$)
        validateDeltas($L_i$)
    foreach layer $L_i$, $i > r$ with decreasing $i$, do
        validateDeltas($L_i$)
    foreach layer $L_i$, $i < r$ with decreasing $i$, do
        createDeltas($L_i$)
        validateDeltas($L_i$)
    foreach layer $L_i$, $i < r$ with increasing $i$, do
        validateDeltas($L_i$)
    foreach linear segment $S$ do
        move each $v \in S$ by $\Delta(S)$
end

// Create initial values $\Delta(v)$ for each $v \in L_i$, indicating by how much
// the node wants to move
procedure createDeltas($L_i$)
    $\Delta_l := 0$                                    // The last assigned $\Delta$ value
    foreach $v \in L_i$ in increasing order of $r_i(v)$, $\Delta(S(v)) = \perp$, do
        if $i > r$ then $E_v := E_i(v)$ else $E_v := E_o(v)$
        if $|E_v| = 0$ then
            $\Delta(S(v)) := \Delta_l$
        else
            $\Delta(S(v)) := \frac{1}{|E_v|} \sum_{\{u,v\} \in E_v} (y(u) + \Delta(S(u)) - y(v))$
            $\Delta_l := \Delta(S(v))$
end

// Change the $\Delta$ values so the size of the graph does not grow and nodes do not overlap
procedure validateDeltas($L_i$)
    $\Delta_m := s(L_r) - s(L_i)$                      // The maximal allowed $\Delta$ value
    foreach $v \in L_i$ in decreasing order of $r_i(v)$ do
        $\Delta_m := \min\{\Delta_m, \max\{\Delta(S(v)), 0\}\}$
        $\Delta(S(v)) := \Delta_m$
end
```

5.1.5 Edge Routing

In classical hierarchical polyline drawing the edge routing phase is very easy: draw all edges connecting vertices of subsequent layers by a straight line, and insert bend points where dummy vertices were used to split a long edge. If we want to achieve

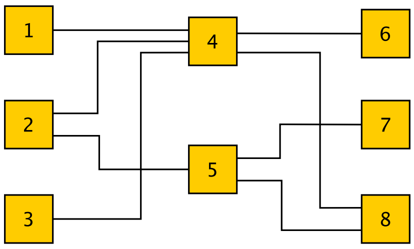


Figure 5.5: Rectilinear edge routing between layers using vertical line segments

rectilinear edge routing, there is a lot more work to do, as each edge that cannot be represented by a single horizontal line needs a vertical line segment (see Figure 5.5). A proper order of vertical line segments is important to avoid additional edge crossings. To accomplish this, each edge e connecting vertices from layers L_i and L_{i+1} is assigned a *routing slot* of rank $r(e)$, which is then drawn at the horizontal position $x := x(L_i) + b(L_i) + r(e)d$, where $x(L_i)$ is the horizontal position at which layer L_i is drawn, $b(L_i)$ is the amount of horizontal space needed by layer L_i, and d is the minimal distance to be left blank between any two line segments. Two bend points are inserted to create the vertical line segment: $(x, y_s(e))$ and $(x, y_t(e))$, where $y_s(e)$ and $y_t(e)$ are the fixed vertical positions of the source and target port of e, respectively. If $E_i \subseteq E$ is the set of edges leaving layer L_i, the amount of horizontal space needed for routing slots depends on the maximal assigned rank $r_{i,\max}$, and the position of L_{i+1} can be determined as $x(L_{i+1}) = x(L_i)+b(L_i)+(r_{i,\max}+1)d$. The set of vertical positions occupied by an edge e is $Y(e) := [\min\{y_s(e), y_t(e)\}, \max\{y_s(e), y_t(e)\}]$; the basic rule for rank assignment is $r(e) \neq r(e')$ for edges e, e' with $Y(e) \cap Y(e') \neq \emptyset$.

I propose a straightforward edge routing method: sort the edges E_i by some criteria and then assign ranks as needed (see Algorithm 5.6). For two edges e, e' with $Y(e) \cap Y(e') \neq \emptyset$ these criteria must decide which edge should get the smaller rank. For example, the following checks could be performed to decide whether $r(e) < r(e')$:

1. $y_s(e) < y_t(e)$ and $y_s(e') < y_t(e')$ and $y_s(e) > y_s(e')$
2. $y_s(e) > y_t(e)$ and $y_s(e') > y_t(e')$ and $y_t(e) < y_t(e')$

The symmetric cases apply for $r(e) > r(e')$, and for remaining cases the edges are treated as equivalent.

Algorithm 5.6: EdgeSortingEdgeRouter

```
procedure routeEdges($L_i, L_{i+1}$: layers)
    $E_i' := \{e \in E_i : y_s(e) \neq y_t(e)\}$    // Set of edges which need a vertical segment
    sort the edges $E_i'$ by some criteria, yielding a sequence $e_1, \ldots, e_{|E_i'|}$
    foreach edge $e_j \in E_i'$ with increasing $j$ do
        $r(e_j) := \min\{n \in \mathbb{N} \mid \forall e_k \in E_i', k < j : r(e_k) = n \Rightarrow Y(e_j) \cap Y(e_k) = \emptyset\}$
end
```

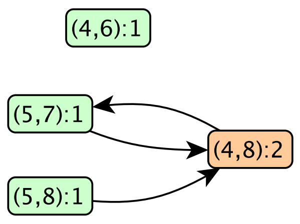


Figure 5.6: A possible coloring of the edges from the second to the third layer of Figure 5.5

Sander suggests sorting the source and target ports of edges and then assigning ranks according to this order [52], as in Algorithm 5.7. Although it is more efficient, this basic version of the proposed method can lead to additional edge crossings, as it would place the edge $(4, 8)$ left of $(5, 8)$ in Figure 5.5. The rank assignment in line 8 needs to be altered to improve this.

Algorithm 5.7: EndpointSortingEdgeRouter

```
1  **procedure** routeEdges($L_i, L_{i+1}$: layers)
2      create ordered list $C$ of entries $(e, y)$, sorted by $y$
3      **foreach** $e \in E_i$, $y_s(e) \neq y_t(e)$, **do**
4          insert $(e, y_s(e))$ and $(e, y_t(e))$ into $C$ in proper order
5      $U := \emptyset$                          // *Set of edges which occupy the current position*
6      **foreach** $(e, y) \in C$ in increasing order **do**
7          **if** $y = \min\{y_s(e), y_t(e)\}$ **then**
8              $r(e) := \max\{r(e') : e' \in U\} + 1$
9              $U := U \cup e$
10         **else**
11             $U := U \setminus e$
12 **end**
```

Another approach is given by Baburin [2], who reduces the problem of rank assignment for edge routing to a graph coloring problem. A new directed graph $G_i^* = (V_i^*, E_i^*)$ is created for each pair (L_i, L_{i+1}) of layers for which edges need to be routed. The nodes of G_i^* are the edges E_i from layer L_i to layer L_{i+1}, and for two edges e_1, e_2 we define $(e_1, e_2) \in E_i^*$ if and only if $Y(e_1) \cap Y(e_2) \neq \emptyset$ and assigning ranks $r(e_1) < r(e_2)$ would not lead to more edge crossings than in the case $r(e_1) > r(e_2)$ (see Figure 5.6). The checks that need to be performed here are the same needed for sorting of the edges in Algorithm 5.6. Now the vertices of G_i^* need to be colored, *i.e.* a map $c : V_i^* \to \mathbb{N}$ must be determined such that $c(e_1) \neq c(e_2)$ for all $(e_1, e_2) \in E_i^*$ and $\max\{c(e) : e \in V_i^*\}$ must be as low as possible. Furthermore we require $c(e_1) < c(e_2)$ if $(e_1, e_2) \in E_i^*$ and $(e_2, e_1) \notin E_i^*$. With the resulting coloring we can set $r(e) := c(e)$.

Until now only Algorithm 5.6 is implemented for edge routing in our layout framework (see Section 7.1).

5.2 Port Constraints

5.2.1 Fixed Ports

When ports are used to determine the source and target point of each edge, the number of crossings does no longer depend only on the order of vertices, but also on the order of ports for each vertex. In the FixedPorts scenario (see Section 4.3) the algorithms for crossing reduction must be extended to handle the prescribed port positions. The FixedPortOrder scenario is reduced to FixedPorts by distributing ports evenly on each side, although optimizations would be conceivable.

In the case of the Barycenter and Median methods the extension is straightforward, because we only need to adjust the formulas to calculate the $a(v)$ values (see Section 5.1.3). This is done by using local *port ranks* for each vertex and extending vertex ranks so that for each $v \in L_1$ and $p \in P(v)$ the sum of the rank of v and the rank of p is unique. Given a set of ordered ports $P(v) = \{p_1, \dots, p_k\}$ for each vertex v and port ranks $r(p_j) := j$, we define the ranks of the ordered vertices $v_1, \dots, v_h$ in the layer L_1 to be

$$r_1(v_i) := \sum_{g<i} |P(v_g)|.$$

Now the new formulas for crossing reduction are as follows.

Barycenter:

$$a_b(v) := \frac{1}{|E_i(v)|} \sum_{(u,v)\in E_i(v)} (r_1(u) + r(p_s(u, v)))$$

Median:

$$a_m(v) := r_1(v(\bar{p})) + r(\bar{p})$$

for $\tilde{P} := \{p_s(e) : e \in E_i(v)\} = \{p_1, \dots, p_k\}$, $r_1(v(p_i)) + r(p_i) < r_1(v(p_{i+1})) + r(p_{i+1})$ for all $i \in \{1, \dots, k-1\}$, $\bar{p} := p_{\lfloor k/2 \rfloor}$

An additional difficulty comes up when the source port of an edge is not on the right side of the source vertex, or the target port is not on the left side of the target vertex. In these cases additional bend points are needed to route the edge around the vertex, as seen in Figure 5.7. For this purpose routing slots of different ranks must be assigned on each side of a vertex, similarly to layer-to-layer edge routing in Section 5.1.5. An additional phase is added after crossing reduction and before node placement; all edges which need additional bend points are processed here, as well as self-loops. A rough sketch of the method is shown in Algorithm 5.8. The ranks assigned in this phase must be considered in the edge routing phase following later on. The maximal number of bend points per edge is 8 if routing around edges is needed, as opposed to 4 in the simple case.

For example, the self-loop $(4, 4)$ in Figure 5.7 is assigned routing slots of rank 1 on the left, bottom and right side of vertex 4, while the edge $(2, 4)$ is assigned a routing slot of rank 2 on the bottom side of vertex 4.

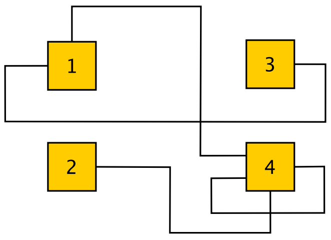


Figure 5.7: Routing of edges around vertices

Algorithm 5.8: AdditionalEdgeRouter

```
procedure routeEdges($L_1, \ldots, L_k$: layers)
    foreach layer $L_i$ do
        foreach $v \in L_i$ do
            foreach port $p$ of $v$ do
                assign routing slots for all self−loops starting in $p$
                assign routing slots for all other edges starting or ending in $p$
            assign ranks to all routing slots of $v$
end
```

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

5.2.2 Free Ports

If the FreePorts scenario is given, a preprocessing step sorts all ports of a vertex by input and output ports, *i.e.* ports at which only incoming or only outgoing edges are incident, respectively. All input ports are put to the left of the corresponding vertex, while output ports are put to the right. By this we achieve

- conformance with the usual drawing convention for ports (inputs left, outputs right),
- consistency with the representation of flow by drawing layers from left to right, and
- no additional bend points, as the additional routing discussed in Section 5.2.1 is not needed.

For this reason it is sufficient to consider the FixedSides scenario as the only alternative to FixedPorts. So now let the side of each port be fixed, but the order of ports on each side be variable. This leads to the additional task of finding an order of ports for each vertex that minimizes the number of crossings, together with the order of vertices in each layer. Again the extension of the methods of crossing reduction from Section 5.1.3 is quite simple: instead of calculating values $a(v)$ to order the vertices, calculate values $a(p)$ to order the ports first, then determine $a(v)$ as the average of all $a(p)$ values for the ports of v. For each port p let $E_i(p)$ be the set of edges which are incoming at that port.

Barycenter:

$$a_b(p) := \frac{1}{|E_i(p)|} \sum_{(u,v)\in E_i(p)} (r_1(u) + r(p_s(u,v)))$$

Median:

$$a_m(p) := r_1(v(\bar{p})) + r(\bar{p})$$

for $\tilde{P} := \{p_s(e) : e \in E_i(p)\} = \{p_1, \dots, p_k\}$, $r_1(v(p_i)) + r(p_i) < r_1(v(p_{i+1})) + r(p_{i+1})$ for all $i \in \{1, \dots, k-1\}$, $\bar{p} := p_{\lfloor k/2 \rfloor}$

5.2.3 Hyperedges

The HYPEREDGES convention must be considered at multiple phases:

- Layer assignment (Section 5.1.2): if more than one long edge is incident at one port, the same dummy vertices should be used (see Figure 5.8).
- Edge routing (Sections 5.1.5 and 5.2.1): edges incident at the same port should use the same routing slots.

The first point can be handled using Algorithm 5.9, which is executed for each edge in the graph and creates connections between vertices of subsequent layers. However, this extension can lead to problems in the following layout phases if crossing reduction is not adjusted. If, for example, backwards crossing reduction is performed for the second layer of the graph in Figure 5.8 while keeping the vertices of the third layer fixed as $(3, \mathsf{c}, \mathsf{d})$, it can happen that the dummy vertex b is placed above a because of its outgoing connection to vertex 3. This means that the linear segments $\{\mathsf{a}, \mathsf{c}\}$ and $\{\mathsf{b}, \mathsf{d}\}$ would cross each other, ultimately leading to a forbidden cycle in the segment ordering graph for node placement.

To resolve this problem, two new rules must be added for each linear segment $S = \{v_1, \dots, v_k\}$ representing a long edge:

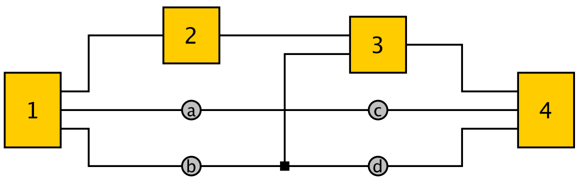


Figure 5.8: The long edges $(1, 3)$ and $(1, 4)$ share the dummy vertex b in layer 2.

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

Algorithm 5.9: LongHyperedges

```
1  procedure createLayerConnection($L_1, \ldots, L_k$: layers, $e$: edge)
2      let $L_s$ be the layer for which $v_s(e) \in L_s$
3      let $L_t$ be the layer for which $v_t(e) \in L_t$
4      if $t - s = 1$ then
5          directly connect $v_s(e)$ and $v_t(e)$
6      else
7          // Associations between ports and existing linear segments are created in line 27
8          get the linear segment $S$ associated with the source port $p_s(e)$
9          if $S = \perp$ then
10             get the linear segment $S$ associated with the target port $p_t(e)$
11         if $S = \perp$ then
12             create a new dummy node $d$ in $L_i$, $i := s + 1$
13             create a linear segment $S$ for $d$
14             connect $v_s(e)$ and $d$
15         else
16             // Another edge with the same source or target port exists
17             connect $v_s(e)$ and the dummy node in $S$ whose layer is $L_{s+1}$
18             find the dummy node $d$ in $S$ whose layer $L_i$ has maximal $i < t$

20         while $i < t - 1$ do
21             create a new dummy node $d'$ in $L_{i+1}$
22             add $d'$ to $S$
23             connect $d$ and $d'$
24             $d := d'$, $i := i + 1$

26         connect $d$ and $v_t(e)$
27         associate $S$ with $p_s(e)$ and $p_t(e)$
28 end
```

Routing of hyperedges can be handled by creating routing slots and assigning ranks for each port instead of doing it for each edge. Then all edges which belong to the same port can use the same rank values for routing.

5.2.4 Model Hierarchy and External Ports

If the data flow model to be layouted has HIERARCHY (see Section 2.4), the basic graph structure is extended so that each vertex is itself a nested graph $v = (V_v, E_v)$, which can be empty. Then layout of the resulting tree structure can be performed recursively, as it is done in Algorithm 5.10.

Algorithm 5.10: HierarchyLayout

```
procedure hierarchyLayout(($V, E$): directed hierarchical graph)
    foreach $v = (V_v, E_v) \in V$, $V_v \neq \emptyset$, do
        hierarchyLayout(($V_v, E_v$))
        adjust the size of $v$ to contain the drawing of $(V_v, E_v)$
    perform flat layout of $(V, E)$ using the layered approach
end
```

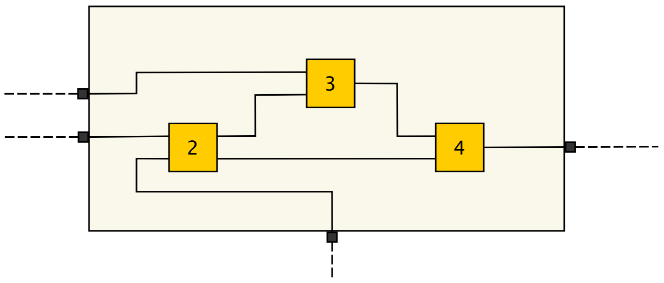


Figure 5.9: Routing of edges to external ports

The additional specification of ExternalPorts leads to another form of nested graphs, in which for each vertex $v = (V_v, E_v)$ the edges of the nested graph may also contain v itself, that is $E_v \subseteq V_v \times V_v \cup \{v\} \times V_v \cup V_v \times \{v\}$. To handle this, the ports $P(v)$ of v are treated as ordinary vertices in the layered graph (V_v, E_v) with layers $L_1, \ldots, L_k$. The first layer L_1 is reserved for input ports, while the last layer L_k is reserved for output ports. This is allowed because input ports are always sources and output ports are always sinks of the layered graph.

During edge routing these input and output port layers must be specially handled, especially if there are input ports which are not on the left side, or output ports which are not on the right side. These cases require additional bend points (see Figure 5.9), and if there are multiple edges which need to be routed along the top or bottom side of the parent vertex, proper ranks need to be assigned to minimize the number of crossings. This could be done using similar techniques as those used for layer-to-layer edge routing.

Our currently used implementation is quite simplistic, since it assigns routing slots on the external sides of the drawing *on demand*, without considering which order of edges actually minimizes the number of crossings. Better methods need yet to be found for this specific subproblem.

6 Alternative Approach: Orthogonal Layout

The term *orthogonal layout* is sometimes used for any layout method that produces rectilinear edge shape, but here it is used in the narrower sense of a method based on orthogonalization, which is called the *topology-shape-metrics* approach [59].

There are three main phases for this approach:

1. Planarization determines the *topology* of the graph, that is a specific planar embedding. If the graph is not planar, vertices must be added to eliminate edge crossings.

2. Orthogonalization adds *shape* by computing 90° bends for all edges, but without specifying the exact length of each line segment. The total number of bends is minimized for a specific embedding, because bend minimization over all planar embeddings is NP-complete [29].

3. Compaction computes the final *metrics*, so that all vertices and bend points have assigned positions.

The basic version of this method, first proposed by Tamassia [57], processes undirected graphs and is therefore not suited for the DIRECTION aesthetic. More recent research has led to extensions for directed graphs, which will be discussed in the following sections.

Many of the algorithms used in this approach expect the input graph to be connected. For this reason we will proceed by applying the topology-shape-metrics approach to each connected component and then recomposing their drawings.

6.1 Planarization

6.1.1 Planar Subgraphs and Edge Reinsertion

The basic scheme for planarization is shown in Algorithm 6.1. To complete the algorithm, we need a method for finding an embedded planar subgraph $G' = (V, E')$, and a method for inserting an edge into a given planar embedding. The latter can be solved efficiently by maintaining the dual graph G^* for the fixed embedding, see Algorithm 6.2. A Breadth First Search (BFS) in the dual graph is used to determine the minimal set of edges which need to be crossed, as seen in Figure 6.1.

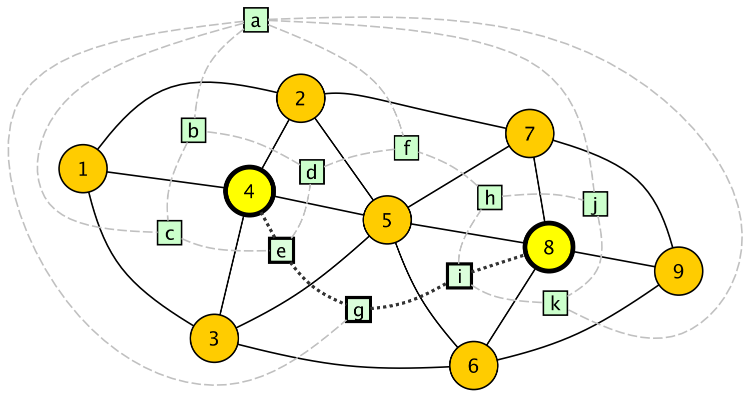


Figure 6.1: The edge $\{4, 8\}$ can be inserted using the shortest path in the dual graph which passes the faces e, g, and i, shown with dotted lines. There are several other admissible shortest paths in this case.

Algorithm 6.1: Planarization

```
procedure planarize($G = (V, E)$: graph)
    find a planar subgraph $G' = (V, E')$ of $G$
    determine a planar embedding for $G'$
    foreach $e \in E \setminus E'$ do
        insert $e$ into $G'$ minimizing the number of crossings
        replace each edge crossing by a dummy node
        update the embedding of $G'$
end
```

Algorithm 6.2: FixedEdgeInsertion

```
procedure insertEdge($e = (u, v)$: edge, $G'$: embedded planar graph, $G^*$: dual graph of $G'$)
    determine sets of faces $F_u, F_v$ which respectively have $u, v$ on their border
    foreach $f \in F_u$ do
        perform a BFS in $G^*$ until a face $f' \in F_v$ is reached
        determine the BFS path $f_1, \ldots, f_h$ with $f_1 = f$, $f_h = f'$
    insert $e$ along the edges used for the first shortest path
end
```

The trivial way of finding a planar subgraph is to use $G' = (V, \emptyset)$, because a graph with no edges is always planar. Another simple alternative is to use a spanning tree of G, which is also always planar. However, planarizing the graph with too many edges which are inserted into a fixed embedding can lead to bad results in some cases (see Figure 6.2). For optimal results a *maximum* planar subgraph must be computed, which is a planar subgraph with the greatest possible number of edges. This problem is NP-hard [27]. Another problem, finding a *maximal* planar subgraph, consists in finding a planar subgraph for which any additionally added edge would make the subgraph non-planar. This can be solved efficiently using Algorithm 6.3

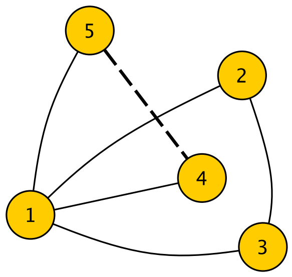


Figure 6.2: The edge $(4, 5)$ cannot be inserted into the fixed embedding without creating a crossing, although the graph is planar.

and a method for testing planarity of a graph, which is usually done by testing each biconnected component, because a graph is planar if and only if its biconnected components are planar. There are linear time algorithms to test a biconnected graph for planarity, *e.g.* the algorithm of Hopcroft and Tarjan [36].

Algorithm 6.3: MaximalPlanarSubgraph

```
procedure createSubgraph($G = (V, E)$: graph)
    initialize subgraph $G' = (V, E')$ with $E' = \emptyset$
    foreach $e \in E$ do
        if $(V, E' \cup \{e\})$ is planar then
            $E' := E' \cup \{e\}$
end
```

Gutwenger, Mutzel, and Weiskircher have shown that it is possible to insert an edge into a planar graph minimizing the number of crossings over all possible embeddings [34]. This is done using *SPQR trees* [33, 48], which are a representation of the triconnected components of a graph.

6.1.2 Upward Planarization

If the input graph is directed, as this is the case for data flow diagrams, planarization must be extended to enable the edges to be drawn in a consistent direction. The task of finding a planar embedding that allows such a consistent edge direction is called *upward planarization*.

Testing whether an arbitrary directed graph is upward planar is NP-complete [29], but for single-source graphs, *i.e.* graphs which have exactly one source, it can be done efficiently [38]. The hierarchical layout method in Chapter 5 shows that an upward drawing does always exist if the input graph G is acyclic. This leads to Algorithm 6.4 for upward planarization, which uses a cycle removal step as discussed in Section 5.1.1, then creates a single-source graph and planarizes it. Methods for finding upward planar subgraphs and proper edge insertion are given by Chimani *et al.* [9].

Algorithm 6.4: UpwardPlanarization

```
procedure planarize($G = (V, E)$: directed graph)
    eliminate cycles of $G$
    add a node $\hat{s}$ to $V$
    foreach source $s'$ in $G$ do
        add an edge $(\hat{s}, s')$ to $E$
    determine a suitable upward planar subgraph $G' = (V, E')$
    foreach $e \in E \setminus E'$ do
        insert $e$ into $G'$ in a proper manner
        replace each edge crossing by a dummy node
        update the upward embedding of $G'$
end
```

The output of upward planarization is a planar embedding with some annotations for the layering of vertices. To be able to use this information in the topology-shape-metrics approach, the orthogonalization phase must be able to handle the embedding properly so that the resulting shape of the drawing is really upward. An alternative would be to use the annotated embedding as input for the node placement and edge routing phases of the hierarchical layout method. In this case upward planarization is used to replace the layering and crossing reduction phases used in hierarchical layout [9].

As upward planarity has been investigated for quite some time, there are more approaches to it, such as the work of Didimo [12]. Their complete discussion would exceed the scope of this thesis.

6.1.3 Embedding Constraints

In a recent work Gutwenger, Klein, and Mutzel proposed a formalism to describe constraints for the planarization phase and gave algorithms to efficiently planarize a graph under such constraints [31]. Constraints on the order of incident edges can be assigned individually to each vertex.

An *embedding constraint* $C(v)$ for a vertex v is an ordered tree whose leaves are the edges which are incident to v. The inner vertices of $C(v)$ are called *constraint nodes* and are assigned an attribute of type *gc*, *mc*, or *oc*. The order of the leaves of $C(v)$ together with the constraint nodes represents the set of admissible cyclic clockwise orders of the edges incident to v. Constraint nodes can influence these admissible orders:

- **gc** (grouping constraint node): The order of children may be arbitrarily permuted.
- **mc** (mirror constraint node): The order of children may be reversed.
- **oc** (oriented constraint node): The order of children is fixed.

The constraint tree shown in Figure 6.3 admits 12 different cyclic orders for the edges $e_1, \ldots, e_6$; for example, $(e_1, e_2, e_3, e_5, e_4, e_6)$ and $(e_3, e_2, e_1, e_6, e_4, e_5)$ are allowed.

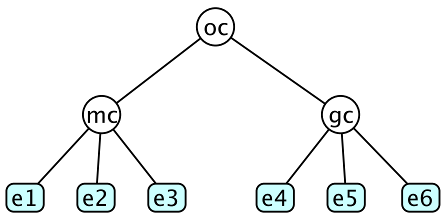


Figure 6.3: An embedding constraint for a vertex of degree 6

Embedding constraints are very convenient to describe all four scenarios for port constraints given in Section 4.3. Given a vertex v with ports $P(v)$, the corresponding embedding constraint can be constructed as follows:

FreePorts: Set the root of $C(v)$ to a gc-node n_r. Create a gc-node n_p for each port $p \in P(v)$ and add it to n_r. Add each incident edge as child of n_p for the corresponding port p. The side of each port must be determined in the orthogonalization phase.

FixedSides: Set the root of $C(v)$ to an oc-node n_r. Create a gc-node n_s for each side $s \in \{\mathrm{N}, \mathrm{E}, \mathrm{S}, \mathrm{W}\}$. Create a gc-node n_p for each port $p \in P(v)$ and add it to n_s for the corresponding side s. Add each incident edge as child of n_p for the corresponding port p.

FixedPortOrder Set the root of $C(v)$ to an oc-node n_r. Iterate over the ports $p \in P(v)$ in proper order, create a gc-node n_p for each p and add it to n_r. Add each incident edge as child of n_p for the corresponding port p.

FixedPorts: Since the exact position of ports is not relevant for planarization, the same structure as for FixedPortOrder can be used.

A planar embedding of a graph G with embedding constraints C is called *ec-planar* with respect to C if for all constrained vertices $v \in V$ the order of edges incident to v is admissible in $C(v)$. (G, C) is *ec-planar* if there exists an ec-planar embedding of G with respect to C. Gutwenger *et al.* have shown that ec-planarity can be tested by performing a suitable *expansion* $E(G, C)$ of G, testing the resulting graph $E(G, C)$ for planarity and applying additional checks to the SPQR tree of each biconnected component of $E(G, C)$ [31, 48]. They also gave an algorithm to insert an edge respecting embedding constraints with the minimal number of crossings among all ec-planar embeddings.

Optimal edge insertion was not yet realized in our current implementation; instead of that, I propose a modification of Algorithm 6.2 that can be used to insert an edge e into a fixed ec-planar embedding. The exact description of such an algorithm is missing in the paper of Gutwenger *et al.* [31].

What we need is a method for finding the set of admissible faces at the endpoints of $e = \{u, v\}$ depending on the embedding constraints $C(u)$ and $C(v)$. Let $G' = (V, E')$

be a graph with an ec-planar embedding represented by incidence lists $E(v)$ for each $v \in V$. The *rank* of an edge e with respect to an endpoint v of e is the index of e in the incidence list of v. Given an edge $e_i \notin E'$ and an endpoint v of e_i, Algorithm 6.5 traverses the constraint tree and recursively computes for each constraint node n the number of already placed edges $\pi(n)$, the smallest rank $\varrho(n)$ of already placed edges, and the set $\alpha(n)$ of admissible ranks for e_i, relative to the edges in n.

Algorithm 6.5: ECAdmissibleFaces

```
procedure admissibleFaces($e_i$: edge to be inserted, $v$: an endpoint of $e_i$)
    let $r$ be the root of the embedding constraint tree $C(v)$
    visitConstraint($r$)
    return $\alpha(r)$
end

// Traverse the constraint tree with recursive calls and collect information about the edges
procedure visitConstraint($n$: constraint node)
    if $n$ is an edge then
        if $n = e_i$ then
            $\pi(n) := 0$
            $\alpha(n) := \{0\}$
        else if $n \in E'$ then
            $\pi(n) := 1$
            $\varrho(n) :=$ rank of $n$ in $v$
        else $\pi(n) := 0$
    else
        // The constraint node has children $\{c_1, \ldots, c_k\}$
        foreach child $c_j$ do visitConstraint($c_j$)
        $\pi(n) := \sum_{j<k} \pi(c_j)$
        $\varrho(n) := min\{\varrho(c_j) : j < k, \pi(c_j) > 0\}$

    if $n$ is an oc−node then
        There is at most one child $c$ which has $\alpha(c) \neq \perp$; copy $\alpha(n)$ from $\alpha(c)$, if it exists,
            and add the sum of $\pi(c_j)$ values for preceding $c_j$ to each value in $\alpha(n)$.
    if $n$ is a mc−node then
        Determine whether normal or mirrored order is used from the position of the child
            $c$ with smallest $\varrho(c)$.
        if this order is ambiguous then consider both variants in $\alpha(n)$
        Copy $\alpha(n)$ from the child which has $\alpha(c) \neq \perp$, if it exists, and add the sum of
            $\pi(c_j)$ values for subsequent (mirrored order) or preceding (normal order) $c_j$
            to each value in $\alpha(n)$.
    if $n$ is a gc−node then
        reorder the children of $n$ according to their $\varrho$ values
        if there is a child $c$ for which $\alpha(c) \neq \perp$ then
```

```
            if $\pi(c) > 0$ then
                Copy $\alpha(n)$ from $\alpha(c)$, and add the sum of $\pi(c_j)$ values for preceding $c_j$ in
                    the new order to each value in $\alpha(c)$.
            else
                $\alpha(n) := \alpha(c)$
                foreach $c_j$ in the new order do
                    foreach $r$ in $\alpha(c)$ do
                        add $r + \sum_{i<j} \pi(c_i)$ to $\alpha(n)$
end
```

6.2 Orthogonalization

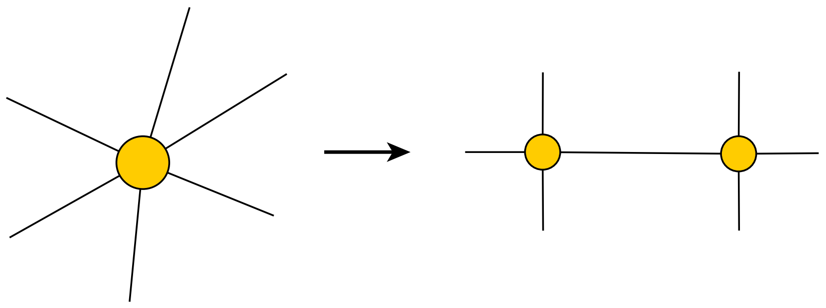
6.2.1 The GIOTTO Approach

The first algorithm for orthogonalization was proposed by Tamassia [57]. Each vertex is represented by a single point in a grid, and the approach is limited to vertices with maximal degree of four. The idea is to determine a series of orthogonal left and right bends for each edge, and to specify the length of each line segment in the following compaction phase (Section 6.3). This method was extended in GIOTTO [59] to handle vertex degrees greater than four: each vertex v is expanded to a rectangular structure of vertices $v_1, \ldots, v_k$ such that the incident edges of v can be distributed to $v_1, \ldots, v_k$ and each v_i, $i \leq k$, has at most four incident edges (see Figure 6.4). In the final drawing v is represented by a rectangle which is made large enough to cover all grid points assigned to $v_1, \ldots, v_k$. The order of incident edges is inherited from the planar embedding; in addition, each edge must be assigned a side of the vertex at which it is attached. This is very convenient with FixedSides or more restricted scenarios for port constraints, but can lead to unnecessary bends in the FreePorts scenario.

Klau and Mutzel have proposed another extension which does not need to specify the side of each edge [43]. Here each high degree vertex is expanded to a ring structure as shown in Figure 6.5(a), and the enclosed face is constrained to be of rectangular shape. To prevent vertices from becoming too large in the final drawing, a quasi-orthogonal drawing is obtained using straight-line segments as in Figure 6.5(b). If a real orthogonal drawing is required, an additional bend could be inserted as in Figure 6.5(c).

The approaches for orthogonalization discussed so far are originally implemented with a transformation to a flow network, for which a flow of minimal cost is computed. An alternative is to formulate the problem as an Integer Linear Program (ILP), which can be processed using a constraint solver library. For this purpose we will introduce some new notation.

Given an embedded undirected graph $G = (V, E)$, we define the set of directed


(a) Structure of vertices created for a vertex with degree 6

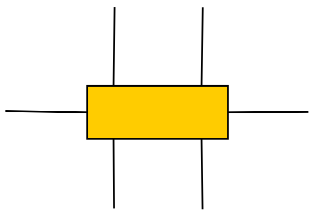

(b) Final representation of the vertex

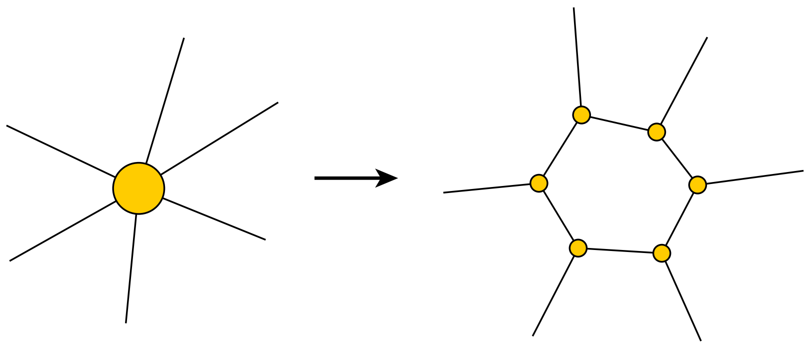
Figure 6.4: Representation of high degree vertices in the GIOTTO model

edges ε and the cyclic ordered sets of incident edges $\varepsilon(v)$ for each $v \in V$, and write

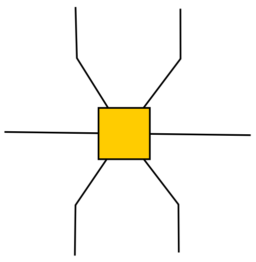
$$\begin{aligned} \varepsilon &:= \{(v,w),(w,v) : \{v,w\} \in E\}, \\ \varepsilon(v) &:= \{(v,w) : \{v,w\} \in E\} = \{e_1,\dots,e_k\}, \end{aligned}$$

and F_I, F_O for the sets of inner and outer faces, respectively, where $|F_O| = 1$. The edges in $\varepsilon(v)$ are ordered according to the given embedding. Each face f is a cyclic ordered set containing the edges found when traversing the border of f with f on the right side (see Figure 6.6). For each $e = (v,w) \in \varepsilon$ we define variables r_e for the number of right bends and l_e for the number of left bends on the edge $\{v,w\}$ in direction v to w. Furthermore we define for each $e = (v,w) \in \varepsilon$ a variable a_e which denotes the angle between e and its cyclic predecessor in $\varepsilon(v)$ at vertex v as multiples of 90°. The resulting ILP for the GIOTTO model is the following [18]:

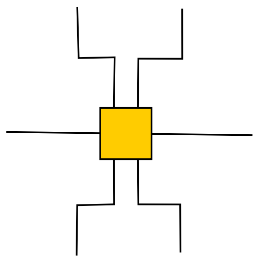
$$\begin{array}{lll} \text{minimize} & \sum\limits_{\{v,w\}\in E} \left(l_{(v,w)} + r_{(v,w)}\right) \text{ subject to} & \\ (T1) & \sum\limits_{e\in\varepsilon(v)} a_e = 4 & \forall v \in V \\ (T2) & \sum\limits_{e\in f} (a_e + l_e - r_e) = \begin{cases} 2|f| - 4 & f \in F_I \\ 2|f| + 4 & f \in F_O \end{cases} & \forall f \in F_I \cup F_O \\ (T3) & l_{(v,w)} = r_{(w,v)} & \forall (v,w) \in \varepsilon \\ & l_e, r_e \in \mathbb{N} & \forall e \in \varepsilon \\ & a_e \in \{1,\dots,4\} & \forall e \in \varepsilon \end{array}$$



(a) Structure of vertices created for a vertex with degree 6



(b) Quasi-orthogonal representation



(c) Orthogonal representation with additional bends

Figure 6.5: Representation of high degree vertices proposed by Klau and Mutzel

$(T1)$ expresses the fact that the angles of incident edges around each vertex must form a total of 360°. $(T2)$ is a condition used to keep the edge bends consistent with the shape of each face [57]. $(T3)$ is needed in order that the two directed versions of each undirected edge have consistent bend numbers.

6.2.2 The Kandinsky Approach

The *Kandinsky* approach of Fößmeier and Kaufmann aims at supporting high degree vertices without having to expand them as in the GIOTTO approach [21, 22]. This is done by allowing multiple edges to touch the same side of a vertex. To avoid overlapping of vertices and edges an important constraint is added: for two edges $\{v, w\}$ and $\{v, u\}$ which are attached at the same side of v at least one of the edges must bend. Edge bends which are added due to this restriction are called vertex-bends, whereas other bends are called face-bends.

Using the notation from Section 6.2.1 we introduce new variables $l_e^v, r_e^v, l_e^w, r_e^w$ for the vertex-bends of each edge $e = (v, w) \in \varepsilon$. The variables l_e, r_e are used for the

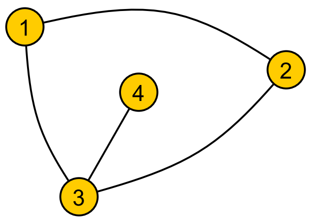


Figure 6.6: The inner face can be represented by f={(1,2),(2,3),(3,4),(4,3),(3,1)}.

face-bends of e. The ILP formulation of the Kandinsky model is as follows:

$$\begin{array}{lll}
\text{minimize} & \sum_{\{v,w\}\in E}\left(l^v_{(v,w)} + l_{(v,w)} + l^w_{(v,w)} + r^v_{(v,w)} + r_{(v,w)} + r^w_{(v,w)}\right) \text{ subject to} & \\
(K1) & \sum_{e\in\varepsilon(v)} a_e = 4 & \forall v \in V \\
(K2) & \sum_{e=(v,w)\in f} (a_e + l^v_e + l_e + l^w_e - r^v_e - r_e - r^w_e) & \\
 & \quad = \begin{cases} 2|f|-4 & f\in F_I \\ 2|f|+4 & f\in F_O \end{cases} & \forall f\in F_I\cup F_O \\
(K3) & l^v_{(v,w)} + r^v_{(v,w)} \le 1 & \forall (v,w)\in\varepsilon \\
(K4) & a_{(v,w)} + l^v_{(v,u)} + r^v_{(v,w)} \ge 1 & \forall v\in V\ \forall (v,u),(v,w) \\
 & & \text{subsequent in } \varepsilon(v) \\
(K5) & l^v_{(v,w)} = r^v_{(w,v)},\ l_{(v,w)} = r_{(w,v)},\ l^w_{(v,w)} = r^w_{(w,v)} & \forall (v,w)\in\varepsilon \\
 & l^v_{(v,w)}, r^v_{(v,w)}, l^w_{(v,w)}, r^w_{(v,w)} \in \{0,1\} & \forall (v,w)\in\varepsilon \\
 & l_e, r_e \in \mathbb{N} & \forall e\in\varepsilon \\
 & a_e \in \{0,\dots,4\} & \forall e\in\varepsilon
\end{array}$$

Constraints $(K1)$, $(K2)$, and $(K5)$ are analogous to $(T1)$, $(T2)$, and $(T3)$, respectively. $(K3)$ means that each edge may have at most one vertex-bend for each endpoint. $(K4)$ expresses the basic rule that at least one of multiple edges attached at the same vertex side must bend.

Eiglsperger *et al.* presented extensions of the Kandinsky method to add drawing constraints using additional variables and rows for the ILP [18, 16]. These ideas can be used to add side constraints for FixedSides or more restricted port constraints scenarios. A fixed order of ports does not need to be handled in the orthogonalization phase, because if the given embedding is ec-planar (see Section 6.1.3), the port order is implied by the fixed order of incident edges for each vertex. The port constraints introduced by Eiglsperger *et al.* are only needed in a mixed scenario that allows constrained edges and free edges at the same vertex. As stated in Section 2.3, we have no need for such a scenario. Therefore I propose slightly modified side constraints to obtain a simplified model based on Eiglsperger's ideas [16].

Now let the side of each $(v,w)\in\varepsilon$ be prescribed by values $s^v_{(v,w)}, s^w_{(v,w)} \in \{\mathrm{N},\mathrm{E},\mathrm{S},\mathrm{W}\}$ for the attachment side at vertices v and w. For each $v\in V$ we require the ordered list $\varepsilon(v) = \{e_1, \ldots, e_k\}$ to start with the first edge $e_1 = (v, w)$ as seen from the top left corner of v in clockwise direction, and we introduce a new variable c_v, which denotes the angle between e_1 and a fictive edge attached at the top side of v, as multiples of 90°. For example, a vertex v which has only one incident edge attached on the right side would have a value $c_v = 1$. To ensure that the c_v values remain consistent in the whole graph, we add the following rows to the ILP [18]:

$$\begin{array}{lll}
(C1) & c_v + \sum\limits_{1<i\leq k} a_{e_i} \leq 4 & \forall v \in V, \varepsilon(v) = \{e_1, \ldots, e_k\} \\
(C2) & c_v + \sum\limits_{1<i\leq k'} a_{e_i} + l^v_{(v,w)} + l_{(v,w)} + l^w_{(v,w)} & \forall \{v,w\} \in E, \\
 & -r^v_{(v,w)} - r_{(v,w)} - r^w_{(v,w)} - c_w - \sum\limits_{1<i\leq l'} a_{e'_i} & \varepsilon(v) = \{e_1, \ldots, e_k\}, (v,w) = e_{k'}, \\
 & +4h_{(v,w)} = 2 & \varepsilon(w) = \{e_1, \ldots, e_l\}, (w,v) = e_{l'}
\end{array}$$

Constraint $(C1)$ expresses that the angles of edges cannot exceed 360°. $(C2)$ is used to keep the c_v values consistent between all connected vertices. The variables h_e are added for each edge $e \in \varepsilon$; in $(C2)$ they express that the sum of the preceding variables modulo 4 should be 2. The prescribed sides can be set using the following constraint:

$$(C3) \quad c_v + \sum_{1<i\leq k} a_{e_i} = \begin{cases} 0 & s^v_{(v,w)} = \mathrm{N} \\ 1 & s^v_{(v,w)} = \mathrm{E} \\ 2 & s^v_{(v,w)} = \mathrm{S} \\ 3 & s^v_{(v,w)} = \mathrm{W} \end{cases} \quad \begin{array}{l} \forall (v,w) \in \varepsilon, \\ \varepsilon(v) = \{e_1, \ldots, e_k\}, (v,w) = e_{k'} \end{array}$$

The correctness of the ILP formulation of the Kandinsky model and of additional constraints has been shown by Eiglsperger [16].

Further work on the Kandinsky model was done by Barth, Mutzel, and Yildiz, who provided a polynomial time algorithm which approximates the minimal number of bends, thus avoiding the possibly exponential running time for solving the ILP [4].

6.3 Compaction

The final phase of this orthogonalization approach has as input an *orthogonal representation*, that is an embedded graph enriched with information about bends and edge directions, and computes concrete lengths for all line segments of edges, so that position values for vertices and bend points can be implied. Optimization goals are minimal area of the final drawing, minimal total edge length and minimal length of the longest edge. All three optimization problems are NP-complete for general orthogonal representations [49].

As most compaction algorithms expect the graph to have a maximal vertex degree of four, the graph must be transformed into a normalized form [16], if this was not already done for the orthogonalization phase (see Section 6.2.1). This is done by replacing each vertex v by a structure which consists of one vertex for each corner of v and one vertex for each port of v, as seen in Figure 6.7. Additionally, all bend points are replaced by vertices.

An orthogonal representation is called *refined* if all faces have rectangular shape. There are very efficient algorithms for refined orthogonal representations, such as Algorithm 6.6, which runs in linear time and minimizes area of the drawing. An orthogonal representation can be turned into a refined form by adding vertices and edges [6].

Algorithm 6.6: FastRefinedCompaction

```
procedure compact($G$: embedded planar graph with refined orthogonal representation)
    Construct graph $G_h^*$ by contracting maximal paths of vertical edges to a node and
        orienting horizontal edges from left to right.
    Construct graph $G_v^*$ by contracting maximal paths of horizontal edges to a node and
        orienting vertical edges top down.

    foreach $v^*$ in $G_h^*$ do
        $x :=$ number of edges on the longest path to a source in $G_h^*$
        foreach $v$ in $G$ for which one of the edges contracted to $v^*$ is incident do
            set the horizontal grid coordinate of $v$ to $x$

    foreach $v^*$ in $G_v^*$ do
        $y :=$ number of edges on the longest path to a source in $G_v^*$
        foreach $v$ in $G$ for which one of the edges contracted to $v^*$ is incident do
            set the vertical grid coordinate of $v$ to $y$
end
```

The drawback of compaction algorithms based on refinement is that the process of refining an orthogonal representation can distort the drawing and can lead to results which are far from optimal. Better approaches have been developed, such as the algorithm of Klau and Mutzel, which is based on an ILP [44], and the algorithm of Eiglsperger and Kaufmann, which is specialized for vertices with prescribed sizes [19].

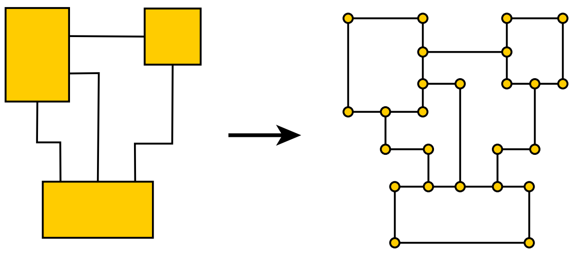


Figure 6.7: Normalization of an orthogonal representation

7 Experimental Results

Implementations of the approaches discussed in this thesis are integrated into the Kiel Integrated Environment for Layout for the Eclipse RichClientPlatform (KIELER)[1]. This chapter gives an overview of KIELER, the main Application Programming Interface (API) used for automatic diagram layout, and experimental results of the implementation.

7.1 The KIELER Framework

The KIELER framework is a platform for experimental approaches to graphical model-based design and for combination of different aspects of graphical modeling, such as methods of model editing, visualization of simulation, and automatic layout. Unlike its preceding project, the Kiel Integrated Environment for Layout (KIEL), which was developed as a stand-alone Java application [50], KIELER builds on Eclipse (Eclipse Foundation, Inc.), an extensible platform comprised of various integrated development environments. The Eclipse project was originally created by IBM in 2001 and is now maintained by an open source community.

Eclipse has some subprojects which form an important basis for KIELER:

- The Eclipse Modeling Framework (EMF) contains tools to create structured data models which can be used as a basis for graphical editors or central data structures of an application. EMF features Java code generation from an abstract model and model storage in eXtensible Markup Language (XML) format. Additional projects provide transformation, validation, and comparison of EMF models.
- The Graphical Editing Framework (GEF) supplies the functionality needed to create a graphical editor in Eclipse, following the model-view-controller design pattern.
- The Graphical Modeling Framework (GMF) can be used to generate Java code for graphical editors on top of EMF and GEF. This is done with additional models which describe different aspects of the generated editor.

One goal of KIELER is to provide multiple graphical editors for model-based design. As a base for experimental layout methods for data flow diagrams a simple data flow editor was created using GMF (see Figure 7.1). This editor was used to create and modify example diagrams, together with an additional generator for data flow models.

[1]http://rtsys.informatik.uni-kiel.de/trac/kieler

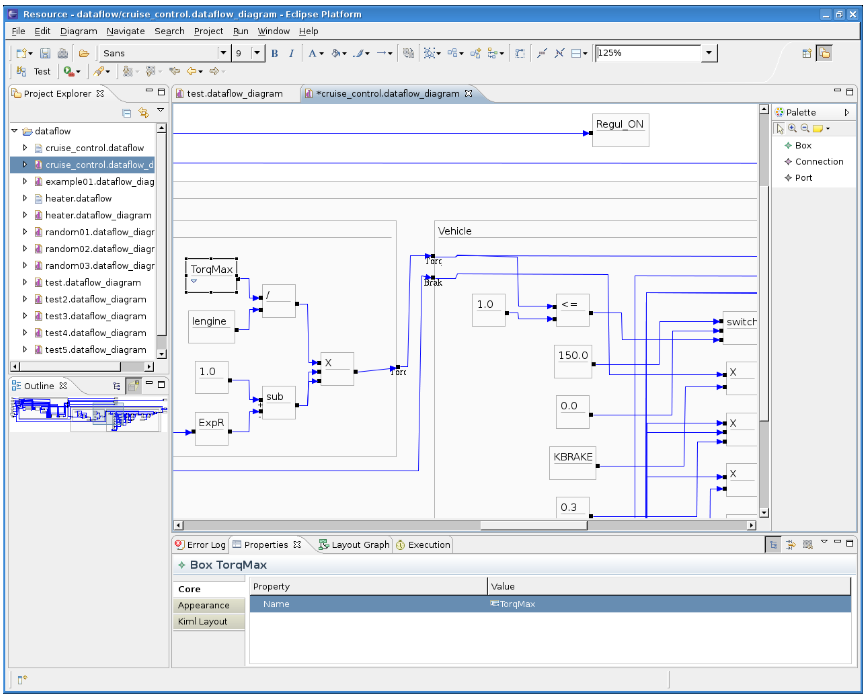


Figure 7.1: A simple data flow editor created with Eclipse GMF

A central subproject of KIELER is the KIELER Infrastructure for Meta Layout (KIML), which has an interface that connects GMF diagram editors with the layout algorithms. The basic work on KIML was done by Schipper [53], who also created a KIML interface for the Graphviz layout tool (see Section 3.3).

7.2 Implementation

The Eclipse platform is composed of a multitude of plug-ins, *i. e.* Java libraries which are packed into JAR files and loaded on demand. Consequently, all subprojects of KIELER, including the single layout algorithms, are implemented in Java as Eclipse plug-ins. This allows users to install separate subprojects independently from each other. The implementation of the hierarchical layout algorithm described in Chapter 5 is available as a *feature* for Eclipse on the KIELER web page[2]. The same files can alternatively be used as Java class libraries outside of Eclipse. The API is outlined in Section 7.2.1; more details and the complete source code can be found on the project web page.

[2]http://rtsys.informatik.uni-kiel.de/trac/kieler

7.2.1 Layout API

Layout algorithms are able to connect to KIML using the abstract class KimlAbstractLayoutProvider, which is the superclass for all base classes of implemented layout algorithms. The central data structure is KGraph, an EMF model used to exchange graph-based data between different modules, to add supplementary information to each element of the graph, and to store it as XML file. This graph model can contain ports as well as layout data such as location and size of each vertex. Progress of an algorithm can be tracked using a *progress monitor*, which can be used to display a progress bar during computation, enable the user to abort it, and measure execution times (see Section 7.3.2). The relevant classes for the layout API are shown as a class diagram in Figure A.1 (Appendix A).

The steps required to execute a KIML layout algorithm are as follows.

1. Create an instance of KGraph that represents the graph to be layouted. If the source is a diagram, this means mapping the model underlying the diagram to an instance of the EMF model KGraph. If performance of automatic layout is crucial, keeping a single instance consistent with the diagram is also an option.

2. Create a progress monitor instance, and connect it to the user interface if needed.

3. Create an instance of the selected *layout provider*, that is a subclass of KimlAbstractLayoutProvider holding an implementation of the desired layout algorithm. This instance may be buffered, so that the same instance is used when layout is performed repeatedly, which significantly boosts performance.

4. Execute the doLayout method of the layout provider, passing the top vertex of the KGraph instance and the progress monitor as arguments. The reason why a vertex is passed instead of the whole graph is that each vertex may contain internal vertices and edges to support hierarchy as introduced in Section 2.4. If the graph has hierarchical structure, layout must be applied recursively, beginning with the deepest nested graph.

5. The KGraph instance is now enriched with layout information; read this data and apply it to the original graph model, or directly to the corresponding diagram.

General options of the layout algorithms can be set using a *preference store*, that is an interface that maps named preferences to their values. Eclipse provides preference stores that can be manipulated on *preference pages* (see Figure 7.2) and stored in XML format (see Listing 7.1). If the layout library is used outside of Eclipse, a specialized implementation of preference stores is required, otherwise the algorithms will only use default settings.

Options related to a special layout instance can be set as supplementary information for each object in the KGraph structure. Such options include the scenarios for

Listing 7.1: A data flow model stored in XML format

```
<?xml version="1.0" encoding="UTF−8"?>
<xmi:XMI xmi:version="2.0" xmlns:xmi="http://www.omg.org/XMI" xmlns:dataflow="http://www.inf...
  <dataflow:DataflowModel xmi:id="_e5UlIOfiEd20OaZJ2fh_UQ">
    <boxes xmi:type="dataflow:Box" xmi:id="_hVwyYOfiEd20OaZJ2fh_UQ" name="Box">
      <inputs xmi:type="dataflow:InputPort" xmi:id="_jE9fIOfiEd20OaZJ2fh_UQ"/>
      <outputs xmi:type="dataflow:OutputPort" xmi:id="_jWnP0OfiEd20OaZJ2fh_UQ"/>
      <boxes xmi:type="dataflow:Box" xmi:id="_p53hcOfiEd20OaZJ2fh_UQ" name="Box One">
        <inputs xmi:type="dataflow:InputPort" xmi:id="_Yg_−4OfnEd2J26Kmvy4vvQ"/>
        <outputs xmi:type="dataflow:OutputPort" xmi:id="_ZBakUOfnEd2J26Kmvy4vvQ"/>
        <outputs xmi:type="dataflow:OutputPort" xmi:id="_wgqdAOfnEd2J26Kmvy4vvQ"/>
        <connections xmi:type="dataflow:Connection" xmi:id="_7TTgMOfnEd2J26Kmvy4vvQ" ...
        <connections xmi:type="dataflow:Connection" xmi:id="_75sx0OfnEd2J26Kmvy4vvQ" ...
      </boxes>
      ...
```

port constraints presented in Section 2.3. This means that for each vertex it may be individually set whether its ports should be kept fixed or may be moved by the layout provider.

7.2.2 Internal Structure

The layout algorithms were implemented using the *strategy* design pattern, which consists in providing an interface for each phase of the algorithm and one or more implementations. In our application, these concrete algorithm implementations can be selected by the user to customize the layout method. For example, all algorithms for the cycle removal phase implement the interface ICycleRemover. Furthermore, all algorithm implementations extend the abstract class AbstractAlgorithm, which provides basic functionality to handle progress monitors.

Figure A.2 shows a class diagram with all modules of hierarchical layout. The following modules and corresponding implementations are included:

ICycleRemover: Interface for cycle removal (see Section 5.1.1)

- DFSCycleRemover: Cycle removal using DFS
- GreedyCycleRemover: Implementation of Algorithm 5.1

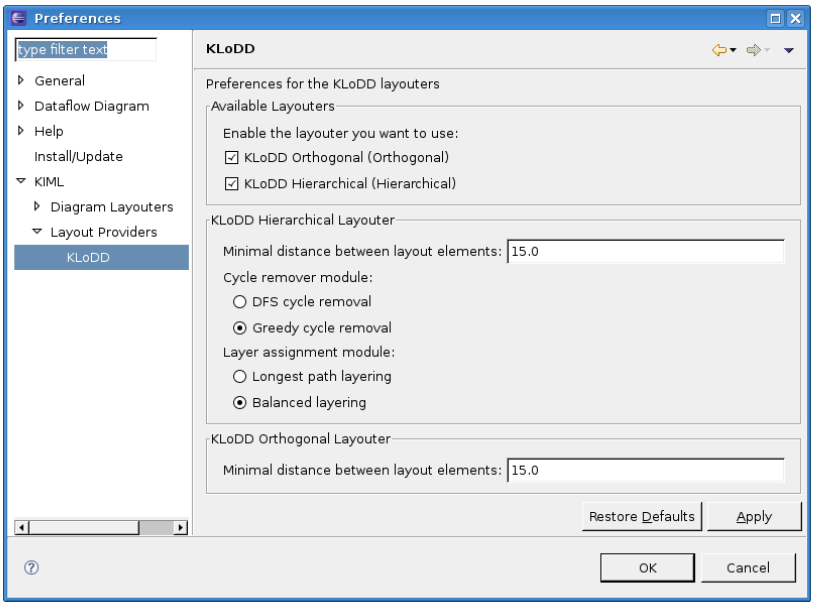
ILayerAssigner: Interface for layer assignment (see Section 5.1.2)

- LongestPathLayerAssigner: Implementation of Algorithm 5.2
- BalancingLayerAssigner: Implementation of Algorithm 5.3

ICrossingReducer: Interface for Crossing Reduction (see Section 5.1.3)

- LayerSweepCrossingReducer: Crossing reduction with a layer-by-layer sweep; needs a single-layer crossing reducer to process each layer.

ISingleLayerCrossingReducer: Interface for crossing reduction for the two-layer crossing reduction problem


Preferences

type filter text

- General
- Dataflow Diagram
- Help
- Install/Update
- KIML
  - Diagram Layouters
  - Layout Providers
    - KLoDD

KLoDD

Preferences for the KLoDD layouters

Available Layouters

Enable the layouter you want to use:

☑ KLoDD Orthogonal (Orthogonal)

☑ KLoDD Hierarchical (Hierarchical)

KLoDD Hierarchical Layouter

Minimal distance between layout elements: 15.0

Cycle remover module:

○ DFS cycle removal

● Greedy cycle removal

Layer assignment module:

○ Longest path layering

● Balanced layering

KLoDD Orthogonal Layouter

Minimal distance between layout elements: 15.0

Figure 7.2: A preference page for customization of layout algorithms

BarycenterCrossingReducer: Implementation of the barycenter method

INodewiseEdgePlacer: Interface for routing of edges around vertices (see Section 5.2.1); the term *nodewise* is used to distinguish this module from the routing of edges between layers, which is called *layerwise* edge placement.

SortingNodewiseEdgePlacer: Implementation for routing around vertices that sorts the edges according to some criteria

INodePlacer: Interface for vertex placement (see Section 5.1.4)

BasicNodePlacer: Implementation of Algorithm 5.4

BalancingNodePlacer: Implementation of Algorithm 5.5

IEdgeRouter: Interface for edge routing (see Section 5.1.5)

RectilinearEdgeRouter: Uses a layerwise edge placer to determine bend points for the edges between each pair of subsequent layers, and provides routing to external ports (see Section 5.2.4).

ILayerwiseEdgePlacer: Interface for routing of edges between two subsequent layers

SortingLayerwiseEdgePlacer: Implementation of Algorithm 5.6

Internally the hierarchical layout algorithm uses a data structure that represents a layered graph; its class diagram is shown in Figure A.3. This data structure contains a representation of *layer elements*, which are vertices of the layered graph, and are related to either a vertex from the original graph, or a dummy vertex introduced because of a long edge, or an external port of the parent vertex. *Layer connections* are edges of the layered graph that connect layer elements from subsequent layers. *Element loops* are edges of the layered graph that represent self-loops.

The modules of the orthogonal layout algorithm are shown in the class diagram in Figure A.4. The currently implemented modules are the following:

IPlanarizer: Interface for the planarization phase (see Section 6.1)

- PortConstraintsPlanarizer: Creates embedding constraints as described in Section 6.1.3, and uses an implementation of ec-planarization such as EdgeInsertionECPlanarizer.
- EdgeInsertionECPlanarizer: Simple implementation of planarization with embedding constraints that removes all edges from the graph and reinserts them using ECEdgeInserter
- ECEdgeInserter: Edge insertion for a fixed ec-planar embedding using BFS in the dual graph; includes Algorithm 6.5.

IOrthogonalizer: Interface for the orthogonalization phase (see Section 6.2)

- KandinskyILPOrthogonalizer: Uses an ILP solver library to solve the Kandinsky ILP given in Section 6.2.2. Currently the library lp_solve is used, which is an open source C library for mixed integer linear programming.

ICompacter: Interface for the compaction phase (see Section 6.3)

- NormalizingCompacter: Creates a normalized orthogonal representation and applies a compaction algorithm for normalized representations, such as RefiningCompacter.
- RefiningCompacter: Creates a refined orthogonal representation and applies a compaction algorithm for refined representations, such as LayeringCompacter.
- LayeringCompacter: Implementation of Algorithm 6.6

The orthogonal layout algorithm uses a graph structure that represents undirected graphs, because most modules expect their input graph to be undirected. The basic structure is shown in the class diagram in Figure A.5. The edges of this structure called *slim graph* (to distinguish it from the EMF structure KGraph) have designated source and target vertices to allow the structure to express direction of edges, *e.g.* for upward planarization. However, the incoming and outgoing edges of each vertex are organized in a common list of incident edges. Each entry of that list has a type value indicating whether the edge is incoming or outgoing. Similarly, each face holds lists of the edges found when traversing the border of that face, and each entry of such a list has a flag that indicates whether the edge is traversed in forward or in backwards direction.

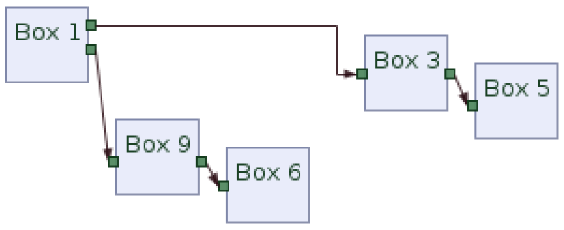


Figure 7.3: A sample output of the orthogonal layout method

7.3 Results

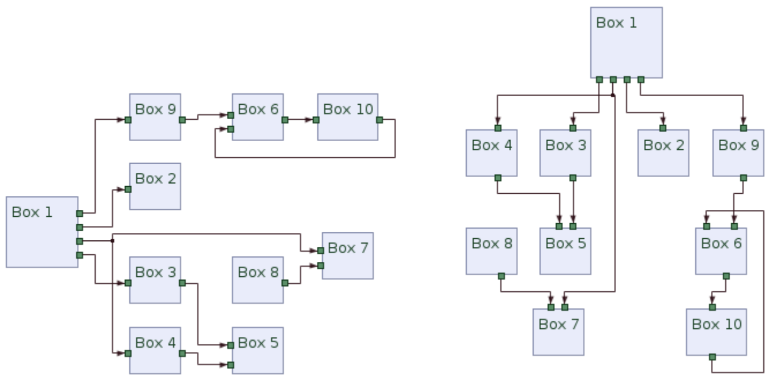
Sections 7.3.1 and 7.3.2 cover experimental results for the hierarchical layout algorithm. For multiple reasons the orthogonal layout was only partly implemented:

- The algorithms for planarization are very complex, especially where ec-planarity is required. Up to now only the edge insertion algorithm for fixed ec-planar embeddings was implemented, and the graph with no edges is used as planar subgraph for edge insertion.
- The Kandinsky model with simplified side constraints is currently implemented as an ILP for orthogonalization (see Section 6.2.2). Probably the current implementation is not yet correct, because sometimes it yields peculiar results.
- For compaction the orthogonal representation is currently normalized and then refined as described in Section 6.3. It is unclear how to apply the resulting grid coordinates to a drawing in which vertices have prescribed size. This leads to very unpleasant drawings, where edges which are meant to be straight appear oblique (see Figure 7.3). Obviously other compaction methods need to be evaluated.

A sample output of the algorithm is shown in Figure 7.3, where the current problems of orthogonalization and compaction are apparent.

The images showing the output of the implemented algorithms were created by directly drawing the layout information of the KGraph data structure, instead of applying this layout to the original diagram. This ensures that the pure algorithm output is visualized, without modification by the diagram framework.

The implementation of hierarchical layout supports both horizontal and vertical layout (see Figure 7.4), but only horizontally arranged drawings are shown to be compatible with the original diagrams from SCADE, Simulink, and Ptolemy, which are also drawn horizontally.



(a) Horizontal layout (b) Vertical layout

Figure 7.4: Output of hierarchical layout with different layout options

7.3.1 Comparison of Layouts

Seven outputs of the hierarchical layout algorithm are presented in Figures 7.5 to 7.11 and compared with hand-made layouts from real examples. This was done by reproducing the original examples in the KIELER simple dataflow editor and then using KIML to execute the layout algorithm. The current implementation provides only rectangular vertex shapes, therefore the operators of the original diagrams are replaced by rectangular boxes.

The FixedPorts scenario was applied to all examples, and the following modules were used.

Cycle removal: GreedyCycleRemoval (Algorithm 5.1)

Layer assignment: BalancedLayering (Algorithm 5.3)

Crossing reduction: Layer-by-layer sweep with barycenter heuristic, one iteration

Node placement: BalancedNodePlacement (Algorithm 5.5)

Edge routing: EdgeSortingEdgeRouter (Algorithm 5.6), including routing around vertices and routing to external ports

Now we will take a closer look at the example diagrams and their generated layouts.

1. (Figure 7.5) In the first example both the hand-made and the automatic layout have a very clear appearance, and the structure of both drawings is similar. The SCADE diagram holds a special detail: the PRE and the initialization operators on the bottom right (merged to a FBY box for the automatic layout) are reversed so they point to the same direction as the following leftward edges, which helps to make the drawing more compact. This feature is not yet supported by present implementations of automatic layout, and it is not clear how desirable this is in practice.

2. (Figure 7.6) The hand-made layout of the second example achieves a good aspect ratio by routing a part of the diagram to the top left, thus compromising the left-to-right flow of data. The automatic layout has some problems, although the direction of flow has a better representation than in the hand-made layout. The routing of edges around the ExternalConditions operator could be improved by changing the order of edges. The two FBY operators and their connected boxes PointReset and null_speed on the bottom right could be moved two layers further to the left. The automode operator would induce less edge crossings if it was placed above the or box.
3. (Figure 7.7) In the third example the LQR and the Discrete State Estimator operators are again reversed in the hand-made drawing, which is arranged very accurately and has only one edge crossing. The automatic layout has some more crossings between the second and the third layer, but the other parts of the drawing have a clear routing of edges.
4. (Figure 7.8) The two back edges of the fourth example are routed through the bottom of the hand-made drawing, while the automatic layout routes them through the middle, which leads to additional crossings. Although the placement of vertices is quite similar in both variants, the overall edge routing of the automatic layout is clearly inferior in this case.
5. (Figure 7.9) The original diagram of example 5 has a good aspect ratio at the cost of less uniform edge direction. In contrast, the automatic layout achieves a very clear emphasis of the data flow, but its aspect ratio is worse. Here we also see that in Ptolemy the representation of edges is slightly different than in the previous examples, as it is not strictly orthogonal.
6. (Figure 7.10) The hand-made layout of example 6 uses two back edges from Symbol1 and Symbol2 to the Multiplexor operator to obtain a better aspect ratio. Again the automatic layout reveals its focus on maintaining the direction of flow. The overall appearance of its drawing is quite good, but the edge going from the GaussNoise box to SeqToArray (names were abbreviated to limit box sizes) has a negative interference with the edge from Symbol2 to correlator2, as they have horizontal line segments that are drawn very close to each other.
7. (Figure 7.11) In the last example we see a very unpleasant hand-made layout, which has obviously received little effort by the designer. The routing of connections is much more perceivable in the automatic layout. However, the latter does not make good use of the diagram area, since a large part does not contain any vertices. A better variant would be to place some of the addition and multiplication operators on top of each other instead of arranging them in a row, but that would violate the proper layering, and would require an extension of the whole layout method.

Since the first six examples that were chosen for comparison originate from official demonstrations of their respective software product, it is likely that the manually arranged layout of these diagrams has been done with special care. This is not always possible in industrial applications, since it can be very time consuming to perform a good manual diagram layout, and time is often scarce when deadlines are pressing. The last example (see Figure 7.11) is taken from a practical course on model-based design, in which students were given the task to model a distributed real-time system using SCADE. This extract of a SCADE diagram demonstrates very well the result of a quick-and-dirty hand-made layout, which is clearly inferior to the automatic layout in this case.

Some statistics for aesthetics criteria in the example drawings and the original diagrams are presented in Table 7.1. The number given for Direction indicates the number of leftward edges in the drawing, which should be as small as possible for an optimal emphasis of flow direction.

The statistics show that the first phase of the hierarchical layout method, cycle removal, has the greatest impact on the drawing by emphasizing Direction, since for this criterion the automatic layout is equal or better than the hand-made drawing in all seven cases. Another factor that contributes to this, but leads to a worse aspect ratio in most cases, is the restriction in layer assignment that edges may only point to the right. Crossing reduction and bend minimization have a rather low priority in the hierarchical method, which is directly reflected by the data in Table 7.1.

Generally, the output of hierarchical layout becomes more confusing when there are many long edges in the layering. Diagrams such as the one in Figure 7.12, which is a generated model with only one outgoing edge per vertex, receive a very compact layout. In the comparisons covered above the average number of outgoing edges per vertex is 1.34.

An interesting aspect that is not covered by the examples so far is the layout of diagrams with hierarchy as in Section 2.4. The hard part here is the routing to external ports. The current implementation of the hierarchical layout method has a basic support for such routing, as can be seen in Figure 7.13.

Table 7.1: Statistics of example diagrams (orig. means *original diagram*, and hl. means *hierarchical layouter*)

	Crossings		Direction		Bends		AspectRatio	
Example	orig.	hl.	orig.	hl.	orig.	hl.	orig.	hl.
(SCADE) 1	0	4	7	1	63	70	1.88	2.06
(SCADE) 2	7	22	5	3	57	81	1.67	2.35
(Simulink) 3	1	6	5	1	23	42	2.77	2.84
(Simulink) 4	1	8	2	2	18	27	2.02	1.48
(Ptolemy) 5	1	2	3	1	19	31	1.57	5.29
(Ptolemy) 6	6	4	2	0	28	25	1.94	3.19
(SCADE) 7	22	18	4	0	46	38	1.53	2.44

7.3.2 Execution Time Analysis

This section presents measurement data for the execution time of the hierarchical layout method, shown in Figure 7.14. Execution times were determined on an AMD Sempron 3000+ processor for different randomly generated graphs. Each value was calculated as the average of the values for five random graphs of equal size, where for each graph the lowest execution time of five consecutive runs was taken. Time values were measured using a specialized progress monitor which allows to track the execution of each submodule of the algorithm. The resulting values can be displayed in a tree view, as seen in Figure 7.15. The example shows that in the current implementation most time is taken by the crossing reduction phase, followed by the edge routing phase.

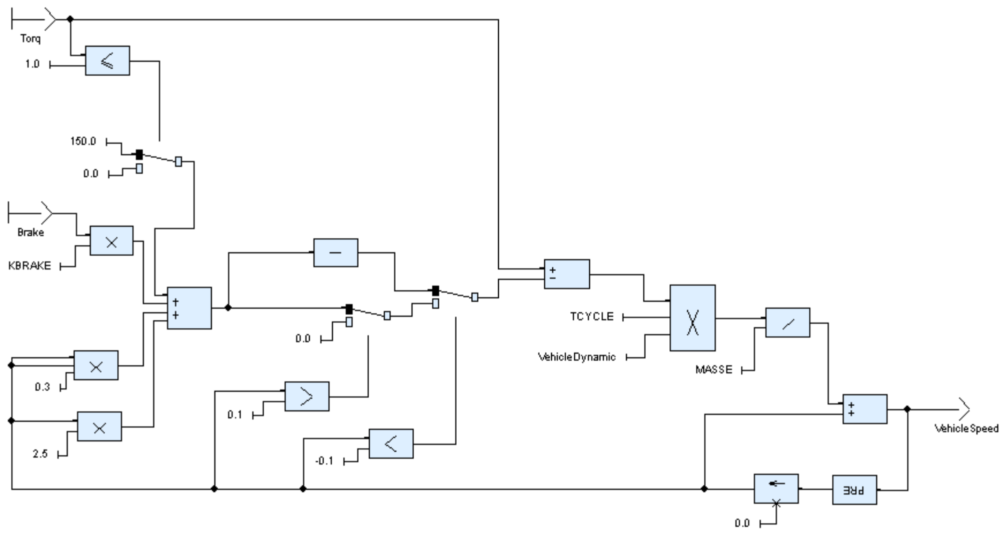
When large graphs are created in GMF, the diagram framework consumes a great amount of memory, so that the user interface becomes too slow to be able to perform proper execution time measurements. For this reason all measurements were carried out using a new separate Java application, which is able to create random graphs, execute a layout algorithm, and store the results of its progress monitor in Comma Separated Values (CSV) format.

Figure 7.14(a) presents measurements for generated graphs $G = (V, E)$ with varying $|V|$ and $|E| = |V|$ in logarithmic scale. The curve is roughly linear with an approximate slope of 1.1, hence the overall runtime behavior is nearly linear[3] in the number of vertices. For graphs with about 15 000 or less vertices the algorithm takes less than a second, which proves its suitability for automatic layout in a user interface environment.

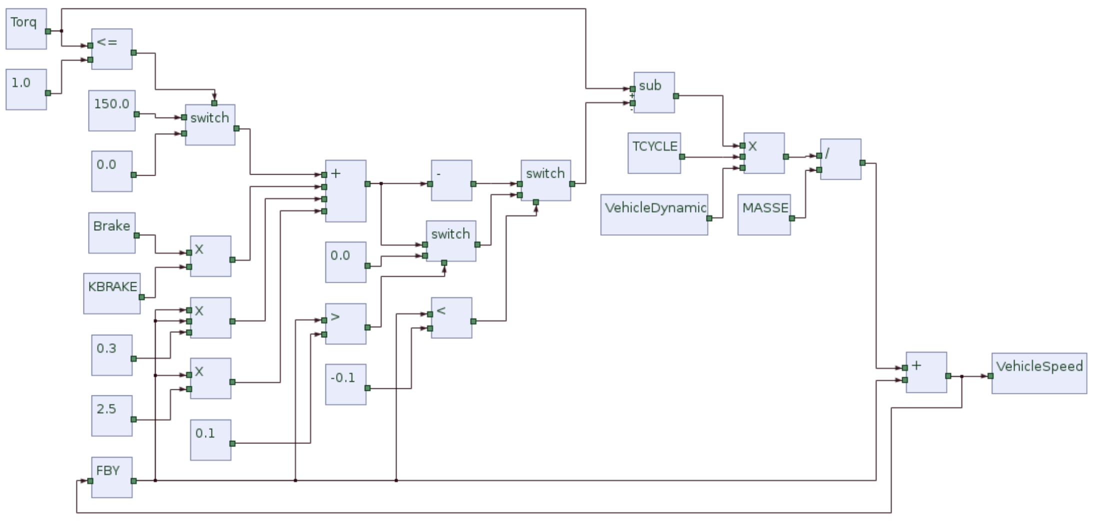
The runtime behavior for generated graphs with a fixed number of 100 vertices and varying number of edges is shown in linear scale in Figure 7.14(b). Here we see that the curve rises much faster than in the previous case, and reaches execution time of one second for only 100 vertices and about 1 700 edges. This proves that the average vertex degree has a much greater impact on the execution time than the total number of vertices and edges. One reason for this is that for vertices with a lot of incident edges the number of long edges that stretch over multiple layers is likely to be high, so that dummy vertices must be inserted to obtain a proper layering. The consequence is that the problem size rises with regard to the total number of vertices. Methods to avoid this in favor of a better asymptotical running time are discussed by Eiglsperger *et al.* [20].

To give an impression of the complexity of the layout problem for such large graphs, Figure 7.16 shows the result of the automatic layout of a graph with 100 vertices and 500 edges.

[3]Real linear runtime behavior would yield a linear curve of slope 1 in logarithmic scale.

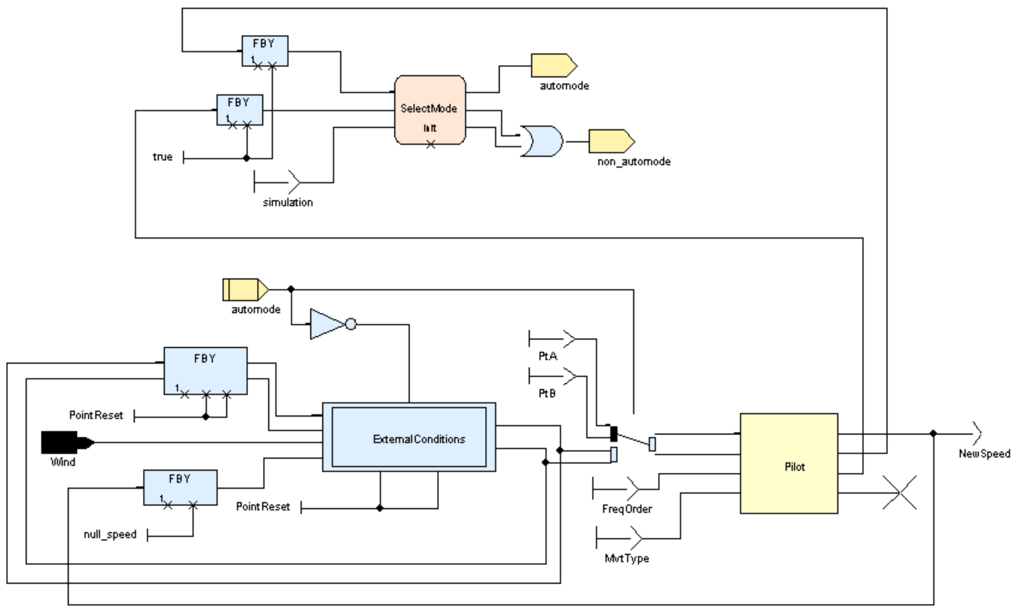


(a) Original SCADE diagram

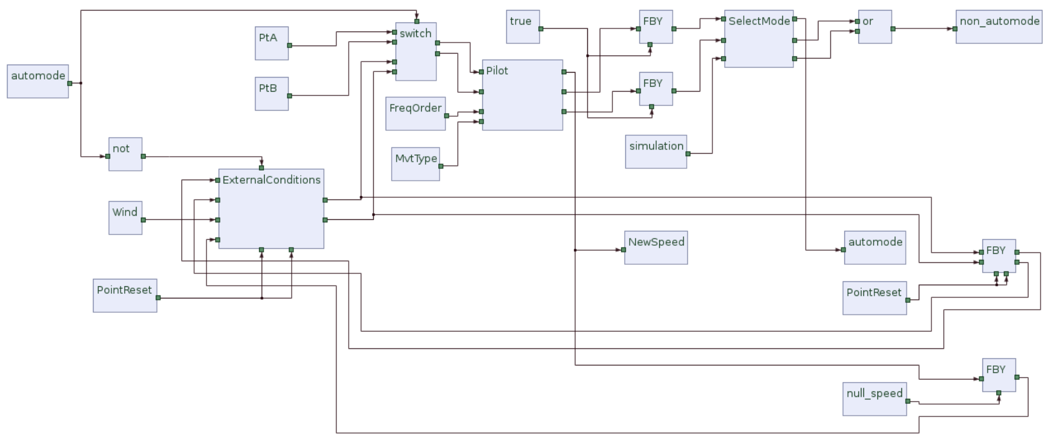


(b) Hierarchical layout

Figure 7.5: Comparison Example 1. Calculation of the speed of a vehicle for the environment simulation of a cruise control system, from a SCADE demonstration

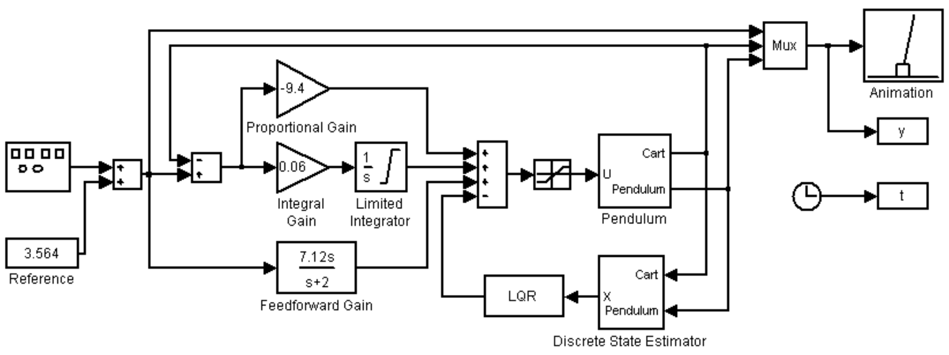


(a) Original SCADE diagram

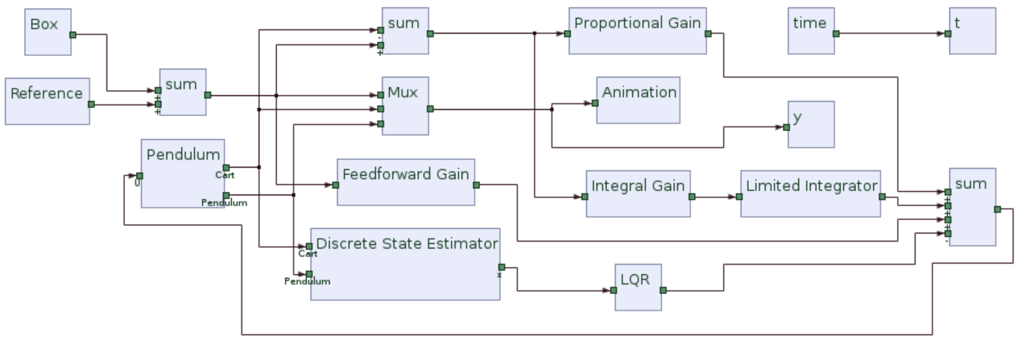


(b) Hierarchical layout

Figure 7.6: Comparison Example 2. Environment model of an airplane system, from a SCADE demonstration

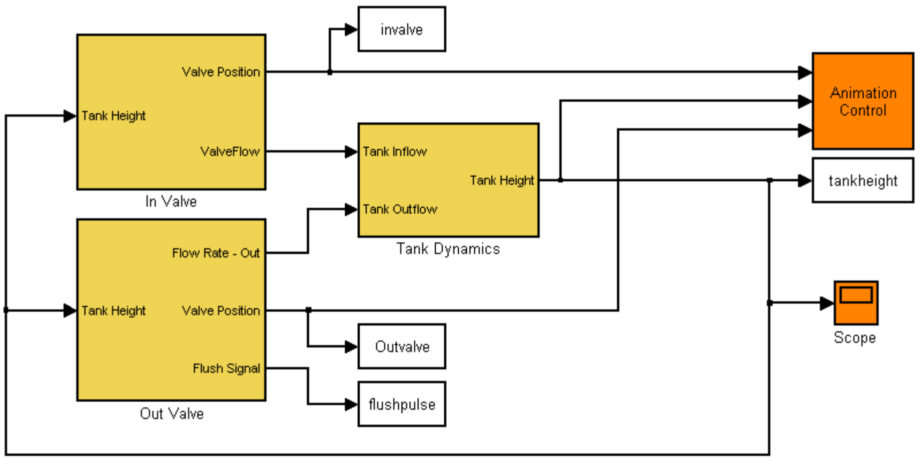


(a) Original Simulink diagram

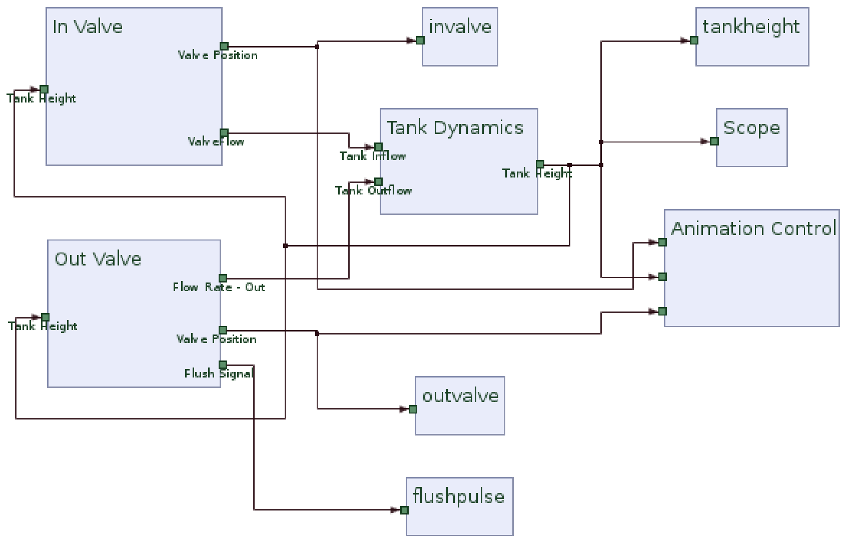


(b) Hierarchical layout

Figure 7.7: Comparison Example 3. Simulation of a cart carrying an inverted pendulum, from a Simulink demonstration

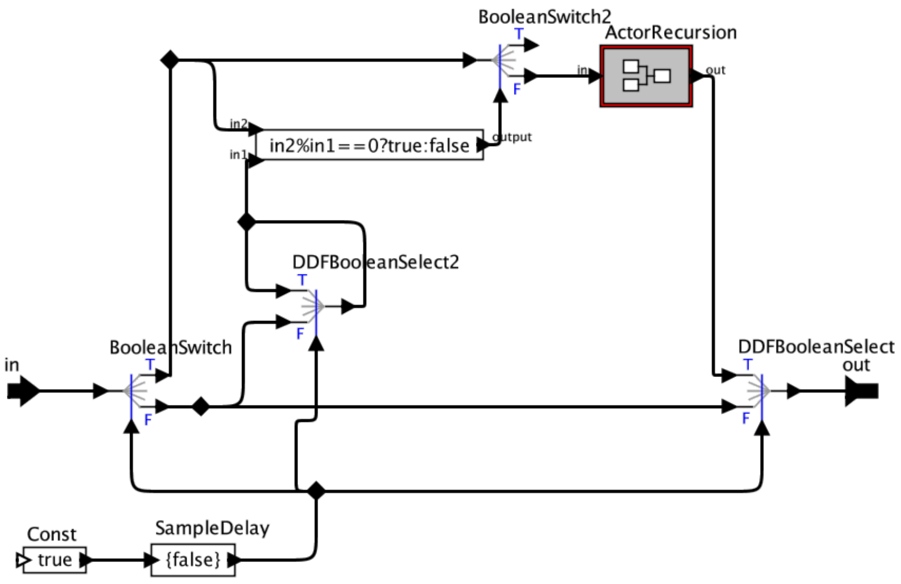


(a) Original Simulink diagram

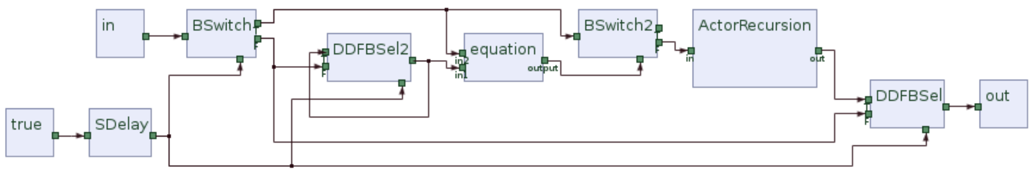


(b) Hierarchical layout

Figure 7.8: Comparison Example 4. Simulation of liquid dynamics in a tank, from a Simulink demonstration

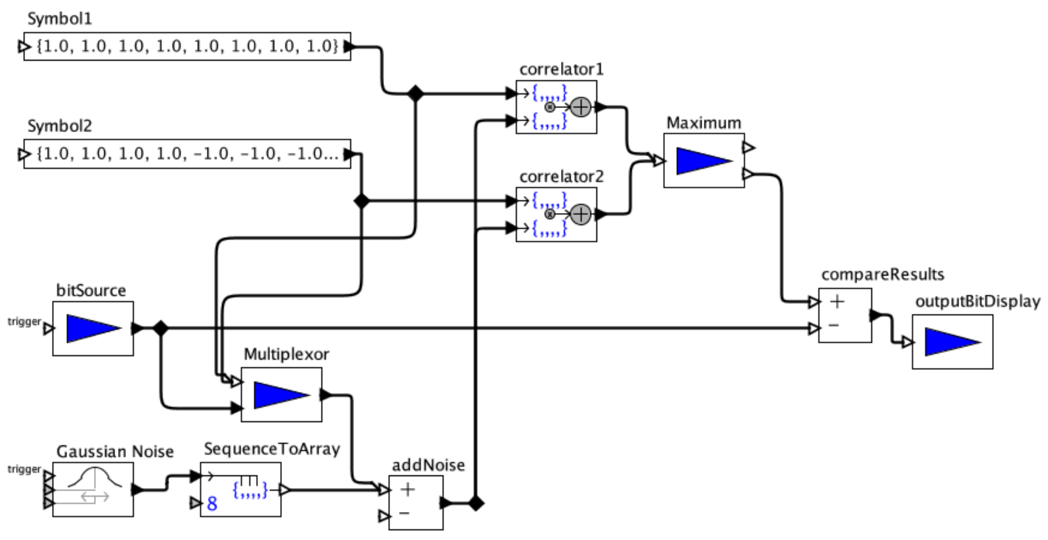


(a) Original Ptolemy diagram

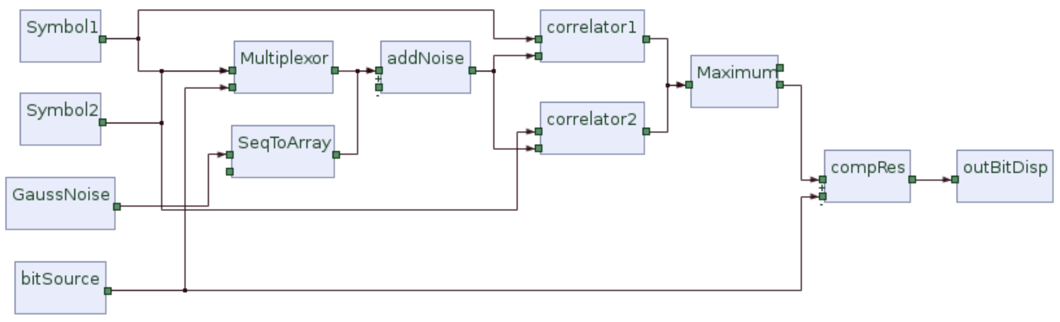


(b) Hierarchical layout

Figure 7.9: Comparison Example 5. Computation of prime numbers using the sieve of Eratosthenes, from a Ptolemy demonstration

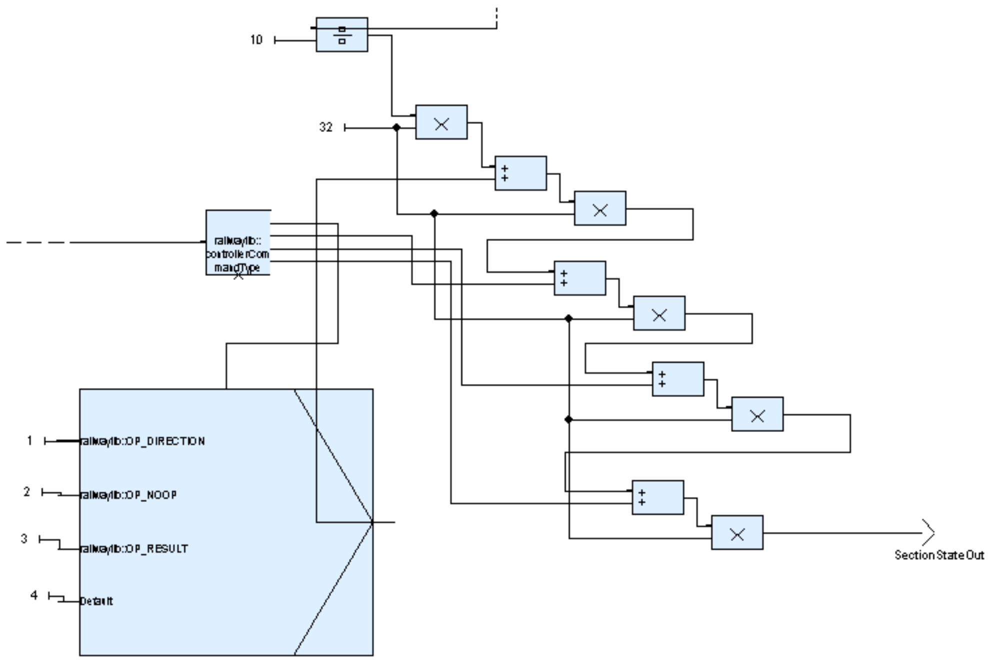


(a) Original Ptolemy diagram

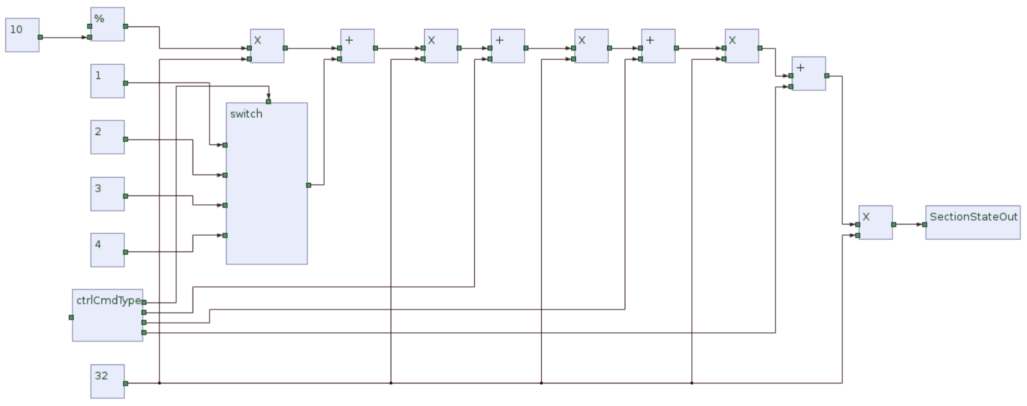


(b) Hierarchical layout

Figure 7.10: Comparison Example 6. Synchronous dataflow model for orthogonal communication, from a Ptolemy demonstration



(a) Original SCADE diagram



(b) Hierarchical layout

Figure 7.11: Comparison Example 7. Extract of a controller for a model railway driven by a time-triggered network, from a practical course on model-based design

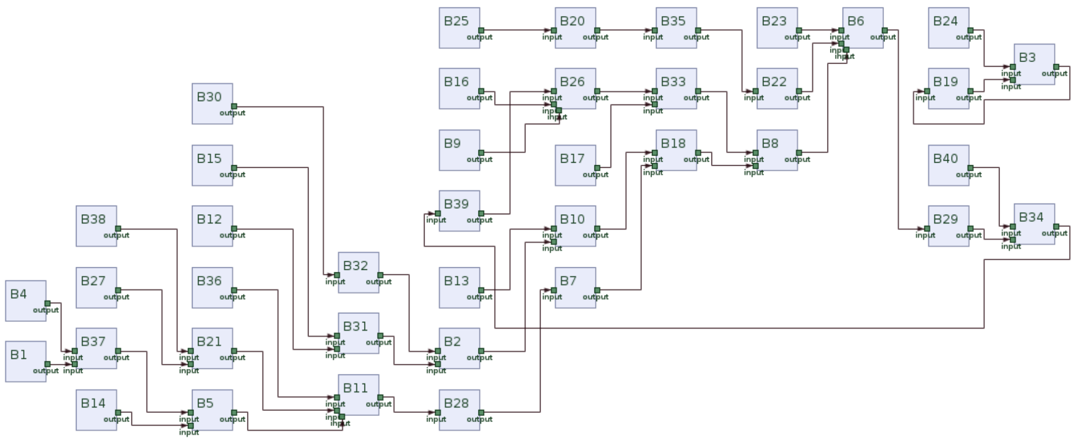


Figure 7.12: Hierarchical layout of a diagram created using a random model generator, with 30 vertices and one outgoing edge per vertex

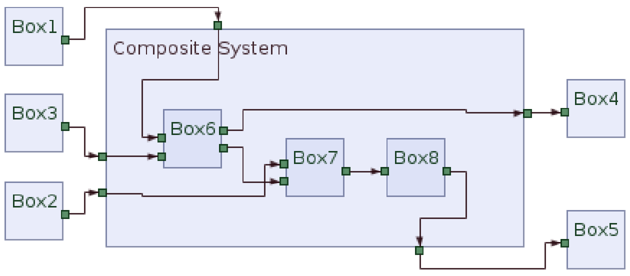
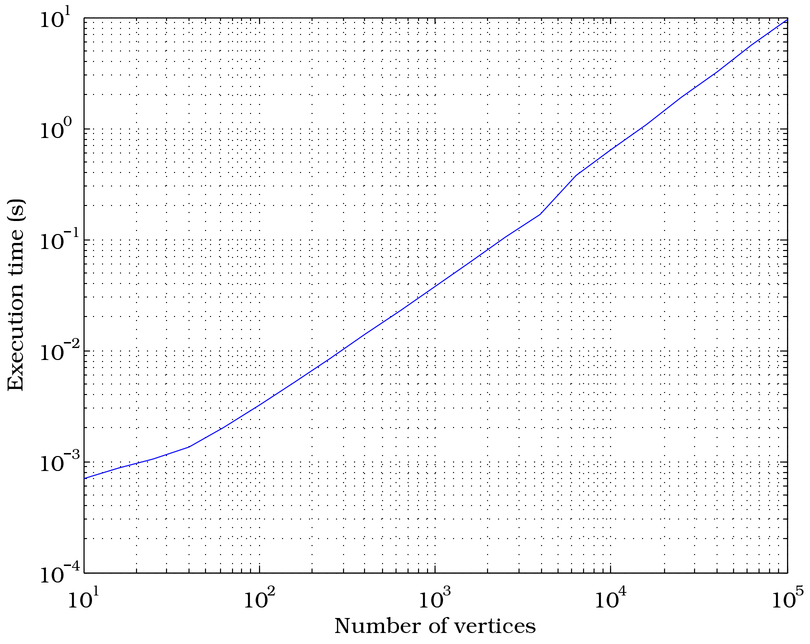
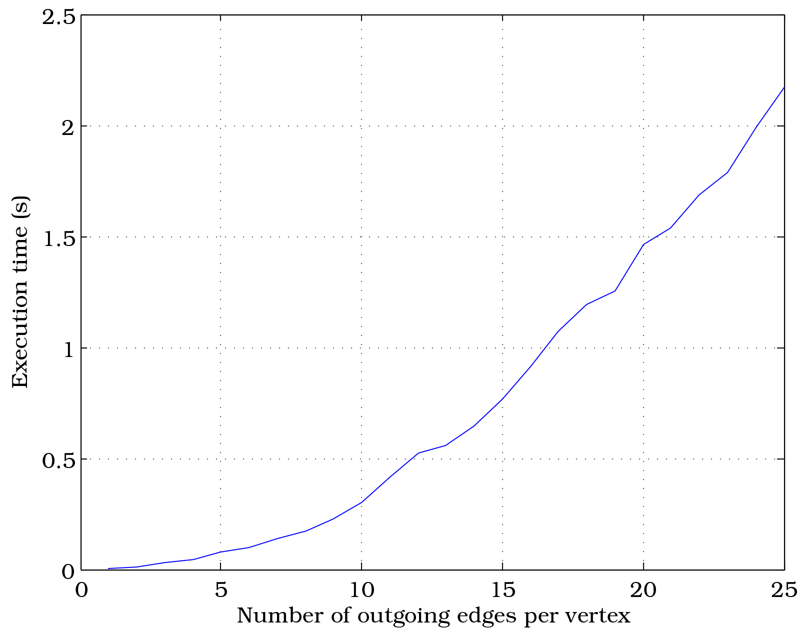


Figure 7.13: Hierarchical layout implementing edge routing to external ports



(a) Varying number of vertices, with one outgoing edge per vertex



(b) Varying number of outgoing edges per vertex for 100 vertices

Figure 7.14: Execution times of hierarchical layout

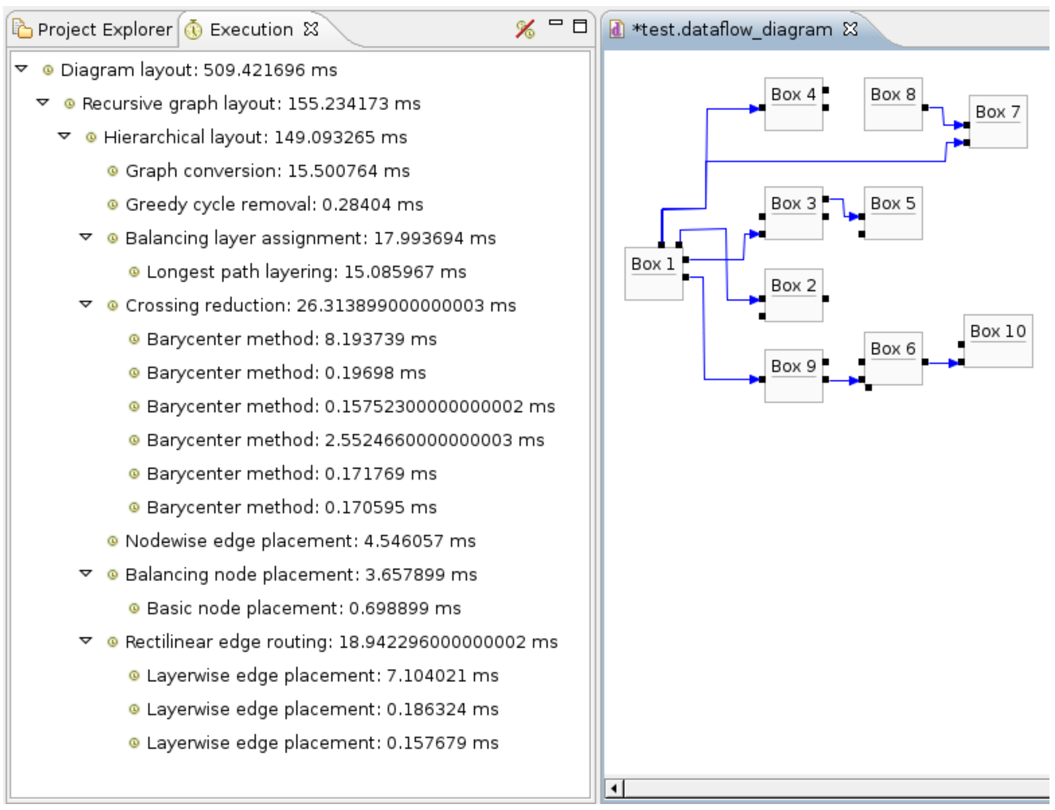
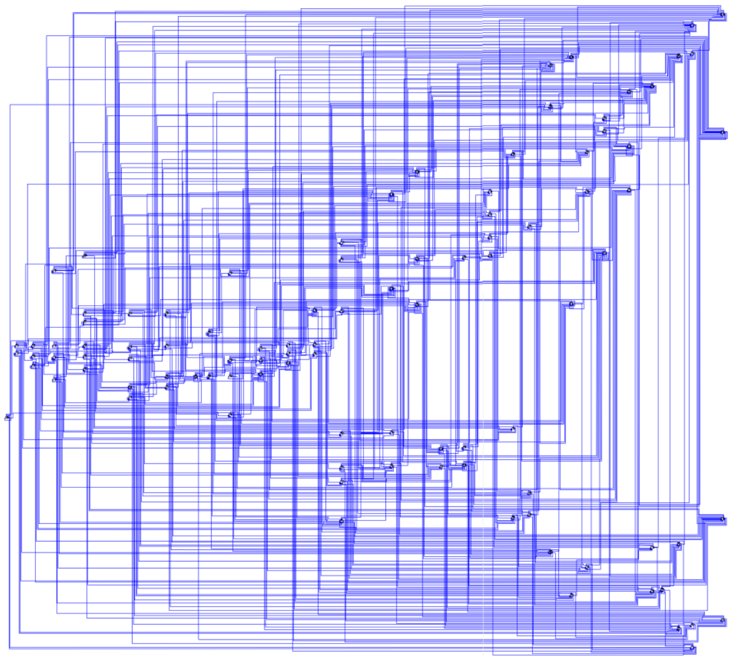


Diagram layout: 509.421696 ms
Recursive graph layout: 155.234173 ms
Hierarchical layout: 149.093265 ms
Graph conversion: 15.500764 ms
Greedy cycle removal: 0.28404 ms
Balancing layer assignment: 17.993694 ms
Longest path layering: 15.085967 ms
Crossing reduction: 26.313899000000003 ms
Barycenter method: 8.193739 ms
Barycenter method: 0.19698 ms
Barycenter method: 0.15752300000000002 ms
Barycenter method: 2.5524660000000003 ms
Barycenter method: 0.171769 ms
Barycenter method: 0.170595 ms
Nodewise edge placement: 4.546057 ms
Balancing node placement: 3.657899 ms
Basic node placement: 0.698899 ms
Rectilinear edge routing: 18.942296000000002 ms
Layerwise edge placement: 7.104021 ms
Layerwise edge placement: 0.186324 ms
Layerwise edge placement: 0.157679 ms

Figure 7.15: Execution times for each single algorithm phase on a graph with ten vertices, distributed to four layers (screenshot)


(a) Overview of the drawing

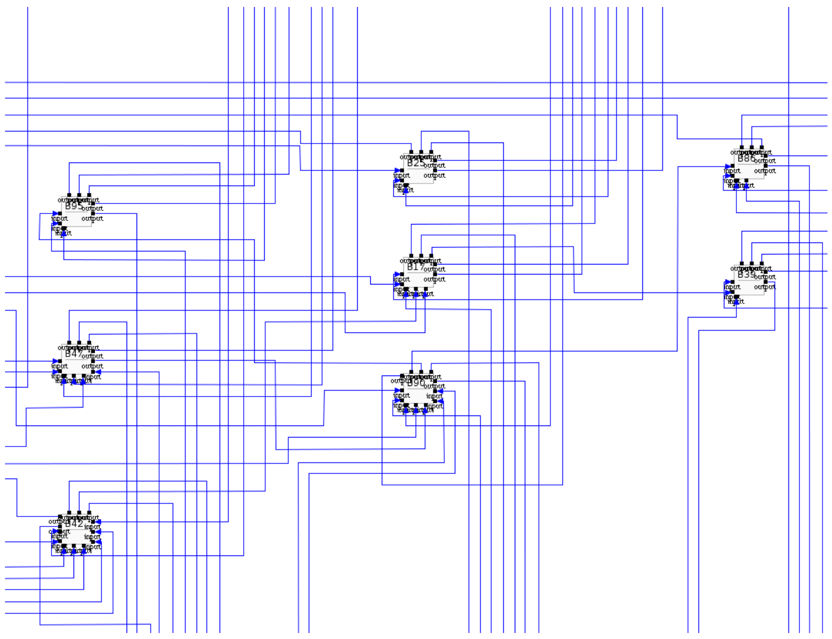

(b) Close-up view of a small part

Figure 7.16: Hierarchical layout of a graph with 100 vertices and 500 edges

8 Conclusion

This thesis introduces the problem of graph drawing for port based graphs with its special aspects of port constraints, hyperedges, and hierarchy with external ports. Two methods are discussed with regard to this problem: hierarchical layout, using the layered approach, and orthogonal layout, using the topology-shape-metrics approach. The former could be implemented successfully, while the latter requires further investigation to obtain a proper implementation.

The layered approach is extended by algorithms for crossing reduction with different scenarios of port constraints, handling of hyperedges that connect multiple ports, and a new approach for displaying model hierarchy. The effectiveness of these new concepts is demonstrated by the outputs of the implementation, which yields acceptable results and is very fast if the average vertex degree is not too high. Thus the hierarchical layout method appears very suitable for extension and adaptation, and its implementation can be used as a solid basis for further improvements. The strategy design pattern, which was consequently realized in the implementation, allows simple substitution of single modules. Hierarchical layout is especially effective for sparse graphs, *i.e.* graphs with few edges, but an increasing number of edges per vertex leads to confusing drawings and long running times. The main emphasis of this algorithm lies on the direction of flow, which is consequently arranged in one direction as much as possible.

Although the experimental results of the hierarchical and orthogonal layout methods cannot be directly compared yet, the topology-shape-metrics approach has some theoretical advantages over the layered approach. Since its first phase, planarization, has the goal of minimizing the global number of edge crossings, the resulting drawings are likely to become more readable than the output of hierarchical layout, which suffers from a large number of crossings in many cases. Furthermore, methods that are based on planarity have very strong theoretical backgrounds, since the topics of planarity and planarization have gained intensive research during the past decades, whereas the layered approach is built largely on applicatory heuristics that are difficult to examine mathematically. One should therefore expect the topology-shape-metrics approach to yield satisfying results after more resarch about its application to data flow diagrams and more work on its implementation. Promising concepts are those of upward planarization, which can be used to add direction of edges to this method, and of embedding constraints, which were shown to be suitable for representation of the scenarios of port constraints in data flow diagrams.

8.1 Future Work

The following matters can be analyzed to improve the hierarchical layout method:

- The balanced layering algorithm can be extended to more special cases. For example, the two FBY operators on the bottom right of Figure 7.6(b) and the two connected boxes PointReset and null_speed could be moved three layers to the left, thus shortening the incident connections.
- The implementation of the crossing reduction phase can be optimized, for in some cases it leads to obviously suboptimal vertex orderings, as with the automode box in Figure 7.6(b), which would have a better placement on top of the or box.
- The ordering of edges for routing around vertices needs improvement. In Figure 7.6(b), the edges that are routed around the ExternalConditions box on the left have a quite confusing order; two crossings could be eliminated by choosing a different edge placement.
- The whole layout algorithm could be extended for better handling of the FreePorts scenario. Some edge bends can be eliminated by moving the corresponding ports.
- Operators for which all outgoing edges were reversed for cycle removal could be reversed to emphasize their role as part of the cycles and possibly obtain a more compact drawing. Prior to the implementation of this feature it should be evaluated how such layouts are perceived.
- Diagrams such as the one in Figure 7.11(b) would probably benefit from the possibility to place vertices on top of each other that would normally be arranged in a row. As this would require to allow connections between vertices of the same layer, a simpler approach could be to reverse some of the edges in such a row (see the X and + operators in Figure 7.11(b)) in order to break the linear dependencies and allow a more compact layering.
- A vertex that is very broad forces the layer to which it is assigned to be broad as well. Currently all other vertices are aligned in the center of their layer, which is not good for sources and sinks. This should be extended so that left and right alignment is also possible. Another idea would be to stretch broad vertices over multiple layers.
- The edge routing to external ports can be improved to minimize the number of crossings.

For orthogonal layout the following issues require further research:

- The new approaches of planarization with embedding constraints [31] and upward planarization [9] should be merged to support both concepts.

- Many algorithms for planarization need an implementation of SPQR trees, which is quite complex [33]. The free OGDF library contains such an implementation, but is written in C++. It should be analyzed whether connecting to OGDF via Java Native Interface (JNI) or writing a new implementation would be better with regard to implementation time and effectiveness.
- A variant of GIOTTO orthogonalization should be implemented and compared with the Kandinsky implementation. The latter must be reviewed and checked for errors.
- Different compaction methods should be evaluated and implemented.
- Hyperedges and external ports can be considered for integration into the orthogonal layout method, introducing new constraints and extensions for each layout phase.

Furthermore, other graph drawing methods besides the layered approach and the topology-shape-metrics approach could be analyzed and extended to handle the specialties of data flow diagrams.

Bibliography

[1] Anjali Arya, Anshul Kumar, V. V. Swaminathan, and Amit Misra. Automatic generation of digital system schematic diagrams. In *DAC '85: Proceedings of the 22nd ACM/IEEE Conference on Design Automation*, pages 388–395, New York, NY, USA, 1985. ACM.

[2] Danil E. Baburin. Using graph based representations in reengineering. *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 203–206, 2002.

[3] Brenda S. Baker, Sandeep N. Bhatt, and Frank Thomson Leighton. An approximation algorithm for manhattan routing. In *STOC '83: Proceedings of the Fifteenth Annual ACM Symposium on Theory of Computing*, pages 477–486, New York, NY, USA, 1983. ACM.

[4] Wilhelm Barth, Petra Mutzel, and Canan Yildiz. A new approximation algorithm for bend minimization in the Kandinsky model. In Michael Kaufmann and Dorothea Wagner, editors, *GD 2006: 14th International Symposium on Graph Drawing*, volume 4372 of *Lecture notes in Computer Science*, pages 343–354. Springer-Verlag, 2007.

[5] Carlo Batini, Enrico Nardelli, and Roberto Tamassia. A layout algorithm for data flow diagrams. *IEEE Transactions on Software Engineering*, 12(4):538–546, 1986.

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

[7] Ned Chapin. Some structured analysis techniques. *SIGMIS Database*, 10(3):16–23, 1978.

[8] Norishige Chiba, Kazunori Onoguchi, and Takao Nishizeki. Drawing plane graphs nicely. *Acta Informatica*, 22(2):187–201, 1985.

[9] Markus Chimani, Carsten Gutwenger, Petra Mutzel, and Hoi-Ming Wong. Layer-free upward crossing minimization. In *WEA 2008: Proceedings of the 7th International Workshop on Experimental Algorithms*, volume 5038 of *Lecture Notes in Computer Science*, pages 55–68. Springer-Verlag, 2008.

[10] Alan L. Davis and Robert M. Keller. Data flow program graphs. *Computer*, 15(2):26–41, Feb 1982.

[11] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, June 1994.

[12] Walter Didimo. Upward planar drawings and switch-regularity heuristics. *Journal of Graph Algorithms and Applications*, 10(2):259–285, 2006.

[13] Michael Doorley and Anthony Cahill. Experiences in automatic leveling of data flow diagrams. In *WPC '96: Proceedings of the 4th International Workshop on Program Comprehension*, pages 218–229, Washington, DC, USA, 1996. IEEE Computer Society.

[14] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

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

[16] Markus Eiglsperger. Constraints im Kandinsky-Algorithmus. Diploma thesis, Wilhelm-Schickard-Institut für Informatik, Eberhard-Karls-Universität Tübingen, August 1999.

[17] Markus Eiglsperger. *Automatic Layout of UML Class Diagrams: A Topology-Shape-Metrics Approach.* PhD thesis, Faculty of Information and Cognitive Science, Eberhard-Karls-Universität Tübingen, 2003.

[18] Markus Eiglsperger, Ulrich Fößmeier, and Michael Kaufmann. Orthogonal graph drawing with constraints. In *SODA '00: Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 3–11, Philadelphia, PA, USA, 2000. Society for Industrial and Applied Mathematics.

[19] Markus Eiglsperger and Michael Kaufmann. Fast compaction for orthogonal drawings with vertices of prescribed size. In *GD '01: Revised Papers from the 9th International Symposium on Graph Drawing*, pages 124–138, London, UK, 2002. Springer-Verlag.

[20] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of Sugiyama's algorithm for layered graph drawing. In János Pach, editor, *GD 2004: 12th International Symposium on Graph Drawing*, volume 3383 of *Lecture Notes in Computer Science*, pages 155–166. Springer-Verlag, 2004.

[21] Ulrich Fößmeier and Michael Kaufmann. Drawing high degree graphs with low bend numbers. In *GD '95: Proceedings of the Symposium on Graph Drawing*, pages 254–266, London, UK, 1996. Springer-Verlag.

[22] Ulrich Fößmeier and Michael Kaufmann. Algorithms and area bounds for nonplanar orthogonal drawings. In Giuseppe Di Battista, editor, *GD '97: Proceedings of the 5th International Symposium on Graph Drawing*, pages 134–145, London, UK, 1997. Springer-Verlag.

[23] H. De Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990.

[24] Hauke Fuhrmann and Reinhard von Hanxleden. Enhancing graphical model-based system design—an avionics case study. Technical Report 0901, Christian-Albrechts-Universität Kiel, Department of Computer Science, January 2009.

[25] 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.

[26] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1234, 2000.

[27] 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.

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

[29] Ashim Garg and Roberto Tamassia. On the computational complexity of upward and rectilinear planarity testing. In *GD '94: DIMACS International Workshop on Graph Drawing*, volume 894 of *Lecture Notes in Computer Science*, pages 286–297. Springer-Verlag, 1995.

[30] Carsten Gutwenger, Michael Jünger, Karsten Klein, Joachim Kupke, Sebastian Leipert, and Petra Mutzel. A new approach for visualizing UML class diagrams. In *SoftVis '03: Proceedings of the 2003 ACM Symposium on Software Visualization*, pages 179–188, New York, NY, USA, 2003. ACM.

[31] Carsten Gutwenger, Karsten Klein, and Petra Mutzel. Planarity testing and optimal edge insertion with embedding constraints. In Michael Kaufmann and Dorothea Wagner, editors, *GD 2006: Proceedings of the 14th International Symposium on Graph Drawing*, volume 4372 of *Lecture notes in Computer Science*, pages 126–137. Springer-Verlag, 2007.

[32] Carsten Gutwenger and Petra Mutzel. Planar polyline drawings with good angular resolution. In *GD '98: Proceedings of the 6th International Symposium on Graph Drawing*, volume 1547 of *Lecture Notes in Computer Science*, pages 167–182. Springer-Verlag, 1998.

[33] Carsten Gutwenger and Petra Mutzel. A linear time implementation of SPQR-trees. In *GD '00: Proceedings of the 8th International Symposium on Graph Drawing*, volume 1984 of *Lecture notes in Computer Science*, pages 77–90, London, UK, 2001. Springer-Verlag.

[34] Carsten Gutwenger, Petra Mutzel, and René Weiskircher. Inserting an edge into a planar graph. In *SODA '01: Proceedings of the Twelfth Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 246–255, Philadelphia, PA, USA, 2001. Society for Industrial and Applied Mathematics.

[35] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.

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

[37] Jiang Hu and Sachin S. Sapatnekar. A survey on multi-net global routing for integrated circuits. *Integration, the VLSI Journal*, 31(1):1–49, 2001.

[38] Michael D. Hutton and Anna Lubiw. Upward planar drawing of single source acyclic digraphs. In *SODA '91: Proceedings of the Second Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 203–211, Philadelphia, PA, USA, 1991. Society for Industrial and Applied Mathematics.

[39] Michael Jünger and Petra Mutzel. *Graph Drawing Software.* Springer, October 2003.

[40] 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.

[41] Goos Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16(1):4–32, 1996.

[42] Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs: Methods and Models.* Number 2025 in Lecture Notes in Computer Science (LNCS). Springer-Verlag, Berlin, Germany, 2001. `http://link.springer.de/link/service/series/0558/tocs/t2025.htm`.

[43] Gunnar W. Klau and Petra Mutzel. Quasi-orthogonal drawing of planar graphs. Technical Report MPI-I-98-1-013, Max-Planck-Institut für Informatik, Saarbrücken, 1998.

[44] Gunnar W. Klau and Petra Mutzel. Optimal compaction of orthogonal grid drawings. In *Proceedings of the 7th International IPCO Conference on Integer Programming and Combinatorial Optimization*, pages 304–319, London, UK, 1999. Springer-Verlag.

[45] C. R. Lageweg. Designing an automatic schematic generator for a netlist description. Technical Report 1-68340-44(1998)03, Laboratory of Computer Architecture and Digital Techniques (CARDIT), Delft University of Technology, Faculty of Information Technology and Systems, 1998.

[46] Edward A. Lee. Overview of the Ptolemy project. Technical Memorandum UCB/ERL M03/25, University of California, Berkeley, CA, 94720, USA, July 2003. `http://ptolemy.eecs.berkeley.edu/publications/papers/03/overview`.

[47] 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.

[48] Petra Mutzel. The SPQR-tree data structure in graph drawing. In *Automata, Languages and Programming, 30th International Colloquium*, volume 2719 of *Lecture Notes In Computer Science*, pages 34–46. Springer-Verlag, 2003.

[49] Maurizio Patrignani. On the complexity of orthogonal compaction. In *WADS '99: Proceedings of the 6th International Workshop on Algorithms and Data Structures*, pages 56–61, London, UK, 1999. Springer-Verlag.

[50] Steffen Prochnow and Reinhard von Hanxleden. Statechart development beyond WYSIWYG. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, Nashville, TN, USA, October 2007.

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

[52] Georg Sander. A fast heuristic for hierarchical manhattan layout. In *GD '95: Proceedings of the Symposium on Graph Drawing*, pages 447–458, London, UK, 1996. Springer-Verlag.

[53] Arne Schipper. Layout and Visual Comparison of Statecharts. Diploma thesis, Christian-Albrechts-Universität zu Kiel, December 2008. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ars-dt.pdf`.

[54] Walter Schnyder. Embedding planar graphs on the grid. In *SODA '90: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 138–148, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

[55] K. Shahookar and P. Mazumder. VLSI cell placement techniques. *ACM Computing Surveys*, 23(2):143–220, 1991.

[56] 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.

[57] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal of Computing*, 16(3):421–444, 1987.

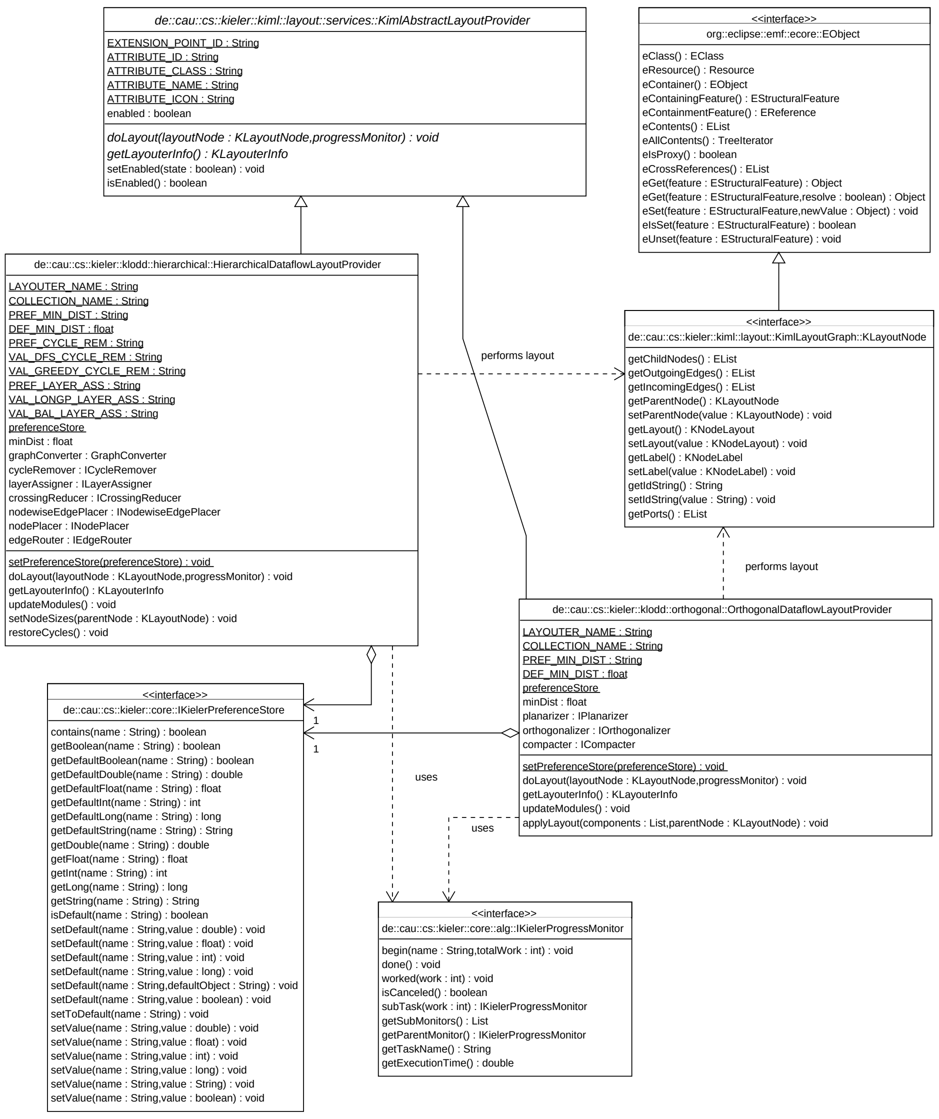
[58] Roberto Tamassia. Constraints in graph drawing algorithms. *Constraints*, 3(1):87–120, 1998.

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

[60] Roberto Tamassia and Ioannis G. Tollis. A unified approach to visibility representations of planar graphs. *Discrete and Computational Geometry*, 1(1):321–341, 1986.

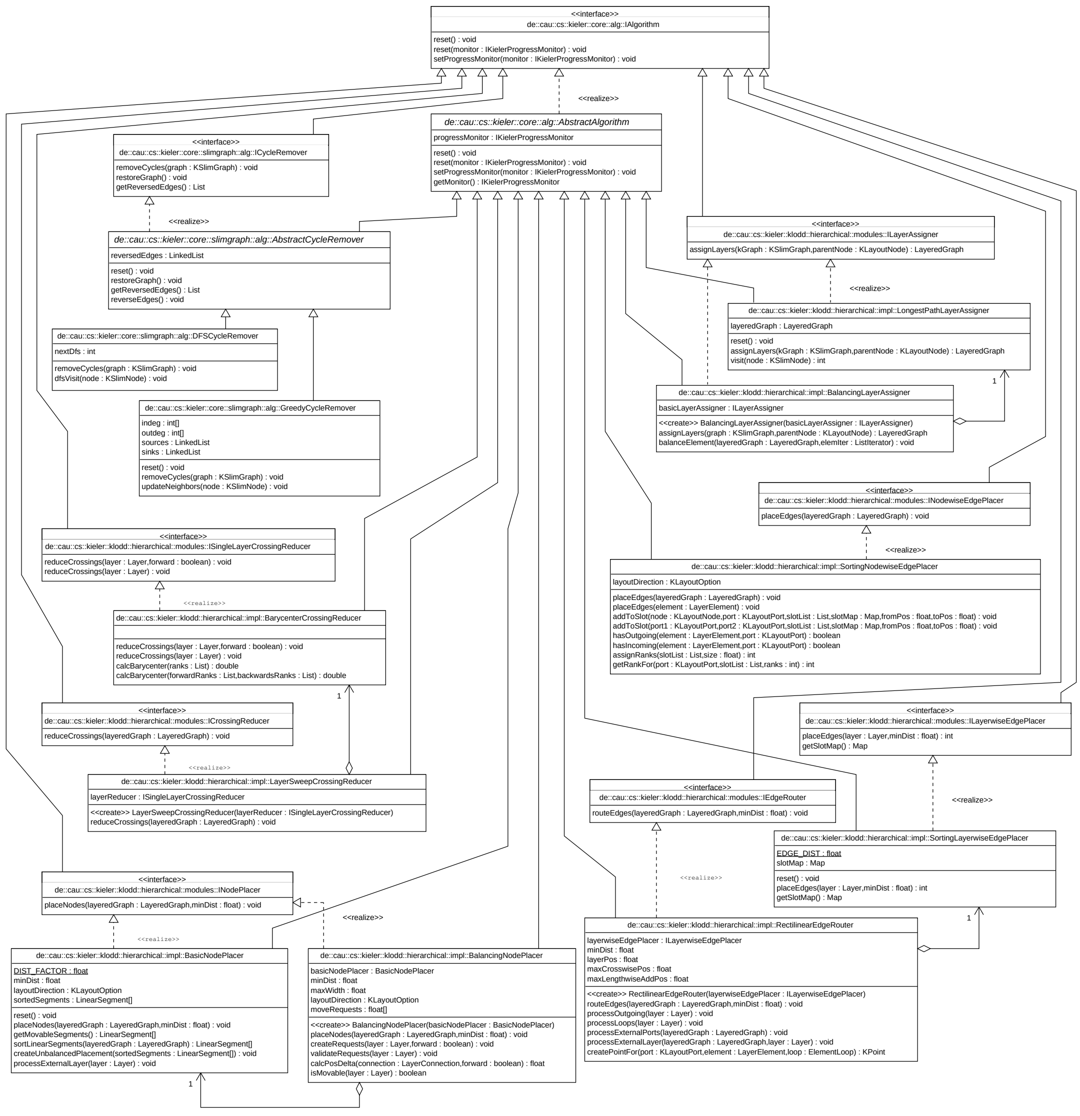
[61] Charlotte Wieners-Lummer. Manhattan channel routing with good theoretical and practical performance. In *SODA '90: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 465–474, Philadelphia, PA, USA, 1990. Society for Industrial and Applied Mathematics.

Appendix A: Class Diagrams



de::cau::cs::kieler::kiml::layout::services::KimlAbstractLayoutProvider
EXTENSION_POINT_ID : String
ATTRIBUTE_ID : String
ATTRIBUTE_CLASS : String
ATTRIBUTE_NAME : String
ATTRIBUTE_ICON : String
enabled : boolean
doLayout(layoutNode : KLayoutNode,progressMonitor) : void
getLayouterInfo() : KLayouterInfo
setEnabled(state : boolean) : void
isEnabled() : boolean
<<interface>>
org::eclipse::emf::ecore::EObject
eClass() : EClass
eResource() : Resource
eContainer() : EObject
eContainingFeature() : EStructuralFeature
eContainmentFeature() : EReference
eContents() : EList
eAllContents() : TreeIterator
eIsProxy() : boolean
eCrossReferences() : EList
eGet(feature : EStructuralFeature) : Object
eGet(feature : EStructuralFeature,resolve : boolean) : Object
eSet(feature : EStructuralFeature,newValue : Object) : void
eIsSet(feature : EStructuralFeature) : boolean
eUnset(feature : EStructuralFeature) : void
de::cau::cs::kieler::klodd::hierarchical::HierarchicalDataflowLayoutProvider
LAYOUTER_NAME : String
COLLECTION_NAME : String
PREF_MIN_DIST : String
DEF_MIN_DIST : float
PREF_CYCLE_REM : String
VAL_DFS_CYCLE_REM : String
VAL_GREEDY_CYCLE_REM : String
PREF_LAYER_ASS : String
VAL_LONGP_LAYER_ASS : String
VAL_BAL_LAYER_ASS : String
preferenceStore
minDist : float
graphConverter : GraphConverter
cycleRemover : ICycleRemover
layerAssigner : ILayerAssigner
crossingReducer : ICrossingReducer
nodewiseEdgePlacer : INodewiseEdgePlacer
nodePlacer : INodePlacer
edgeRouter : IEdgeRouter
setPreferenceStore(preferenceStore) : void
doLayout(layoutNode : KLayoutNode,progressMonitor) : void
getLayouterInfo() : KLayouterInfo
updateModules() : void
setNodeSizes(parentNode : KLayoutNode) : void
restoreCycles() : void
performs layout
<<interface>>
de::cau::cs::kieler::kiml::layout::KimlLayoutGraph::KLayoutNode
getChildNodes() : EList
getOutgoingEdges() : EList
getIncomingEdges() : EList
getParentNode() : KLayoutNode
setParentNode(value : KLayoutNode) : void
getLayout() : KNodeLayout
setLayout(value : KNodeLayout) : void
getLabel() : KNodeLabel
setLabel(value : KNodeLabel) : void
getIdString() : String
setIdString(value : String) : void
getPorts() : EList
performs layout
de::cau::cs::kieler::klodd::orthogonal::OrthogonalDataflowLayoutProvider
LAYOUTER_NAME : String
COLLECTION_NAME : String
PREF_MIN_DIST : String
DEF_MIN_DIST : float
preferenceStore
minDist : float
planarizer : IPlanarizer
orthogonalizer : IOrthogonalizer
compacter : ICompacter
setPreferenceStore(preferenceStore) : void
doLayout(layoutNode : KLayoutNode,progressMonitor) : void
getLayouterInfo() : KLayouterInfo
updateModules() : void
applyLayout(components : List,parentNode : KLayoutNode) : void
<<interface>>
de::cau::cs::kieler::core::IKielerPreferenceStore
1
1
contains(name : String) : boolean
getBoolean(name : String) : boolean
getDefaultBoolean(name : String) : boolean
getDefaultDouble(name : String) : double
getDefaultFloat(name : String) : float
getDefaultInt(name : String) : int
getDefaultLong(name : String) : long
getDefaultString(name : String) : String
getDouble(name : String) : double
getFloat(name : String) : float
getInt(name : String) : int
getLong(name : String) : long
getString(name : String) : String
isDefault(name : String) : boolean
setDefault(name : String,value : double) : void
setDefault(name : String,value : float) : void
setDefault(name : String,value : int) : void
setDefault(name : String,value : long) : void
setDefault(name : String,defaultObject : String) : void
setDefault(name : String,value : boolean) : void
setToDefault(name : String) : void
setValue(name : String,value : double) : void
setValue(name : String,value : float) : void
setValue(name : String,value : int) : void
setValue(name : String,value : long) : void
setValue(name : String,value : String) : void
setValue(name : String,value : boolean) : void
uses
uses
<<interface>>
de::cau::cs::kieler::core::alg::IKielerProgressMonitor
begin(name : String,totalWork : int) : void
done() : void
worked(work : int) : void
isCanceled() : boolean
subTask(work : int) : IKielerProgressMonitor
getSubMonitors() : List
getParentMonitor() : IKielerProgressMonitor
getTaskName() : String
getExecutionTime() : double

Figure A.1: Main classes for the layout API, including layout providers for hierarchical layout and orthogonal layout



<<interface>>
de::cau::cs::kieler::core::alg::IAlgorithm
reset() : void
reset(monitor : IKielerProgressMonitor) : void
setProgressMonitor(monitor : IKielerProgressMonitor) : void
<<realize>>
de::cau::cs::kieler::core::alg::AbstractAlgorithm
progressMonitor : IKielerProgressMonitor
reset() : void
reset(monitor : IKielerProgressMonitor) : void
setProgressMonitor(monitor : IKielerProgressMonitor) : void
getMonitor() : IKielerProgressMonitor
<<interface>>
de::cau::cs::kieler::core::slimgraph::alg::ICycleRemover
removeCycles(graph : KSlimGraph) : void
restoreGraph() : void
getReversedEdges() : List
<<realize>>
de::cau::cs::kieler::core::slimgraph::alg::AbstractCycleRemover
reversedEdges : LinkedList
reset() : void
restoreGraph() : void
getReversedEdges() : List
reverseEdges() : void
de::cau::cs::kieler::core::slimgraph::alg::DFSCycleRemover
nextDfs : int
removeCycles(graph : KSlimGraph) : void
dfsVisit(node : KSlimNode) : void
de::cau::cs::kieler::core::slimgraph::alg::GreedyCycleRemover
indeg : int[]
outdeg : int[]
sources : LinkedList
sinks : LinkedList
reset() : void
removeCycles(graph : KSlimGraph) : void
updateNeighbors(node : KSlimNode) : void
<<interface>>
de::cau::cs::kieler::klodd::hierarchical::modules::ILayerAssigner
assignLayers(kGraph : KSlimGraph,parentNode : KLayoutNode) : LayeredGraph
<<realize>>
de::cau::cs::kieler::klodd::hierarchical::impl::LongestPathLayerAssigner
layeredGraph : LayeredGraph
reset() : void
assignLayers(kGraph : KSlimGraph,parentNode : KLayoutNode) : LayeredGraph
visit(node : KSlimNode) : int
1
de::cau::cs::kieler::klodd::hierarchical::impl::BalancingLayerAssigner
basicLayerAssigner : ILayerAssigner
<<create>> BalancingLayerAssigner(basicLayerAssigner : ILayerAssigner)
assignLayers(graph : KSlimGraph,parentNode : KLayoutNode) : LayeredGraph
balanceElement(layeredGraph : LayeredGraph,elemIter : ListIterator) : void
<<interface>>
de::cau::cs::kieler::klodd::hierarchical::modules::INodewiseEdgePlacer
placeEdges(layeredGraph : LayeredGraph) : void
<<realize>>
de::cau::cs::kieler::klodd::hierarchical::impl::SortingNodewiseEdgePlacer
layoutDirection : KLayoutOption
placeEdges(layeredGraph : LayeredGraph) : void
placeEdges(element : LayerElement) : void
addToSlot(node : KLayoutNode,port : KLayoutPort,slotList : List,slotMap : Map,fromPos : float,toPos : float) : void
addToSlot(port1 : KLayoutPort,port2 : KLayoutPort,slotList : List,slotMap : Map,fromPos : float,toPos : float) : void
hasOutgoing(element : LayerElement,port : KLayoutPort) : boolean
hasIncoming(element : LayerElement,port : KLayoutPort) : boolean
assignRanks(slotList : List,size : float) : int
getRankFor(port : KLayoutPort,slotList : List,ranks : int) : int
<<interface>>
de::cau::cs::kieler::klodd::hierarchical::modules::ISingleLayerCrossingReducer
reduceCrossings(layer : Layer,forward : boolean) : void
reduceCrossings(layer : Layer) : void
<<realize>>
de::cau::cs::kieler::klodd::hierarchical::impl::BarycenterCrossingReducer
reduceCrossings(layer : Layer,forward : boolean) : void
reduceCrossings(layer : Layer) : void
calcBarycenter(ranks : List) : double
calcBarycenter(forwardRanks : List,backwardsRanks : List) : double
1
<<interface>>
de::cau::cs::kieler::klodd::hierarchical::modules::ICrossingReducer
reduceCrossings(layeredGraph : LayeredGraph) : void
<<realize>>
de::cau::cs::kieler::klodd::hierarchical::impl::LayerSweepCrossingReducer
layerReducer : ISingleLayerCrossingReducer
<<create>> LayerSweepCrossingReducer(layerReducer : ISingleLayerCrossingReducer)
reduceCrossings(layeredGraph : LayeredGraph) : void
<<interface>>
de::cau::cs::kieler::klodd::hierarchical::modules::ILayerwiseEdgePlacer
placeEdges(layer : Layer,minDist : float) : int
getSlotMap() : Map
<<realize>>
de::cau::cs::kieler::klodd::hierarchical::impl::SortingLayerwiseEdgePlacer
EDGE_DIST : float
slotMap : Map
reset() : void
placeEdges(layer : Layer,minDist : float) : int
getSlotMap() : Map
1
<<interface>>
de::cau::cs::kieler::klodd::hierarchical::modules::IEdgeRouter
routeEdges(layeredGraph : LayeredGraph,minDist : float) : void
<<realize>>
de::cau::cs::kieler::klodd::hierarchical::impl::RectilinearEdgeRouter
layerwiseEdgePlacer : ILayerwiseEdgePlacer
minDist : float
layerPos : float
maxCrosswisePos : float
maxLengthwiseAddPos : float
<<create>> RectilinearEdgeRouter(layerwiseEdgePlacer : ILayerwiseEdgePlacer)
routeEdges(layeredGraph : LayeredGraph,minDist : float) : void
processOutgoing(layer : Layer) : void
processLoops(layer : Layer) : void
processExternalPorts(layeredGraph : LayeredGraph) : void
processExternalLayer(layeredGraph : LayeredGraph,layer : Layer) : void
createPointFor(port : KLayoutPort,element : LayerElement,loop : ElementLoop) : KPoint
<<interface>>
de::cau::cs::kieler::klodd::hierarchical::modules::INodePlacer
placeNodes(layeredGraph : LayeredGraph,minDist : float) : void
<<realize>>
<<realize>>
de::cau::cs::kieler::klodd::hierarchical::impl::BasicNodePlacer
DIST_FACTOR : float
minDist : float
layoutDirection : KLayoutOption
sortedSegments : LinearSegment[]
reset() : void
placeNodes(layeredGraph : LayeredGraph,minDist : float) : void
getMovableSegments() : LinearSegment[]
sortLinearSegments(layeredGraph : LayeredGraph) : LinearSegment[]
createUnbalancedPlacement(sortedSegments : LinearSegment[]) : void
processExternalLayer(layer : Layer) : void
1
de::cau::cs::kieler::klodd::hierarchical::impl::BalancingNodePlacer
basicNodePlacer : BasicNodePlacer
minDist : float
maxWidth : float
layoutDirection : KLayoutOption
moveRequests : float[]
<<create>> BalancingNodePlacer(basicNodePlacer : BasicNodePlacer)
placeNodes(layeredGraph : LayeredGraph,minDist : float) : void
createRequests(layer : Layer,forward : boolean) : void
validateRequests(layer : Layer) : void
calcPosDelta(connection : LayerConnection,forward : boolean) : float
isMovable(layer : Layer) : boolean

Figure A.2: Modules for the hierarchical layout algorithm, all implemented as subclasses of AbstractAlgorithm

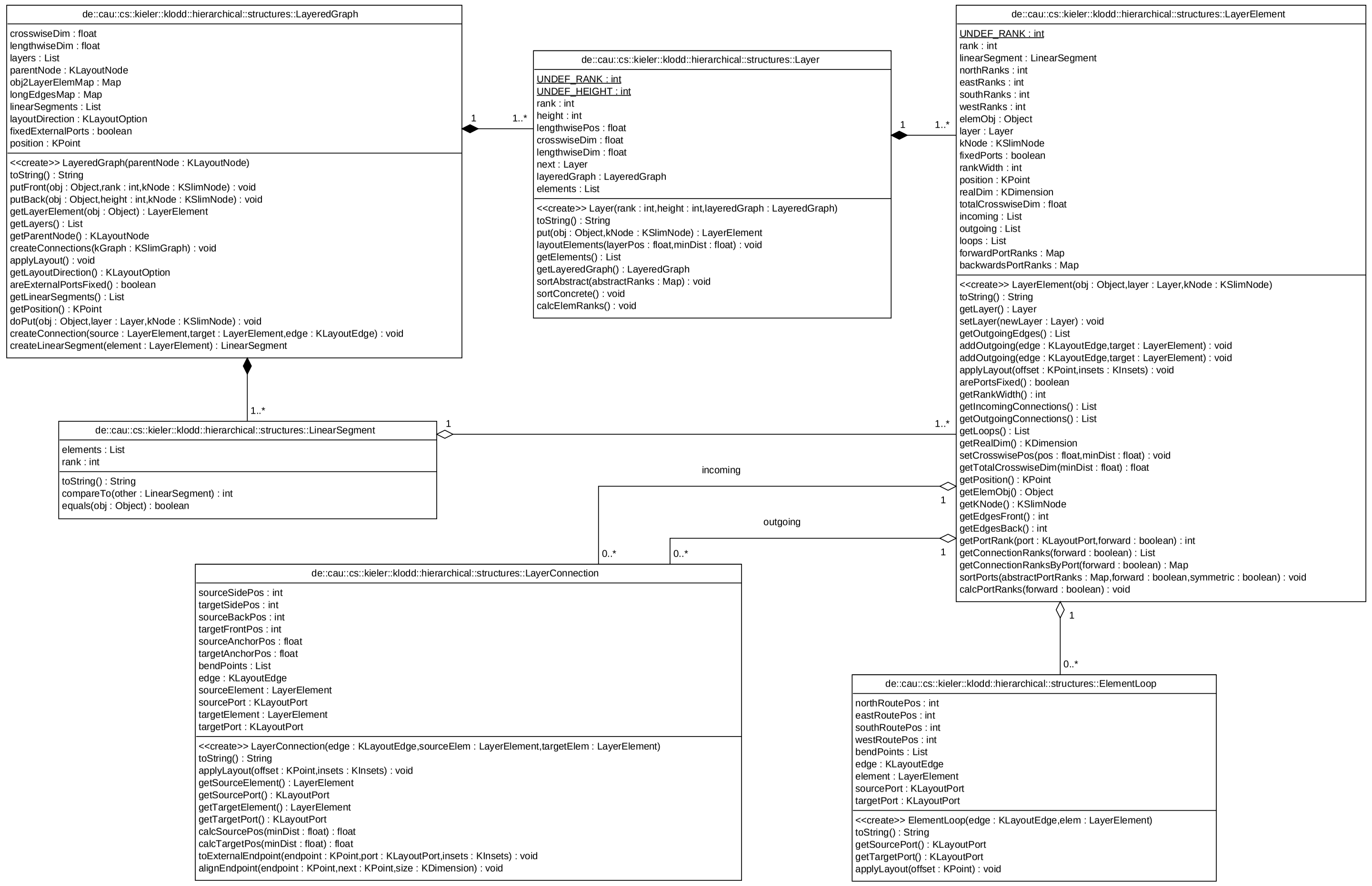

de::cau::cs::kieler::klodd::hierarchical::structures::LayeredGraph
crosswiseDim : float
lengthwiseDim : float
layers : List
parentNode : KLayoutNode
obj2LayerElemMap : Map
longEdgesMap : Map
linearSegments : List
layoutDirection : KLayoutOption
fixedExternalPorts : boolean
position : KPoint
<<create>> LayeredGraph(parentNode : KLayoutNode)
toString() : String
putFront(obj : Object,rank : int,kNode : KSlimNode) : void
putBack(obj : Object,height : int,kNode : KSlimNode) : void
getLayerElement(obj : Object) : LayerElement
getLayers() : List
getParentNode() : KLayoutNode
createConnections(kGraph : KSlimGraph) : void
applyLayout() : void
getLayoutDirection() : KLayoutOption
areExternalPortsFixed() : boolean
getLinearSegments() : List
getPosition() : KPoint
doPut(obj : Object,layer : Layer,kNode : KSlimNode) : void
createConnection(source : LayerElement,target : LayerElement,edge : KLayoutEdge) : void
createLinearSegment(element : LayerElement) : LinearSegment
1
1..*
de::cau::cs::kieler::klodd::hierarchical::structures::Layer
UNDEF_RANK : int
UNDEF_HEIGHT : int
rank : int
height : int
lengthwisePos : float
crosswiseDim : float
lengthwiseDim : float
next : Layer
layeredGraph : LayeredGraph
elements : List
<<create>> Layer(rank : int,height : int,layeredGraph : LayeredGraph)
toString() : String
put(obj : Object,kNode : KSlimNode) : LayerElement
layoutElements(layerPos : float,minDist : float) : void
getElements() : List
getLayeredGraph() : LayeredGraph
sortAbstract(abstractRanks : Map) : void
sortConcrete() : void
calcElemRanks() : void
1
1..*
de::cau::cs::kieler::klodd::hierarchical::structures::LayerElement
UNDEF_RANK : int
rank : int
linearSegment : LinearSegment
northRanks : int
eastRanks : int
southRanks : int
westRanks : int
elemObj : Object
layer : Layer
kNode : KSlimNode
fixedPorts : boolean
rankWidth : int
position : KPoint
realDim : KDimension
totalCrosswiseDim : float
incoming : List
outgoing : List
loops : List
forwardPortRanks : Map
backwardsPortRanks : Map
<<create>> LayerElement(obj : Object,layer : Layer,kNode : KSlimNode)
toString() : String
getLayer() : Layer
setLayer(newLayer : Layer) : void
getOutgoingEdges() : List
addOutgoing(edge : KLayoutEdge,target : LayerElement) : void
addOutgoing(edge : KLayoutEdge,target : LayerElement) : void
applyLayout(offset : KPoint,insets : KInsets) : void
arePortsFixed() : boolean
getRankWidth() : int
getIncomingConnections() : List
getOutgoingConnections() : List
getLoops() : List
getRealDim() : KDimension
setCrosswisePos(pos : float,minDist : float) : void
getTotalCrosswiseDim(minDist : float) : float
getPosition() : KPoint
getElemObj() : Object
getKNode() : KSlimNode
getEdgesFront() : int
getEdgesBack() : int
getPortRank(port : KLayoutPort,forward : boolean) : int
getConnectionRanks(forward : boolean) : List
getConnectionRanksByPort(forward : boolean) : Map
sortPorts(abstractPortRanks : Map,forward : boolean,symmetric : boolean) : void
calcPortRanks(forward : boolean) : void
1..*
de::cau::cs::kieler::klodd::hierarchical::structures::LinearSegment
elements : List
rank : int
toString() : String
compareTo(other : LinearSegment) : int
equals(obj : Object) : boolean
1
1..*
incoming
1
outgoing
1
0..*
0..*
de::cau::cs::kieler::klodd::hierarchical::structures::LayerConnection
sourceSidePos : int
targetSidePos : int
sourceBackPos : int
targetFrontPos : int
sourceAnchorPos : float
targetAnchorPos : float
bendPoints : List
edge : KLayoutEdge
sourceElement : LayerElement
sourcePort : KLayoutPort
targetElement : LayerElement
targetPort : KLayoutPort
<<create>> LayerConnection(edge : KLayoutEdge,sourceElem : LayerElement,targetElem : LayerElement)
toString() : String
applyLayout(offset : KPoint,insets : KInsets) : void
getSourceElement() : LayerElement
getSourcePort() : KLayoutPort
getTargetElement() : LayerElement
getTargetPort() : KLayoutPort
calcSourcePos(minDist : float) : float
calcTargetPos(minDist : float) : float
toExternalEndpoint(endpoint : KPoint,port : KLayoutPort,insets : KInsets) : void
alignEndpoint(endpoint : KPoint,next : KPoint,size : KDimension) : void
1
0..*
de::cau::cs::kieler::klodd::hierarchical::structures::ElementLoop
northRoutePos : int
eastRoutePos : int
southRoutePos : int
westRoutePos : int
bendPoints : List
edge : KLayoutEdge
element : LayerElement
sourcePort : KLayoutPort
targetPort : KLayoutPort
<<create>> ElementLoop(edge : KLayoutEdge,elem : LayerElement)
toString() : String
getSourcePort() : KLayoutPort
getTargetPort() : KLayoutPort
applyLayout(offset : KPoint) : void

Figure A.3: Central data structure for the hierarchical layout algorithm, representing a layered graph

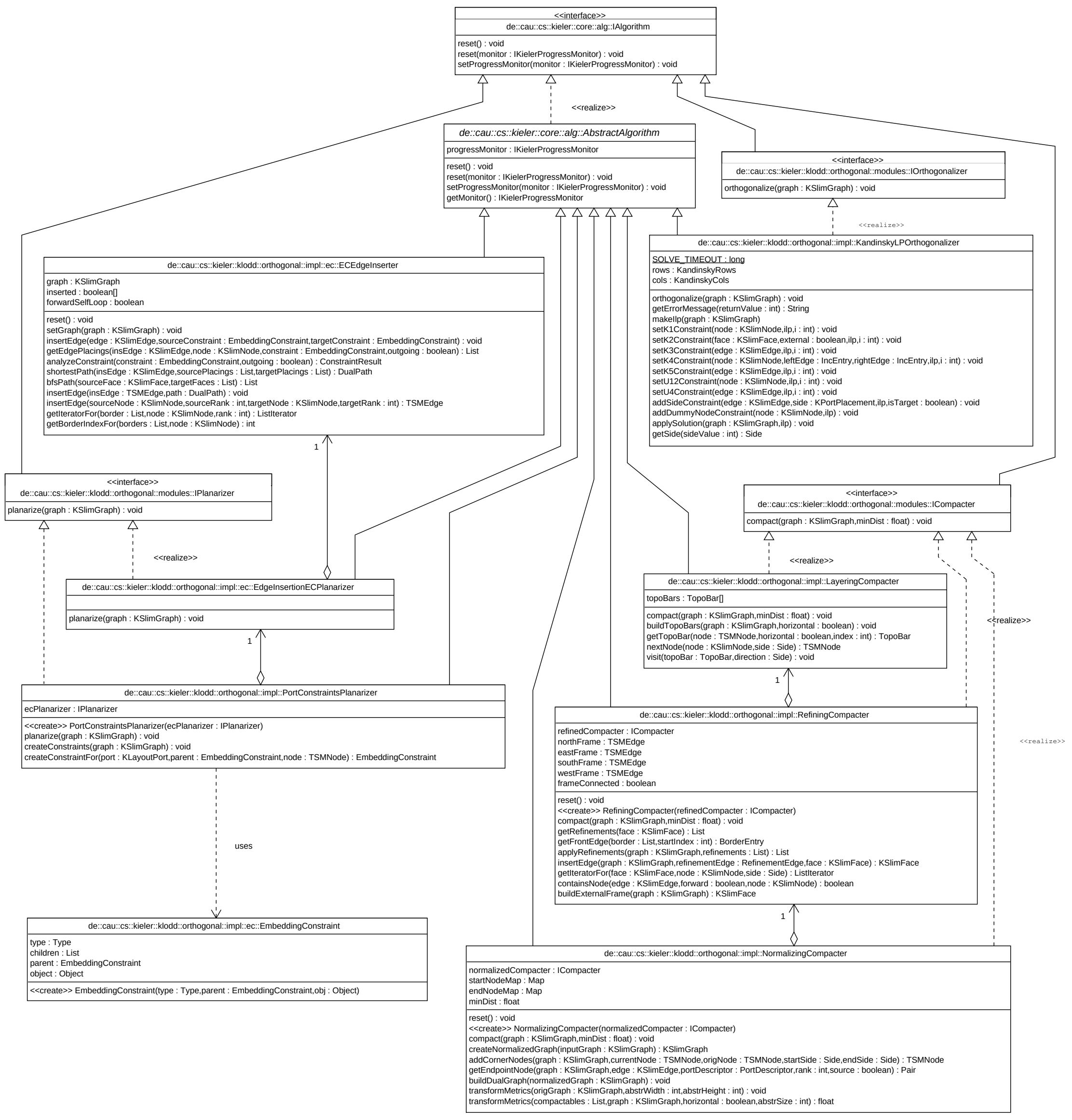

<<interface>>
de::cau::cs::kieler::core::alg::IAlgorithm
reset() : void
reset(monitor : IKielerProgressMonitor) : void
setProgressMonitor(monitor : IKielerProgressMonitor) : void
<<realize>>
de::cau::cs::kieler::core::alg::AbstractAlgorithm
progressMonitor : IKielerProgressMonitor
reset() : void
reset(monitor : IKielerProgressMonitor) : void
setProgressMonitor(monitor : IKielerProgressMonitor) : void
getMonitor() : IKielerProgressMonitor
<<interface>>
de::cau::cs::kieler::klodd::orthogonal::modules::IOrthogonalizer
orthogonalize(graph : KSlimGraph) : void
<<realize>>
de::cau::cs::kieler::klodd::orthogonal::impl::KandinskyLPOrthogonalizer
SOLVE_TIMEOUT : long
rows : KandinskyRows
cols : KandinskyCols
orthogonalize(graph : KSlimGraph) : void
getErrorMessage(returnValue : int) : String
makeIlp(graph : KSlimGraph)
setK1Constraint(node : KSlimNode,ilp,i : int) : void
setK2Constraint(face : KSlimFace,external : boolean,ilp,i : int) : void
setK3Constraint(edge : KSlimEdge,ilp,i : int) : void
setK4Constraint(node : KSlimNode,leftEdge : IncEntry,rightEdge : IncEntry,ilp,i : int) : void
setK5Constraint(edge : KSlimEdge,ilp,i : int) : void
setU12Constraint(node : KSlimNode,ilp,i : int) : void
setU4Constraint(edge : KSlimEdge,ilp,i : int) : void
addSideConstraint(edge : KSlimEdge,side : KPortPlacement,ilp,isTarget : boolean) : void
addDummyNodeConstraint(node : KSlimNode,ilp) : void
applySolution(graph : KSlimGraph,ilp) : void
getSide(sideValue : int) : Side
de::cau::cs::kieler::klodd::orthogonal::impl::ec::ECEdgeInserter
graph : KSlimGraph
inserted : boolean[]
forwardSelfLoop : boolean
reset() : void
setGraph(graph : KSlimGraph) : void
insertEdge(edge : KSlimEdge,sourceConstraint : EmbeddingConstraint,targetConstraint : EmbeddingConstraint) : void
getEdgePlacings(insEdge : KSlimEdge,node : KSlimNode,constraint : EmbeddingConstraint,outgoing : boolean) : List
analyzeConstraint(constraint : EmbeddingConstraint,outgoing : boolean) : ConstraintResult
shortestPath(insEdge : KSlimEdge,sourcePlacings : List,targetPlacings : List) : DualPath
bfsPath(sourceFace : KSlimFace,targetFaces : List) : List
insertEdge(insEdge : TSMEdge,path : DualPath) : void
insertEdge(sourceNode : KSlimNode,sourceRank : int,targetNode : KSlimNode,targetRank : int) : TSMEdge
getIteratorFor(border : List,node : KSlimNode,rank : int) : ListIterator
getBorderIndexFor(borders : List,node : KSlimNode) : int
1
<<interface>>
de::cau::cs::kieler::klodd::orthogonal::modules::IPlanarizer
planarize(graph : KSlimGraph) : void
<<realize>>
de::cau::cs::kieler::klodd::orthogonal::impl::ec::EdgeInsertionECPlanarizer
planarize(graph : KSlimGraph) : void
1
de::cau::cs::kieler::klodd::orthogonal::impl::PortConstraintsPlanarizer
ecPlanarizer : IPlanarizer
<<create>> PortConstraintsPlanarizer(ecPlanarizer : IPlanarizer)
planarize(graph : KSlimGraph) : void
createConstraints(graph : KSlimGraph) : void
createConstraintFor(port : KLayoutPort,parent : EmbeddingConstraint,node : TSMNode) : EmbeddingConstraint
uses
de::cau::cs::kieler::klodd::orthogonal::impl::ec::EmbeddingConstraint
type : Type
children : List
parent : EmbeddingConstraint
object : Object
<<create>> EmbeddingConstraint(type : Type,parent : EmbeddingConstraint,obj : Object)
<<interface>>
de::cau::cs::kieler::klodd::orthogonal::modules::ICompacter
compact(graph : KSlimGraph,minDist : float) : void
<<realize>>
de::cau::cs::kieler::klodd::orthogonal::impl::LayeringCompacter
topoBars : TopoBar[]
compact(graph : KSlimGraph,minDist : float) : void
buildTopoBars(graph : KSlimGraph,horizontal : boolean) : void
getTopoBar(node : TSMNode,horizontal : boolean,index : int) : TopoBar
nextNode(node : KSlimNode,side : Side) : TSMNode
visit(topoBar : TopoBar,direction : Side) : void
<<realize>>
1
de::cau::cs::kieler::klodd::orthogonal::impl::RefiningCompacter
refinedCompacter : ICompacter
northFrame : TSMEdge
eastFrame : TSMEdge
southFrame : TSMEdge
westFrame : TSMEdge
frameConnected : boolean
reset() : void
<<create>> RefiningCompacter(refinedCompacter : ICompacter)
compact(graph : KSlimGraph,minDist : float) : void
getRefinements(face : KSlimFace) : List
getFrontEdge(border : List,startIndex : int) : BorderEntry
applyRefinements(graph : KSlimGraph,refinements : List) : List
insertEdge(graph : KSlimGraph,refinementEdge : RefinementEdge,face : KSlimFace) : KSlimFace
getIteratorFor(face : KSlimFace,node : KSlimNode,side : Side) : ListIterator
containsNode(edge : KSlimEdge,forward : boolean,node : KSlimNode) : boolean
buildExternalFrame(graph : KSlimGraph) : KSlimFace
<<realize>>
1
de::cau::cs::kieler::klodd::orthogonal::impl::NormalizingCompacter
normalizedCompacter : ICompacter
startNodeMap : Map
endNodeMap : Map
minDist : float
reset() : void
<<create>> NormalizingCompacter(normalizedCompacter : ICompacter)
compact(graph : KSlimGraph,minDist : float) : void
createNormalizedGraph(inputGraph : KSlimGraph) : KSlimGraph
addCornerNodes(graph : KSlimGraph,currentNode : TSMNode,origNode : TSMNode,startSide : Side,endSide : Side) : TSMNode
getEndpointNode(graph : KSlimGraph,edge : KSlimEdge,portDescriptor : PortDescriptor,rank : int,source : boolean) : Pair
buildDualGraph(normalizedGraph : KSlimGraph) : void
transformMetrics(origGraph : KSlimGraph,abstrWidth : int,abstrHeight : int) : void
transformMetrics(compactables : List,graph : KSlimGraph,horizontal : boolean,abstrSize : int) : float

Figure A.4: Modules for the orthogonal layout algorithm, all implemented as subclasses of AbstractAlgorithm

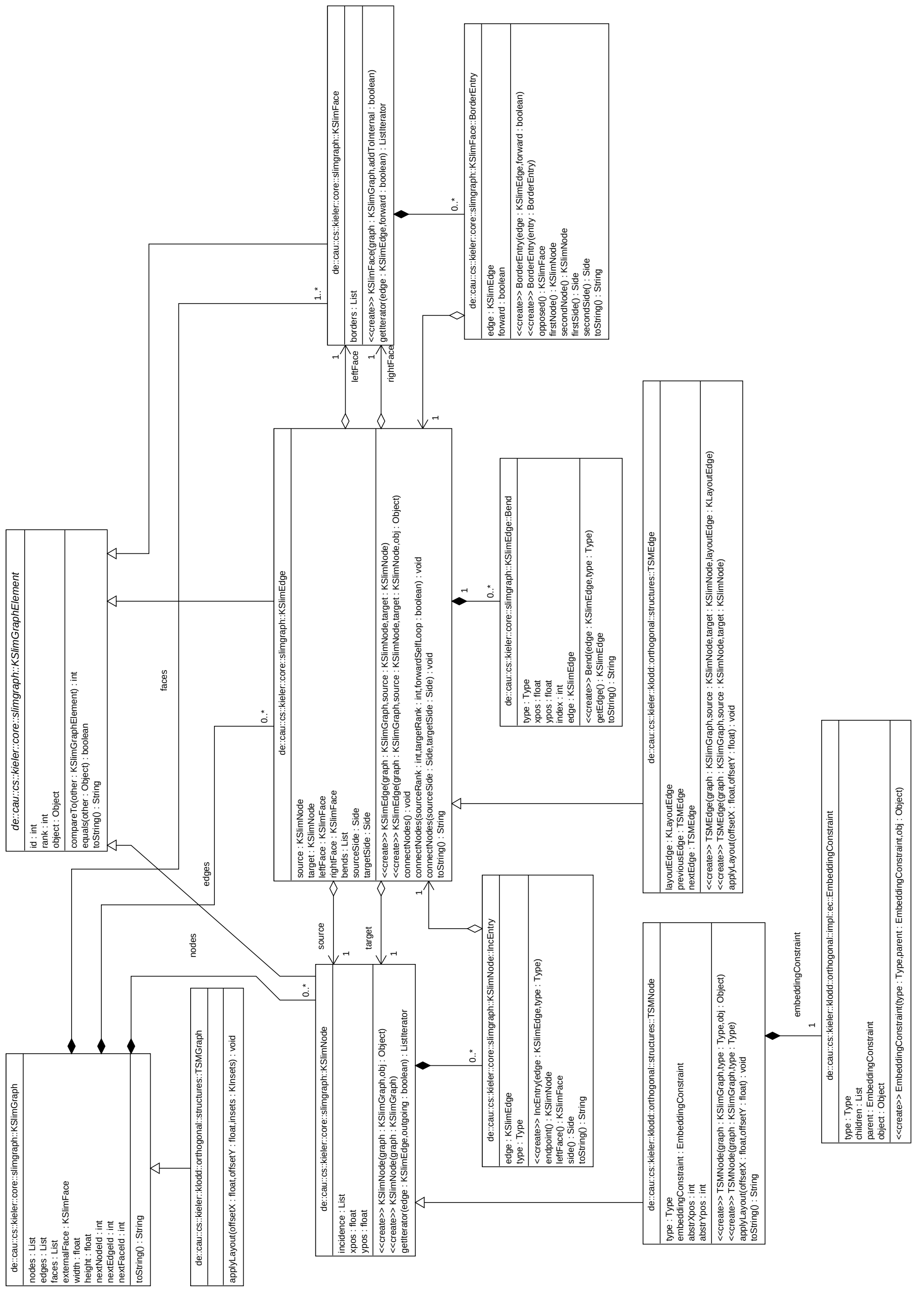

de::cau::cs::kieler::core::slimgraph::KSlimGraph
nodes : List
edges : List
faces : List
externalFace : KSlimFace
width : float
height : float
nextNodeId : int
nextEdgeId : int
nextFaceId : int
toString() : String
de::cau::cs::kieler::klodd::orthogonal::structures::TSMGraph
applyLayout(offsetX : float,offsetY : float,insets : KInsets) : void
de::cau::cs::kieler::core::slimgraph::KSlimGraphElement
id : int
rank : int
object : Object
compareTo(other : KSlimGraphElement) : int
equals(other : Object) : boolean
toString() : String
nodes
edges
faces
0..*
0..*
1..*
de::cau::cs::kieler::core::slimgraph::KSlimNode
incidence : List
xpos : float
ypos : float
<<create>> KSlimNode(graph : KSlimGraph,obj : Object)
<<create>> KSlimNode(graph : KSlimGraph)
getIterator(edge : KSlimEdge,outgoing : boolean) : ListIterator
source
1
target
1
de::cau::cs::kieler::core::slimgraph::KSlimEdge
source : KSlimNode
target : KSlimNode
leftFace : KSlimFace
rightFace : KSlimFace
bends : List
sourceSide : Side
targetSide : Side
<<create>> KSlimEdge(graph : KSlimGraph,source : KSlimNode,target : KSlimNode)
<<create>> KSlimEdge(graph : KSlimGraph,source : KSlimNode,target : KSlimNode,obj : Object)
connectNodes() : void
connectNodes(sourceRank : int,targetRank : int,forwardSelfLoop : boolean) : void
connectNodes(sourceSide : Side,targetSide : Side) : void
toString() : String
1
leftFace
1
rightFace
de::cau::cs::kieler::core::slimgraph::KSlimFace
borders : List
<<create>> KSlimFace(graph : KSlimGraph,addToInternal : boolean)
getIterator(edge : KSlimEdge,forward : boolean) : ListIterator
1
0..*
de::cau::cs::kieler::core::slimgraph::KSlimFace::BorderEntry
edge : KSlimEdge
forward : boolean
<<create>> BorderEntry(edge : KSlimEdge,forward : boolean)
<<create>> BorderEntry(entry : BorderEntry)
opposed() : KSlimFace
firstNode() : KSlimNode
secondNode() : KSlimNode
firstSide() : Side
secondSide() : Side
toString() : String
0..*
1
de::cau::cs::kieler::core::slimgraph::KSlimNode::IncEntry
edge : KSlimEdge
type : Type
<<create>> IncEntry(edge : KSlimEdge,type : Type)
endpoint() : KSlimNode
leftFace() : KSlimFace
side() : Side
toString() : String
1
0..*
de::cau::cs::kieler::core::slimgraph::KSlimEdge::Bend
type : Type
xpos : float
ypos : float
index : int
edge : KSlimEdge
<<create>> Bend(edge : KSlimEdge,type : Type)
getEdge() : KSlimEdge
toString() : String
de::cau::cs::kieler::klodd::orthogonal::structures::TSMNode
type : Type
embeddingConstraint : EmbeddingConstraint
abstrXpos : int
abstrYpos : int
<<create>> TSMNode(graph : KSlimGraph,type : Type,obj : Object)
<<create>> TSMNode(graph : KSlimGraph,type : Type)
applyLayout(offsetX : float,offsetY : float) : void
toString() : String
de::cau::cs::kieler::klodd::orthogonal::structures::TSMEdge
layoutEdge : KLayoutEdge
previousEdge : TSMEdge
nextEdge : TSMEdge
<<create>> TSMEdge(graph : KSlimGraph,source : KSlimNode,target : KSlimNode,layoutEdge : KLayoutEdge)
<<create>> TSMEdge(graph : KSlimGraph,source : KSlimNode,target : KSlimNode)
applyLayout(offsetX : float,offsetY : float) : void
embeddingConstraint
1
de::cau::cs::kieler::klodd::orthogonal::impl::ec::EmbeddingConstraint
type : Type
children : List
parent : EmbeddingConstraint
object : Object
<<create>> EmbeddingConstraint(type : Type,parent : EmbeddingConstraint,obj : Object)

Figure A.5: Central data structure for the orthogonal layout algorithm, representing an undirected graph