INSTITUT FÜR INFORMATIK

Automatic Layout of Data Flow Diagrams in KIELER and Ptolemy II

Miro Spönemann, Hauke Fuhrmann,
Reinhard von Hanxleden

Bericht Nr. 0914
Juli 2009

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Automatic Layout of Data Flow Diagrams in KIELER and Ptolemy II

Miro Spönemann, Hauke Fuhrmann,
Reinhard von Hanxleden

Bericht Nr. 0914
Juli 2009

e-mail:
msp@informatik.uni-kiel.de,
haf@informatik.uni-kiel.de,
rvh@informatik.uni-kiel.de

Technical Report

Abstract

Data flow diagrams are successfully applied in the area of model-based design of complex embedded systems. However, their creation and maintenance can be very time-consuming, because many tools offer little support for the editing and visualization of graphical models. The KIELER project explores new concepts for the pragmatics of graphical modeling and develops algorithms for automatic layout of specific classes of diagrams. These concepts and algorithms are implemented as extensions of the Eclipse framework, which offers generic approaches to create IDEs for graphical modeling.

We have developed a specialized layout algorithm for data flow diagrams. In addition to the embedding in KIELER, we applied this algorithm to Ptolemy, a framework for research on models of computation for use in embedded systems. The results show that our algorithm is well suited for the actor oriented diagrams of Ptolemy, and it can serve as a basis to facilitate the editing of Ptolemy diagrams.

Key words: Automatic layout, graphical modeling, Eclipse, Ptolemy, hierarchical layout, data flow

Contents

List of Figures

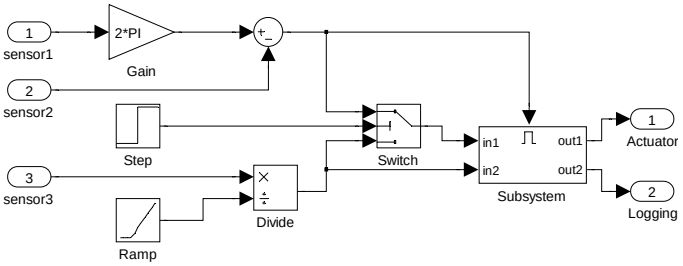
Chapter 1

Introduction

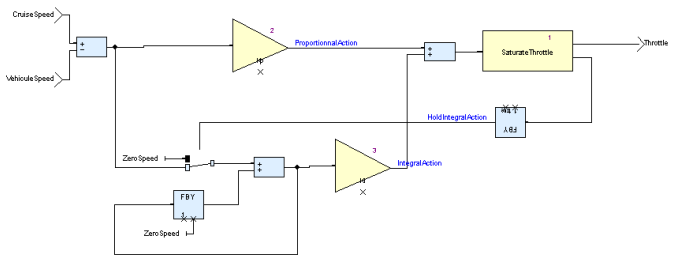
Graphical modeling languages have evolved to appealing and convenient instruments for the development and documentation of systems, both in hardware and in software. There are various examples for graphical modeling frameworks that have become an important part of modern development processes. An important class of modeling diagrams are *data flow diagrams*, which are graphical representations of *data flow models* for the design of complex systems. Applications of data flow diagrams can be found in modern software and hardware development tools. Some of these, such as Simulink (The MathWorks, Inc.), LabVIEW (National Instruments Corporation), and ASCET (ETAS Inc.), are mainly used for model-based design and simulation of embedded systems and digital or analog hardware, while others, such as SCADE (Esterel Technologies, Inc.), are optimized for automatic code generation from high-level system models. The Ptolemy project [7] features data flow diagrams for *actor-oriented design*, where *actors* exchange 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 drag-and-drop manner. Example diagrams are presented in Figure 1.1.

A data flow model is described by a directed graph where the vertices represent *operators* that compute data and the edges represent data paths [4]. Such a data path has a specified *source port* where data is created and a *target port* where data is consumed. A source port may be connected with multiple target ports, thus forming a *hyperedge*. Furthermore, the data flow paths are required to be drawn orthogonally. These properties of data flow diagrams can be defined as a set of constraints for the drawing of the corresponding directed graph.

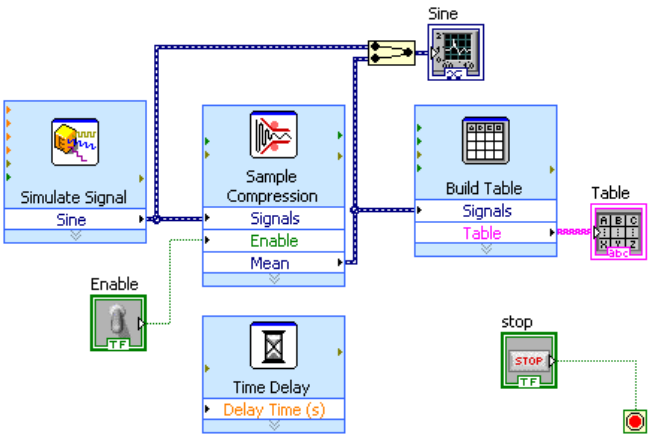
While most of the research on modeling languages has its focus on the *semantics* of these languages, the *pragmatics* of graphical modeling have received relatively little attention so far [11]. The latter includes the aspects



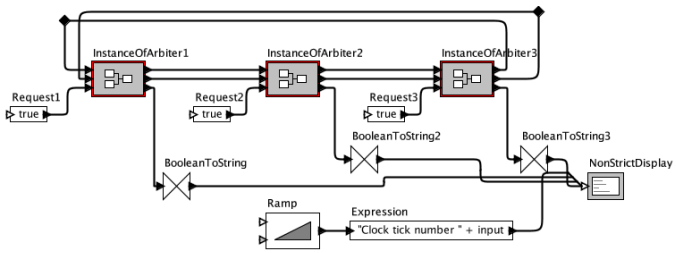
(a) Simulink



(b) SCADE



(c) LabVIEW



(d) Ptolemy

Figure 1.1: Data flow diagrams from graphical modeling tools

of editing, visualization, and simulation of graphical models, and is the principal topic of the KIELER project. The ability to automatically layout a graphical model is a key issue in this context, because it relieves the user of the burden of manually adapting the arrangement of objects after each structural change in the model, or after switching to a different view of the model. In this report we present the basic concepts of how automatic layout is handled in KIELER. We describe our interface to Eclipse, which is the fundamental platform for our implementation, and point out how we enable the embedding of layout algorithms in a large class of graphical editors in Eclipse. Furthermore, we present our integration of a specific layout algorithm into the graphical editor of Ptolemy and show results of the integration. We have implemented this algorithm following the *layered* approach for graph drawing proposed by Sugiyama *et al.* [33], and have extended it for the special requirements of data flow diagrams.

We will proceed as follows. Chapter 2 introduces KIELER and its interface for automatic layout. The layered approach for graph drawing and our extensions for layout of data flow diagrams are described in Chapter 3. Readers who are not interested in the details of the layout algorithm may skip this and proceed to Chapter 4, where we describe how we apply our algorithm to Ptolemy and show some results. We conclude in Chapter 5.

Find a concentrated description in the corresponding paper in cooperation with Mutzel [31].

Chapter 2

KIELER

The pragmatics of model-based design refers to the practical aspects of handling graphical system models. This encompasses a range of activities, such as editing, browsing, or simulating models. We believe that the pragmatics of modeling deserves more attention than it has received so far. We also believe that there is the potential for significant productivity enhancements, using technology that is largely already available. A key enabler here is the capability to automatically and quickly compute the layout of a graphical model, which frees the designer from the burden of manual drawing. This capability also allows to compute customized views of a model on the fly, which offers new possibilities for interactive browsing and for simulation.

2.1 Aims and Project Vision

The project Kiel Integrated Environment for Layout for the Eclipse Rich-ClientPlatform (KIELER)[1] focuses on enhancing pragmatics of graphical model-based system design. By that we comprise all practical aspects of handling a model in a model-based design flow, including the traditional aspect of how a model should be constructed to effectively communicate its meaning.

The goal is to free the user of many mechanical efforts such as manual placing of graphical items on the canvas. This would help the developer to focus on the topology and semantics of the system he or she intends to design, and not to waste too much effort in *enabling steps* such as making space on the canvas for new graphical elements.

On top of that, new methodologies for user interaction are explored to increase modeling productivity and the degree of ability to analyze complex graphical models. The basic technologies will be applied by certain use cases

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

that directly help the developer in the interaction with graphical models. More in-depth information about the vision and goals of the KIELER project can be found elsewhere [11].

2.1.1 Subprojects of KIELER

The KIELER project investigates and evaluates the enabling technologies for graphical modeling and tries to fill gaps where necessary. Its main focus is at investigating new interaction mechanisms with graphical models by different aspects.

View Management and Meta Layout

View Management describes ways how to play with different graphical representations for models. Following the Model-View-Controller (MVC) paradigm it creates different views for different purposes or under different circumstances of a model. It accesses a set of different functions to manipulate a graphical view, such as reducing *levels of detail* of certain diagram elements (*e. g.* folding and unfolding of composite elements or filtering out complete items) or highlighting graphical objects. It tries to show the user the "interesting" parts of a model and hide the others, while the user specifies what "interesting" means in a specific context.

Meta Layout uses different automatic layout approaches—possibly on different parts in the same diagram—to select optimal layouts for given criteria.

Model Editing and Synthesis

Creation and modification of graphical models is an issue, because the what-you-see-is-what-you-get (WYSIWYG) drag-and-drop (DnD) style editing, which is still state-of-the-practice, involves many effort-prone manual editing steps.

With automatic layout, the user can focus on *what* items have to be added or changed *where in the topology* of the model and does not need to care about the locations on the two-dimensional canvas.

Structural Editing is the idea to employ this systematically by only offering structural editing operations, *e. g.* "add a new following state for state B" instead of "add a new state at coordinates x and y and connect it to B." As a fast automatic layout algorithm can result in an immediate graphical feedback of such an operation, the WYSIWYG experience is still available.

Automatic layout opens complete new possibilities with the *synthesis* of graphical models. Any—possibly textual—source could be used to generate

a graphical model automatically in order to better visualize the concepts of the system.

Dynamic Behavior Analysis

Simulation of graphical models allows to execute a system and to analyze its properties and behavior. Dynamical behavior models, *e. g.* for embedded system design or modeling of physical environments, usually change their properties over simulation time. For any simulation step it helps the developer to understand the system if the current *system state* is visualized directly within the graphical model that describes the system behavior. Possible visualizations could be to highlight items representing states, to show variable values, or to display animations of physical system parts.

However, for complex systems and with a limited screen size, it is not possible to cover both goals at the same time: to see the system's overview and to see the components' details. It is a matter of screen real estate to arrange different windows to see during runtime what seem to be interesting parts of the model.

Here view management comes into play and tries to identify *foci* of the model that have to be presented with full levels of detail, while others go into the *context* and might be displayed with less detail or not at all. This likely creates multiple views that are dynamically changed during simulation time—most suitably morphed/animated between to preserve the user's mental map of the system. This has been explored in the KIEL system for Statecharts [22], and is aimed more generally for other graphical syntaxes and execution semantics in KIELER.

Automatic Layout

The basic technology to build upon is a sound automatic layout of the graphical diagrams. There are different approaches to the layout problem, with quite different results, where some algorithms—like the one presented in this paper—focus on aspects of special properties of some graphical languages. Hence for different graphical languages, even for different diagrams in the same language, and for different means of the diagrams, also different layout algorithms might result in optimal views. Therefore KIELER tries to build interfaces to gather different layout algorithms and to unify the API to access them on the one hand, and to create new algorithms on the other hand.

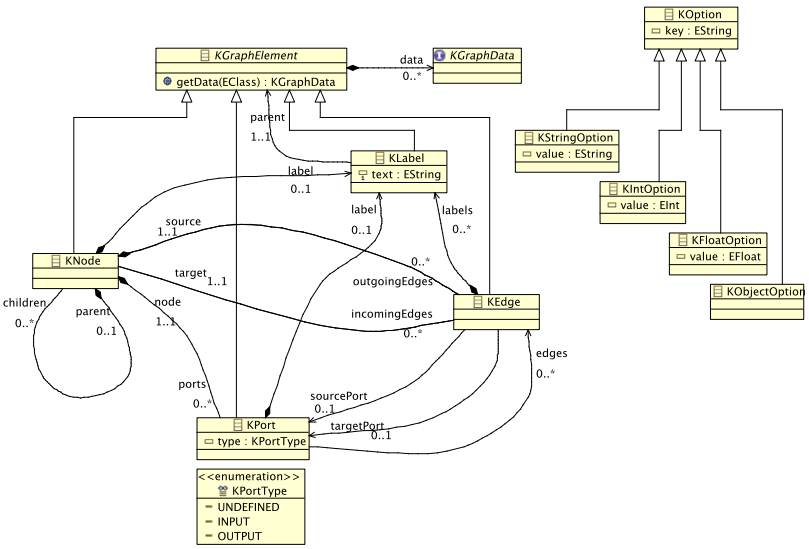


Figure 2.1: Class Diagram for the KGraph data structure

2.2 Implementation Structure

2.2.1 Interface for automatic layout

Input and output for automatic layout is stored using a data structure called KGraph, which is an Eclipse Modeling Framework (EMF) model that describes the basic structure of the graph. As seen in the class diagram in Figure 2.1, each node in this model may contain other nodes in a *parent-child* relationship, thus allowing structural hierarchy. The KGraphData interface is used as generic extension for each element of the graph. In the context of graph layout each element has an attached instance of KLayoutData, which is an extension of KGraphData for storage of layout information. This layout information consists of *layout options* that are passed to the layout algorithm to control its behavior, and of concrete layout data such as position and size for nodes, ports, and labels, or a list of bend points for edges. A class diagram for the KLayoutData extension is shown in Figure 2.2.

Layout algorithms must be implemented as subclasses of AbstractLayoutProvider in the KIELER Infrastructure for Meta Layout (KIML). Such a subclass has a method doLayout, which takes a KNode and an IKielerProgressMonitor as arguments. The first argument is the parent node of the graph for which layout shall be performed, and the second argument is an instance of a *progress monitor*, that is a class used to track progress and execution time of an algorithm.

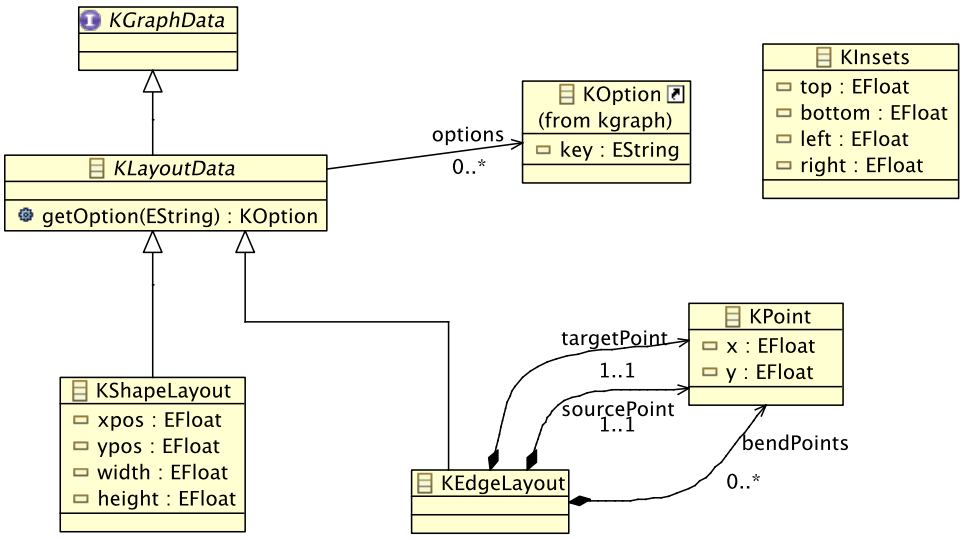


Figure 2.2: Class Diagram for the KLayoutData data structure

Automatic layout in KIELER is done with the following steps.

1. Create a KNode representing the graph that shall be layouted.
2. Add layout options to each element of the graph.
3. Create an instance of IKielerProgressMonitor, possibly with a connection to the User Interface (UI) to display progress of the algorithm.
4. Create an instance of the selected layout provider. This instance may be kept in memory for multiple executions of the algorithm.
5. Call the doLayout method of the layout provider instance, passing the KNode instance and the progress monitor.
6. Read layout data from each graph element and apply it to the original graph.

Layout options are managed by the class LayoutOptions in KIML, which contains utility methods to access the current values of layout options for KLayoutData instances of a graph element. The currently available layout options are the following.

- Options for parent nodes

  Layout Direction: Choose whether edges are aligned left to right (horizontal layout) or top down (vertical layout).

Minimal Spacing: If possible, the distance between all nodes and edges in the drawing should be at least this value.

Insets: Distance between the border of the parent node and the drawing of the contained diagram on the left, right, top, and bottom side.

Layout Hint: Indication on which layout algorithm should be used for the contained diagram, *e. g.* the identification string of a specific layout provider.

- Options for all nodes

  Port Constraints: Determine how the layout algorithm should handle port positions (see Section 3.3).

  Fixed Size: Choose whether the size of the node is fixed.

- Options for ports

  Port Side: Side of the node on which the port is placed: *North*, *East*, *South*, or *West*.

  Port Rank: Ranks are used to order the ports in clockwise direction beginning from the top left corner of the node.

- Options for labels

  Edge Label Placement: Choose whether the label is placed at the head, tail, or in the middle of the edge.

Further layout options can be easily defined if they are needed for specific layout problems, *e. g.* different edge types that need special handling.

2.2.2 Layout of Compound Diagrams

If a node in a diagram contains other nodes, we call this a *compound diagram*, or a diagram with *structural hierarchy*. Since most layout algorithms are designed to handle only flat diagrams, we need a mechanism to extend layout algorithms for hierarchy. In KIML this is done using *layouter engines*, which handle selection of appropriate layout algorithms and mapping of flat diagram layout to hierarchical diagrams. The basic implementation is done in the class RecursiveLayouterEngine, which first performs layout on the most inner diagrams, then on the containing diagram (see Algorithm 2.1). After executing a layout algorithm on an inner diagram, its size is known and can be used as fixed size for the corresponding parent node. This is shown in

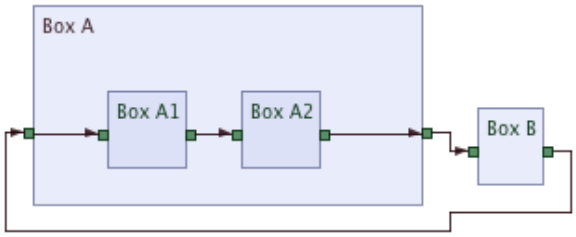


Figure 2.3: Layout of compound diagrams

Figure 2.3, where the diagram in Box A is arranged first, then it is treated as a normal node of fixed size for layout of the surrounding diagram.

Algorithm 2.1: RecursiveLayouterEngine

```
1 procedure recursiveLayout(N: KNode)
2     if N has any children then
3         for each child N_i of N do
4             recursiveLayout(N_i)
5         select a layout provider P for N
6         execute P on N
7 end
```

The recursive layouter engine is only appropriate for diagrams without edges that connect nodes from different levels of the hierarchy. If the diagrams allow these cases, a more complex layout algorithm should be used that directly handles all levels of hierarchy [26, 32]. As an example for such diagrams, some variants of Statecharts allow *inter-level transitions* between states of different hierarchy levels.

If multiple layout providers are available, KIML offers a class KimlLayoutServices to organize them, where layout providers are stored using a hash map with their identifiers as keys. The *Layout Hint* layout option, which can be attached to the layout data of any node, can be used to specify which layout provider to use for each parent node. This enables the user to choose different layouts for multiple levels of hierarchy in the diagram. An example for a diagram for which automatic layout with different layout providers was performed is shown in Figure 2.4.

2.3 Integration into Eclipse

The implementation of KIELER is built upon a Rich Client Platform (RCP). This was done in order to reach a broad community with the tools, and to

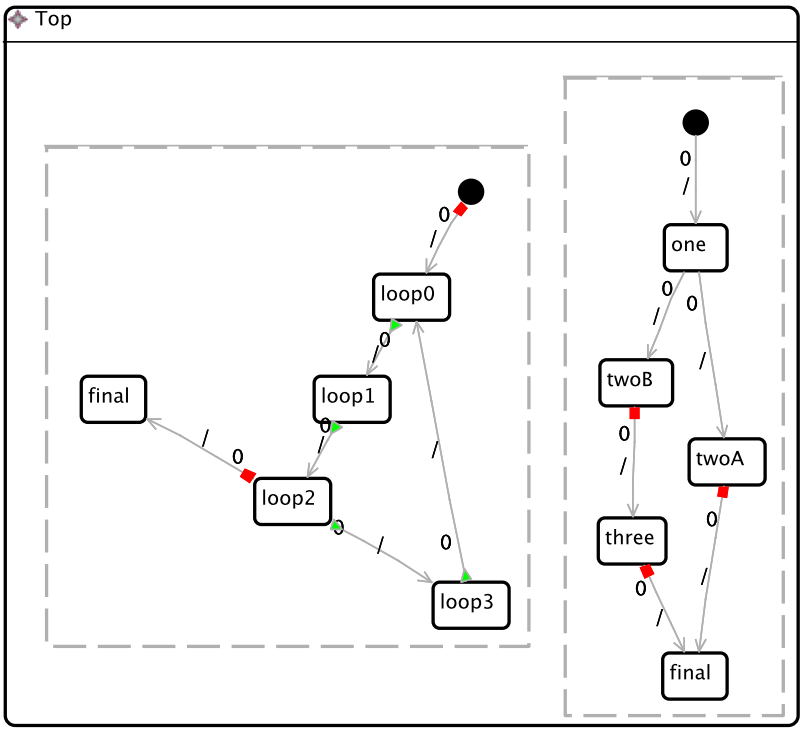


Figure 2.4: A diagram with mixed layouts: force-directed approach in the left region, layered approach in the right region

build upon a wide set of existing frameworks in that platform to get most synergies of existing tools and software solutions.

We settled for the Eclipse platform because it provides simple mechanisms to modularize an application, so-called *plug-ins*, but also to specify *extension points* where future plug-ins may extend your own application. These mechanisms have been accepted and leveraged by a big user community which evolves many sophisticated software solutions of a wide diversification. The KIELER project uses standardized structural backbones of Eclipse in order to concentrate on the implementation of the main KIELER focus. On the other hand, Eclipse has originated not only an Integrated Development Environment (IDE) for many different textual programming languages, but also environments for graphical languages.

If you are not interested in Eclipse and how KIELER works and integrates with it, you can proceed to Chapter 4 and read about Ptolemy and the stand-alone algorithm integration.

2.3.1 Graphical Modeling Framework

Due to the big user community, quite a few graphical editors have been created for different graphical syntaxes and with different features within Eclipse.

To reduce the manual implementation efforts for each and every new graphical editor, the eclipse community has evolved architecture projects[2] to consolidate the similarities of such graphical editors.

First, the Eclipse Modeling Framework (EMF) standardizes the way how to create a semantic *model* that underlies a graphical representation, also called a *domain* or *business model*. It can be used to comfortably create abstract data structures with standardized and machine accessible interfaces and convenient built-in features such as persistence support (XML serialization or database binding), comparison, transformation, *etc.*

Next to this the Graphical Editing Framework (GEF) provides extensive libraries for low level two-dimensional drawing and high level graphical editor functionality. This comprises a controller framework to control user interaction with the model (DnD style editing) and standard widgets such as toolbars and a palette with standard functions.

Still, a graphical editor is manually programmed with the use of these libraries. Especially the graphical part has to be linked to some custom model representation that the developer has to provide.

The Graphical Modeling Framework (GMF) extends the GEF libraries

[2]http://www.eclipse.org/projects/

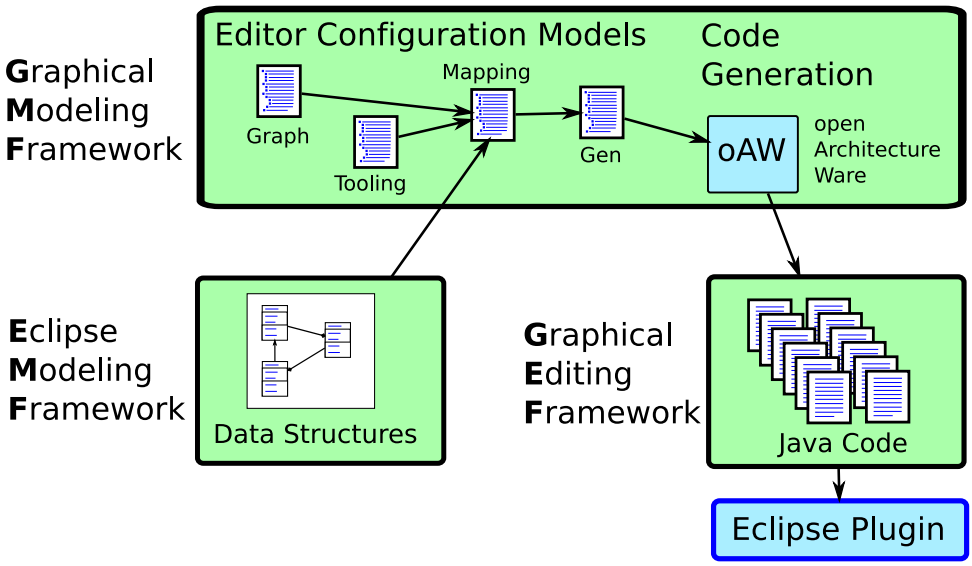


Figure 2.5: Overview of the Eclipse modeling architecture with GMF

and bridges EMF and GEF. It provides a fixed way how to map GEF editors onto EMF models. Additionally it offers a powerful generative approach to automatically generate the required Java code of a GEF editor from high-level editor specifications as depicted in Figure 2.5.

With this integrated approach for rapid prototyping of graphical editors the community has created a wide variety of graphical editors; either for standard graphical syntaxes such as the Unified Modeling Language (UML)[3] or for Domain Specific Modeling Languages (DSMLs). An example editor for Statecharts created in the KIELER project is shown in Figure 2.6.

This puts GMF into an interesting light regarding the goals of KIELER. Any improvements that are fed into any of the modeling projects, EMF, GEF or GMF, will provide the benefits for all graphical editors that are created with the frameworks. If the improvements are implemented into the code generation parts, then all foreign editors will benefit from the new technology after they have regenerated their editor code. If the changes were made to the runtime libraries, then all editors would benefit from it immediately after they updated the libraries. Hence by this means the new methodologies can reach broad new audiences easily.

[3]http://www.eclipse.org/modeling/mdt/

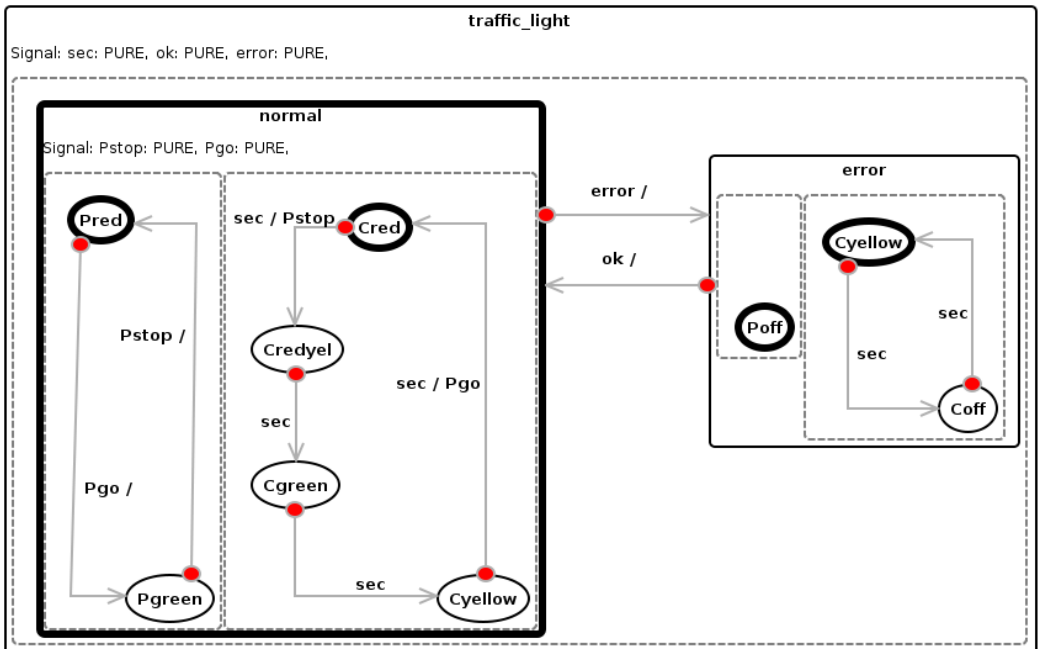


Figure 2.6: A Statechart Editor in Eclipse GMF

2.3.2 Integration with GMF

With the KGraph described in Section 2.2.1, KIELER has an interface to exchange graph data with layout algorithms on the one hand and with actual modeling tools on the other hand. This interface can be directly used to connect an algorithm to some graphical editor like we did for Ptolemy II as described in Chapter 4. This is especially necessary if the modeling tool does not build upon Eclipse but is a pure Java application in order to avoid any dependencies to Eclipse. For other programming languages one could use the KGraph as a file interface, because the KGraph is based on EMF and therefore provides standard mechanisms to serialize and parse a graph to and from XML files.

However, for graphical editors created with Eclipse GMF we can unify the interface even further such that the graphical tool designer (the so-called *toolsmith*) only cares as little as possible about that interface.

Additionally all higher level KIELER services—an overview of the ideas is given in Section 2.1—can access the layout functionality through this interface. Hence these services are yet only available for GMF based editors and not for the stand-alone algorithm.

The interfaces are located in the KIELER Infrastructure for Meta Layout

(KIML) component. In the Eclipse world an interface consists of an *extension point*, schematically shown in Figure 2.7 and explained in the following.

A plug-in consists of Java code and additional textual (and XML) meta information. Eclipse can use this to avoid loading Java classes of a plug-in that is not requested at runtime. Basic information about the plug-in is provided to the platform by this meta information and allows lazy loading of the Java classes.

Eclipse manages fine grain modularity by providing a separate class loader for every plug-in. Hence each plug-in's Java code is contained in the class loaders sand-box and generally has no access to other code. This way code of one plug-in does not interfere with other versions of that plug-in that are present in the platform.

The simplest way to access some code from another plug-in is to set an explicit *dependency* on that plug-in, which is a simple statement in the dependent plug-in's meta information. That enables to access all published Java code (interfaces as well as functionality).

Hence this enables an extending plug-in to use code of some a priori known base plug-in. However, the other way round is not that simple. Imagine a base plug-in allows to be extended by some other plug-in but the base plug-in does not yet know about any particular extending plug-ins yet. Still, the base plug-in might need a way to instantiate a class from the extending plug-in. Say a base plug-in A offers a way to display a visualization widget in an Eclipse view. Then an extending plug-in, say B, will implement such a widget, let's say by drawing an oscilloscope. B needs to hand over the implementation class to A such that A can use it for placing it on its view (A cannot instantiate B's class by itself, because it cannot know a priori what class that will be, as we assume that A and B will not be published at the same time, especially B is not available at A's compile time).

In Eclipse this is solved by the mechanism of *extension points*. For an extension point a base plug-in (A) provides a textual meta description of what data the extending plug-in (B) will have to provide. This usually includes a Java Interface that is defined by the base plug-in and will have to be implemented by the extending plugin. Additionally, A can request specific data in the extension point that A needs to work with before it instantiates the class of B. The data is transmitted via an XML document and may comprise unique identifiers, display names *etc.*. As XML is involved, the specification what data is required is done with an XML Schema Definition (XSD), hence the extending plug-in gets a formal description of what data is required to extend a specific extension point. However, this formal description usually is abstracted by a human friendly GUI front-end.

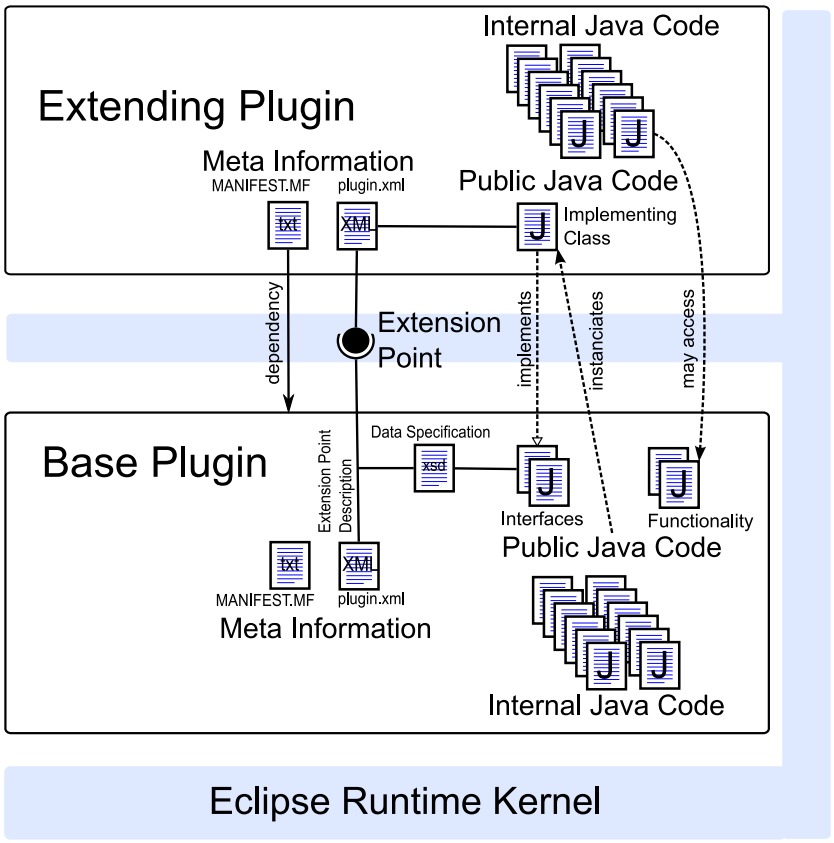


Figure 2.7: Extension Points in Eclipse

KIELER Extension Points

In KIELER the extension point mechanism is used to interface and extend the infrastructure with layout algorithms on the one hand and with actual graphical editors on the other hand. We will discuss these two aspects in the following.

KIELER embeds into Eclipse the functionality to apply any layout algorithm—given by a *Layout Provider*—to an arbitrary GMF editor diagram. To have a clean, standardized interface for all steps, we use the KGraph datastructure as described in Section 2.2.1 as an exchange format.

The steps of a layout run are

1. *Build a layout graph.* From the original GMF diagram a graph structure of the KGraph type is constructed. A diagram can be quite complex and contain information that is not relevant for layout, while the KGraph is designed for layout and comprises only data relevant for that purpose. Specific options for layout can be noted in the KGraph.

2. *Execute layout algorithm.* A selected layout provider gets instantiated and its layout method is run on the KGraph. It will augment the graph structure by the layout information—locations and sizes of nodes and locations of edge bend points.

3. *Apply layout to original diagram.* The layout information gets read from the KGraph and is reapplied to the original diagram.

Extension Point layoutProviders The KIELER Infrastructure for Meta Layout (KIML) subproject takes care about multiple different layout algorithms and contains an extension point to extend the framework by new layout algorithms. All current algorithms are interfaced this way.

The extension point requires to extend the abstract class AbstractLayoutProvider as described in Section 2.2.1. This demands a method doLayout(KNode,IKielerProgressMonitor) that passes a graph in the KGraph representation and expects the implementation to layout the graph by setting the graph's layout attributes. Hence the KGraph datastructure explained above is a main part of the interface and available in the public section of the layout plug-in.

Furthermore, this extension point can be used to define new layout options that can be passed to the algorithms. These options can then be configured for each individual diagram to optimize the result of automatic layout.

Extension Point layoutInfo One goal of KIML is to provide its functionality for all GMF diagrams. However, not all layout algorithms may be suitable for a specific kind of diagram. The extension point layoutInfo can be used to define diagram types and to map specific parts of GMF diagrams to diagram types, so the user is free to choose which algorithms should be applied to each type. For example, each level of hierarchy in a Statecharts diagram can be mapped to the *state machine* diagram type, for which the Graphviz *dot* layouter is a suitable algorithm.

Chapter 3

The Layered Approach for Graph Layout

The layered approach, which is also called *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 [33], and is often called *Sugiyama* layout.

This chapter provides basic definitions for graphs and drawings of graphs, an overview of the hierarchical layout algorithm, some details on our implementation in KIELER, and experimental results. Readers who are not interested in the details of the algorithm may proceed to Chapter 4, which describes our integration into Ptolemy.

3.1 Related Work

As data flow diagrams can be structurally mapped to graphs, basic algorithms for layout of such diagrams can be taken from the area of graph drawing. There are several approaches to the general problem of graph drawing [2, 5, 18, 20, 35], of which a selection is presented in this section.

Layered approach: The *layered* or *hierarchical* layout method first eliminates directed cycles in the graph, then determines a layering of vertices and optimizes this layering with respect to vertex positions. We used this approach for our implementation, therefore it is covered with more detail in Section 3.4.

Force-directed approach: This approach creates a model of physical forces and minimizes the energy of the model [6]. One variant consists in assigning springs with appropriate forces to each pair of adjacent vertices;

such methods are called *spring embedders*.

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. Methods which build on planarization are the following.

Topology-shape-metrics approach: This method computes a bend-minimal orthogonal representation of the graph in an *orthogonalization* phase, and determines final coordinates for vertices and bend points during *compaction* [34, 35].

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 [2, 36].

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 [10, 29], or a biconnected graph [3]. In their basic variants, these algorithms usually yield a straight-line drawing.

Mixed model approach: This approach extends methods of straight-line drawing from the augmentation approach to construct orthogonal or quasi-orthogonal drawings [16, 19].

The main specialties that make layout of data flow diagrams more difficult than layout of general graphs are ports and hyperedges. Previous work on layout with port constraints includes that of Gansner *et al.* [13] and Sander [24], who gave extensions of the hierarchical approach to consider attachment points of edges. These methods are mainly designed for the special case of displaying data structures and are not suited for the more general constraints of data flow diagrams. A more flexible approach is chosen in the commercial graph layout library yFiles (yWorks GmbH), which supports two models of port constraints and hyperedge routing for the hierarchical approach[1], but no details on the algorithm have been published [37]. Other unpublished solutions to drawing with port constraints include ILOG JViews [28] and Tom Sawyer Visualization[2]. Handling of hyperedges in hierarchical layout has been covered by Eschbach *et al.* [9] and Sander [27]. Sugiyama *et al.* [32] and Sander [26] have shown how to draw general compound graphs, but due to the presence of external ports (see Section 3.3), our requirements for structural hierarchy are different.

[1] yFiles Developer's Guide, `http://www.yworks.com/`

[2] Tom Sawyer Software, `http://www.tomsawyer.com/`

The topic of visualization of hardware schematics is quite related to the drawing of data flow diagrams. While traditional approaches for the layout of schematic diagrams follow the general *place and route* technique from VLSI design [1,21], more recent work includes some concepts from the area of graph drawing [8]. However, these concepts are not sufficient for the needs of our application, since they do not address our scenarios for port constraints, but concentrate on partitioning and placement for large schematics and hyperedge routing.

3.2 Graph Drawing

A *directed graph* $G = (V, E)$ consists of a finite set V and a multiset $E \subseteq V \times V$. 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*. The vertices u, v 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. 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. A *subgraph* of $G = (V, E)$ is a graph $G' = (V', E')$ for which $V' \subseteq V$ and $E' \subseteq \{(u, v) \in E : u, v \in V'\}$.

A *path* of a graph is a sequence $(v_1, \ldots, v_k)$ of vertices such that $(v_i, v_{i+1}) \in E$ 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.

A *drawing* of a graph G is a mapping Γ of the vertices and edges of G to subsets of the plane $\mathbb{R}^2$. A drawing is called *polyline* if the drawing $\Gamma(e)$ of each edge e can be decomposed into a sequence of straight lines, and it is *orthogonal*, or *rectilinear*, if all line segments are aligned horizontally or vertically.

Algorithms for automatic layout are programs that compute drawings of the related graphs. These drawings are represented by abstract values such

as the position and size of each vertex, and the list of bend points of each edge. Aside from general restrictions and drawing conventions, algorithms for automatic layout are subject to the goal of optimizing a set of *aesthetics criteria* [2, 23]. 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.

3.3 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 $p \in P(u)$.

In general graph drawing it is sufficient that the drawing of each edge $e = (u, v)$ touches the drawings of u and v anywhere on their border. For port based graphs the drawing of each port $p \in P(v)$ has a specific position on the border of $\Gamma(v)$, and the edges that have p as source or target port may touch $\Gamma(v)$ only at that position.

Some new aspects must be considered when extending a graph layout algorithm to handle ports. Firstly, edges that are incident at the same port are considered as *hyperedges*, *i. e.* edges that may connect more than two vertices. This aspect is handled by merging some line segments of edges that share the same port, as seen in Figure 3.1, and is mainly a matter of proper edge routing. A second aspect concerns port positions, for which we consider four different scenarios:

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

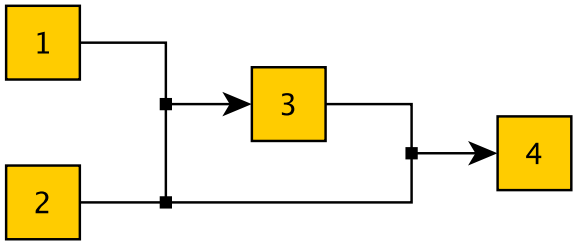


Figure 3.1: A hyperedge that connects four vertices

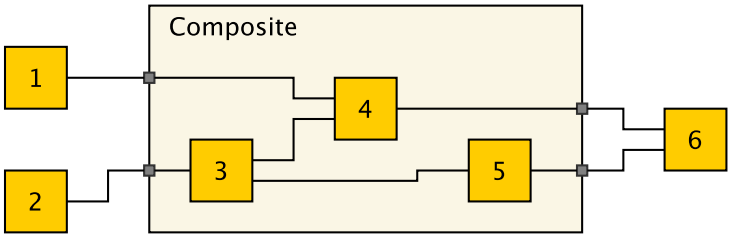


Figure 3.2: The diagram contained in Composite has connections to external ports.

FixedSides The side of the vertex is prescribed for each port, *i. e.* the top, bottom, left, or right border, but the order of ports is free on each side.

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

FixedPorts The exact position is fixed for each port.

When structural hierarchy is applied we use *compound graphs*, where a vertex v is allowed to contain a nested graph G_v. In this case, the ports of v are treated as *external ports* of G_v, and may be connected to the vertices of G_v (see Figure 3.2). As opposed to the ordinary vertices of G_v, the external ports cannot be assigned arbitrary positions, but must stay on the border of v. Additional edge routing mechanisms must be applied to properly connect the external ports with inner vertices.

3.4 The Algorithm

The main phases of our hierarchical layout algorithm with port constraints 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 fi-

nal 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.

3. **Crossing reduction:** Find an ordering of the vertices of each layer that minimizes the number of edge crossings. If the order of ports is not fixed for any vertex, it must also be properly chosen.

4. **Edge routing A:** Depending on port positions, some edges need to be routed around vertices (see Figure 3.3). The number and order of edges that need to be routed on each side of a vertex is determined in this phase.

5. **Node placement:** Determine exact positions of all vertices inside their corresponding layers. The vertices must not overlap each other, the ordering from phase 3 must be respected and the position of each vertex must be well-balanced with respect to its neighbors. We will call this *crosswise* placement.

6. **Edge routing B:** Determine bend points for each edge and the exact distance between subsequent layers, which we will call *lengthwise* placement. Routing to external ports is also handled in this phase.

There are numerous alternative algorithms that can be used for each phase [2, 20, 30], but this report focuses on our current implementation of each phase and on the realization of port constraints.

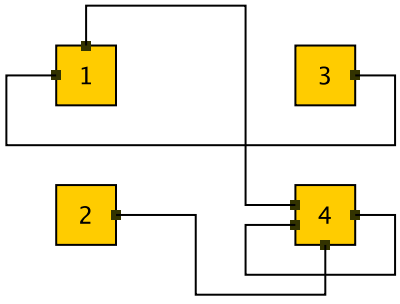


Figure 3.3: Routing of edges around vertices due to prescribed port positions

3.4.1 Implementation

The main class of our layout algorithm is a subclass of AbstractLayoutProvider to match the interface for automatic layout described in Section 2.2.1. Therefore, the input of the algorithm is an instance of KNode (see Figure 2.1), whose direct children represent the graph for which layout is performed.

The phases of the algorithm are modularized using the *Strategy* design pattern [12]: an interface that describes the functionality of the module is created for each phase, and at least one implementation is given for each phase. If there is more than one implementation for an interface, the user may choose from different alternatives for the corresponding phase of the algorithm, possibly leading to differing layout results. With this design pattern it is also possible to experiment with new implementations while keeping the original implementation, and to directly compare their outputs. Figure 3.4 shows a class diagram with all modules of the layout algorithm and their currently available implementations.

Each implementation of a module is a subclass of AbstractAlgorithm, which handles the usage of progress monitors (see Section 2.2.1). These progress monitors cannot only be used to give feedback of the progress of the algorithm during its execution, but can also measure execution times. As seen in Figure 3.5, execution times are tracked for the whole algorithm as well as for each module.

Since for the layered approach the vertices of the input graph are organized in *layers* (see Section 3.4.3), a graph structure that directly expresses this layering is used internally by our layout algorithm. A class diagram for this data structure is shown in Figure 3.6.

The class LayeredGraph, of which there always exists exactly one instance for each run of the algorithm, contains a list of *layers*, represented by the class Layer. A layer contains a list of *layer elements*, and each layer element is assigned to exactly one *linear segment* (see Section 3.4.3). A layer element may have incoming and outgoing connections as well as self-loops, which are represented by the classes LayerConnection and ElementLoop, respectively.

3.4.2 Cycle removal

The goal of this phase is to find a minimal set of edges of a given graph G 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 [14]. A good heuristic is the algorithm *Greedy-Cycle-Removal* from Di Battista *et al.* [2], which 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.

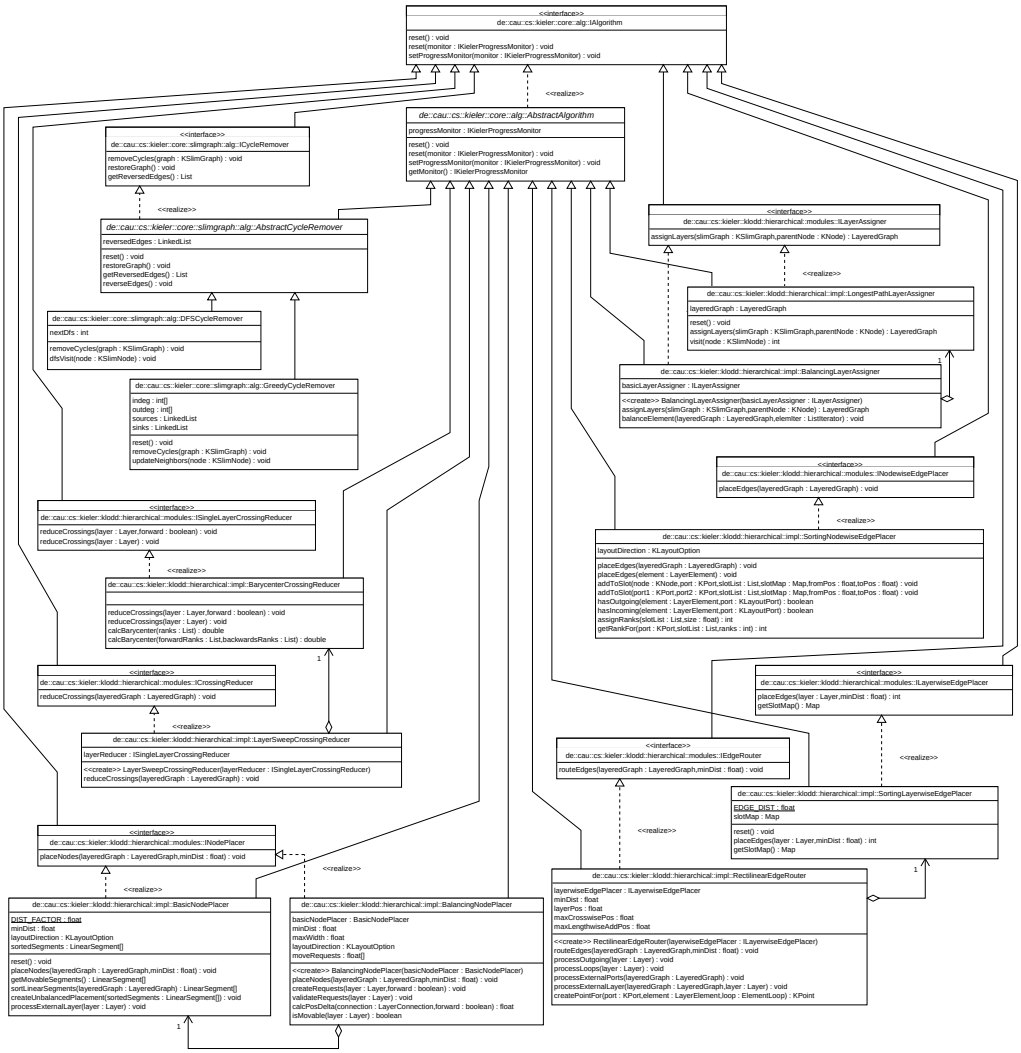


Figure 3.4: Modules for the hierarchical layout algorithm

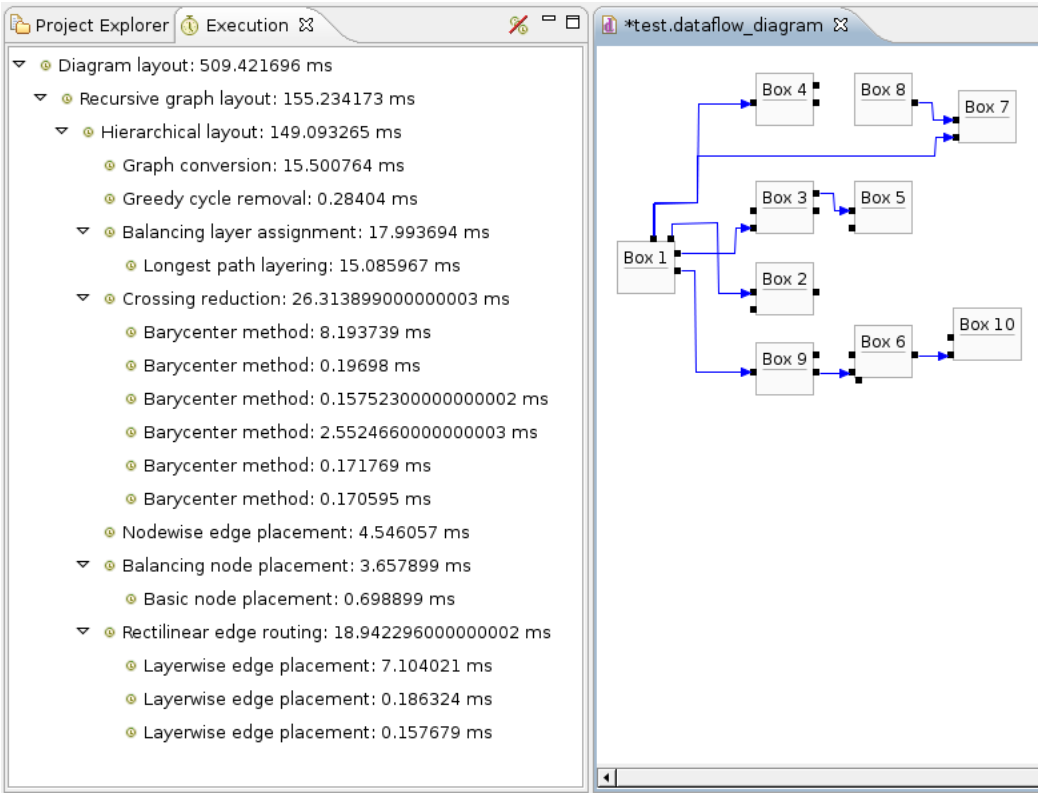


Figure 3.5: Execution time of each module of the algorithm, shown in an Eclipse view after layout was applied to a data flow diagram

Figure 3.6: Central data structure for the hierarchical layout algorithm, representing a layered graph

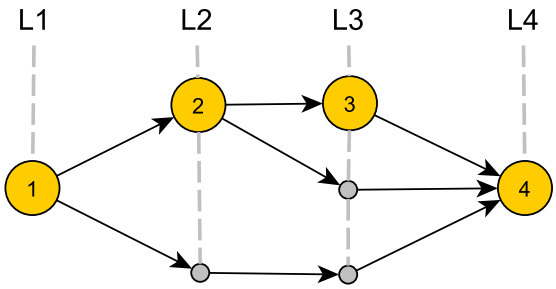


Figure 3.7: A layered graph with two dummy vertices for the long edge (1,4) and one for the edge (2,4)

Since we want to avoid changing the KGraph structure given as input, the graph is first transformed to a much simpler graph structure called SlimGraph, which is not implemented in EMF, but as a set of plain Java classes. Edges are only reversed in the SlimGraph instance for cycle removal.

3.4.3 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 3.7). The *rank* of a layer L_i is $r(L_i) := i$, and its *height* is $h(L_i) := k - i + 1$.

A simple and linear running time heuristic consists in determining the longest path to a sink: all sinks s are put into the last layer, and all other vertices are assigned a layer of height $h(L_i)$ equal to the number of edges on a longest path to a sink plus one. If there are many sinks in the graph, the last layer can become very wide with this layering. Therefore we improve the longest path layering using Algorithm 3.1, which decides locally for each vertex whether moving it to a preceding layer could improve the layering, thus greedily computing a local optimum.

The input of this phase is the SlimGraph instance created for cycle removal together with the original KNode instance. The layering algorithm creates and returns a layered graph, which then requires some post-processing through the method createConnections. This method traverses the layered graph and creates layer connections for all edges that are found in the original graph. This is done by Algorithm 3.2, which also splits connections

Algorithm 3.1: balanceLayering

```
1 procedure balanceLayering(G: directed graph)
2     determine layers L_1, ..., L_k for G using longest path layering
3     foreach layer L_j, j ≥ 3, do
4         foreach v ∈ L_j, indegree of v ≥ outdegree of v, do
5             r := max{i : (u, v) ∈ E, u ∈ L_i} + 1
6             foreach layer L_i, r ≤ i < j with increasing i, until a fitting layer is found,
                  do
7                 if |L_i| ≤ |L_j| then
8                     move v to the fitting layer L_i
9 end
```

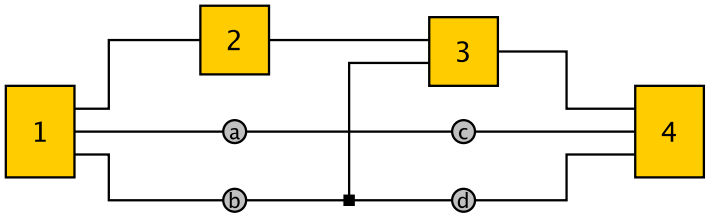


Figure 3.8: The long edges $(1, 3)$ and $(1, 4)$ share the dummy vertex b in layer 2.

that span over multiple layers using dummy vertices, for which new layer elements are created. If two long edges share a common port, thus forming a hyperedge, they must be assigned common dummy vertices, as seen in Figure 3.8.

If the original diagram contains external ports, they are also added as layer elements: input ports, which have only outgoing connections, are assigned the layer with rank 0, while output ports, which have only incoming connections, are assigned the layer with height 0. By this we achieve that external ports can be treated as normal vertices in the following phases of the algorithm, and they are assigned dedicated layers.

3.4.4 Crossing reduction

The problem of crossing reduction for layered graphs, which consists in setting an order of vertices that minimizes the number of crossings for each layer, is NP-complete, even if there are only two layers [15]. 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,

Algorithm 3.2: createLayerConnection

```
1  procedure createLayerConnection(L_1, ..., L_k: layers, e: edge)
2      let L_s be the layer for which v_s(e) ∈ L_s
3      let L_t be the layer for which v_t(e) ∈ 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 =⊥ then
10             get the linear segment S associated with the target port p_t(e)
11         if S =⊥ 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
```

then for each $i \in \{1, \ldots, k-1\}$ optimize the order for layer L_{i+1} while keeping the vertices of layer L_i fixed. Afterwards the same procedure is applied backwards, and it can then be repeated for a specified number of iterations. We will only cover the forward sweep here, because the backwards case is symmetric.

When ports are used to determine the source and target point of each edge, the number of crossings does not depend only on the order of vertices, but also on the order of ports for each vertex. This order is implied by the port ranks which are assigned to the ports using layout options (see Section 2.2.1) and are based on clockwise order. The port ranks must be translated depending on the side of the node, the overall layout direction, and whether a forward or backwards layer-by-layer sweep is performed. For example, if horizontal layout is performed, we must consider clockwise port ranks for a forward sweep, but counter-clockwise port ranks for a backwards sweep.

Based on the translated port ranks we define extended vertex ranks so that for each $v \in L_i$ and $p \in P(v)$ the sum of the rank of v and the rank of p is unique. The *rank width* of a layer element $v \in L_i$ is $w(v) := |P(v)|$ if v originates from a vertex, and $w(v) := 1$ if v was created for a dummy vertex of a long edge or for an external port. The extended vertex ranks of the ordered vertices $v_1, \ldots, v_h$ in the layer L_i are defined as

$$r(v_j) := \sum_{g<j} w(v_g)$$

for all $j \leq h$.

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

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

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

For vertices with FixedSides or FreePorts port constraints we have the additional task of finding an order of ports for each vertex that minimizes the number of crossings. The extension of the method described above is

quite straightforward: instead of calculating values $a(v)$ to order the vertices, calculate values $a(p)$ to order the ports first, then calculate

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

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

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

If there are long hyperedges that share common dummy vertices, as described in Section 3.4.3, crossing reduction must be adapted to avoid inconsistencies in the following phases. If, for example, backwards crossing reduction is performed for the second layer of the graph in Figure 3.8 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 would lead to a crossing of the edges (a, c) and (b, d), which is not allowed for proper vertex placement.

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

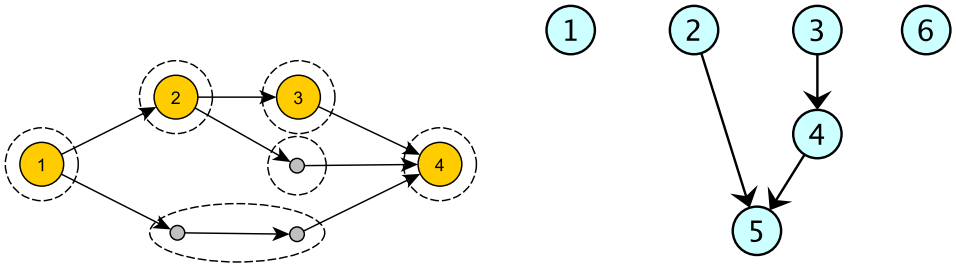
1. For each dummy vertex v_i, $i \in \{2, \ldots, k\}$, only one incoming connection may be considered for crossing reduction, namely (v_{i-1}, v_i).

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

3.4.5 Node placement

From this phase on we will cover only horizontal layout direction, but the concepts for vertical layout are symmetric.

For crosswise vertex placement in horizontal layout the vertices of each layer are arranged vertically. Sander proposes a two-phase method [25]: 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 3.9). 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 $v \in S(v)$.



(a) A layered graph with its linear segments

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

Figure 3.9: Linear segments and their ordering graph

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 $r(v_2) = r(v_1) + 1$. Sander's algorithm 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. Afterwards a *pendulum* method is applied to balance the drawing by moving vertices according to the positions of their neighbors [25, 30]. The exact port positions must be taken into account here to achieve proper vertex placement.

3.4.6 Edge routing

In order to achieve rectilinear edge routing, each edge that cannot be represented by a single horizontal line needs a vertical line segment (see Figure 3.10). 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 $x := x(L_i) + b(L_i) + r(e) \cdot d$, 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. 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 $x(L_{i+1}) = x(L_i) + b(L_i) + (r_{i,\max} + 1) \cdot d$. The set of vertical positions occu-

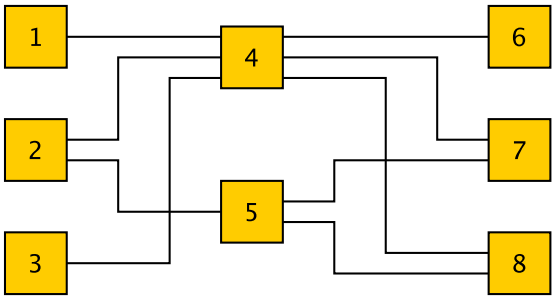


Figure 3.10: Rectilinear edge routing between layers using vertical line segments

pied 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 $Y(e) \cap Y(e') \neq \emptyset$.

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 3.3. For this purpose routing slots of different ranks must be assigned on each side of a vertex, similarly to layer-to-layer edge routing. This is done in an additional phase after crossing reduction; all edges which need additional bend points are processed here, as well as self-loops.

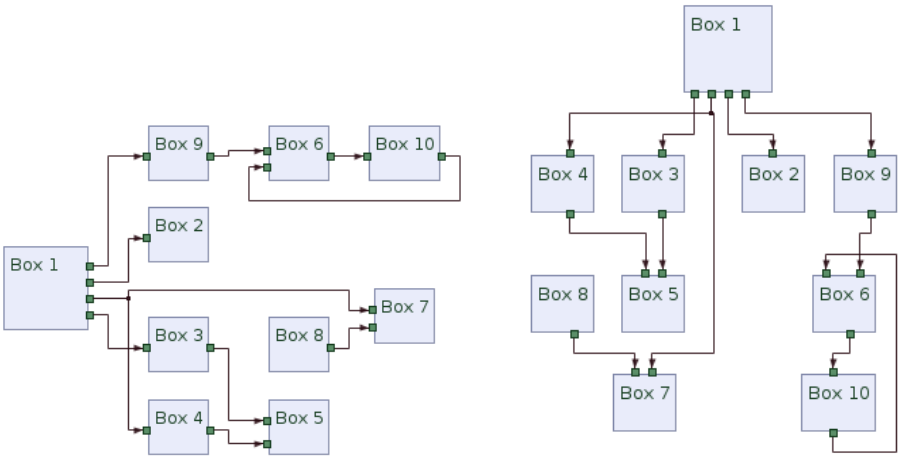
For example, the self-loop $(4, 4)$ in Figure 3.3 is assigned routing slots of rank 1 on the left, bottom and right side of vertex 4, while the edge $(2, 4)$ is assigned a routing slot of rank 2 on the bottom side of vertex 4.

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

3.5 Experimental Results

Here we will look at some direct outputs of our hierarchical layout algorithm. A more detailed analysis of the algorithm is presented elsewhere [30], and more results are shown in Chapter 4, where the algorithm is embedded into Ptolemy.

Figure 3.11 shows how the overall layout direction changes the output of automatic layout. In Figure 3.12 we see a hand-made layout from an



(a) Horizontal layout (b) Vertical layout

Figure 3.11: Output of hierarchical layout with different layout options

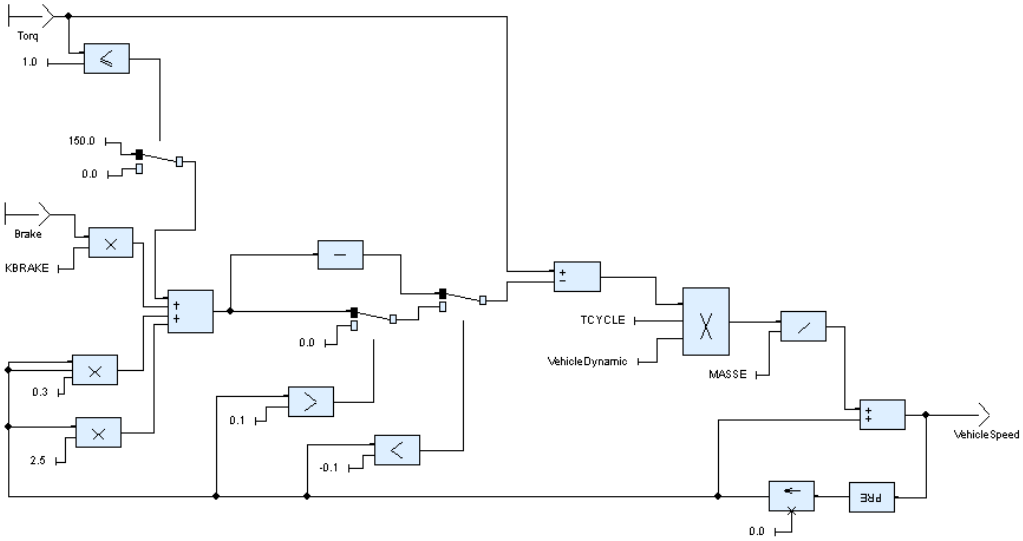
official demonstration of SCADE, and the outcome of automatic layout for the same diagram, which models calculation of the speed of a vehicle for the environment simulation of a cruise control system. This example shows that the quality of our automatic layout is at least comparable with the carefully prepared manual layout. Layout of a compound diagram with connections to external ports is shown in Figure 3.13. Here we see that our algorithm is able to handle edge routing to external ports, even if they are located on the top or bottom side of the parent node.

Measurement data for the execution time of the hierarchical layout method is shown in Figure 3.14. Execution times were determined on an Intel Xeon 3 GHz 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.

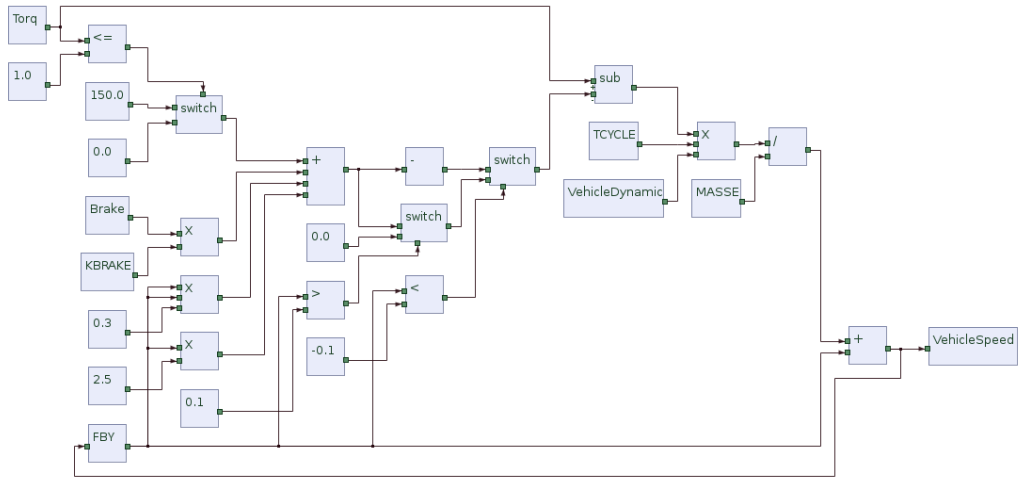
Figure 3.14a presents measurements for generated graphs $G = (V, E)$ with varying $|V|$ and $|E| = |V|$ in logarithmic scale. The curve is roughly linear with an approximate slope of 1.16, hence the overall runtime behavior is nearly linear[3] in the number of vertices. For graphs with about 25 000 or 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 3.14b.

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



(a) Original SCADE diagram



(b) Hierarchical layout

Figure 3.12: Comparison of hand-made layout with automatic layout

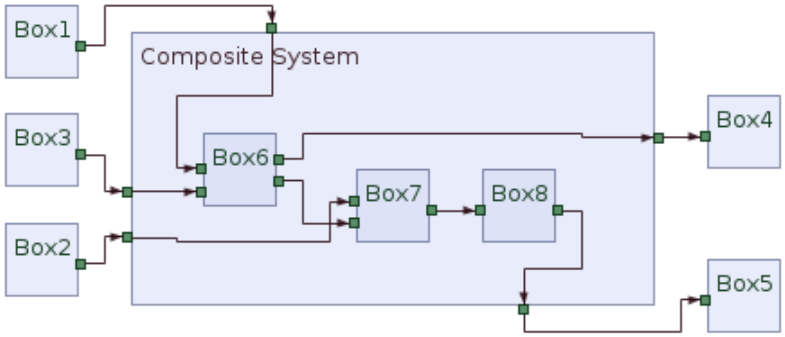
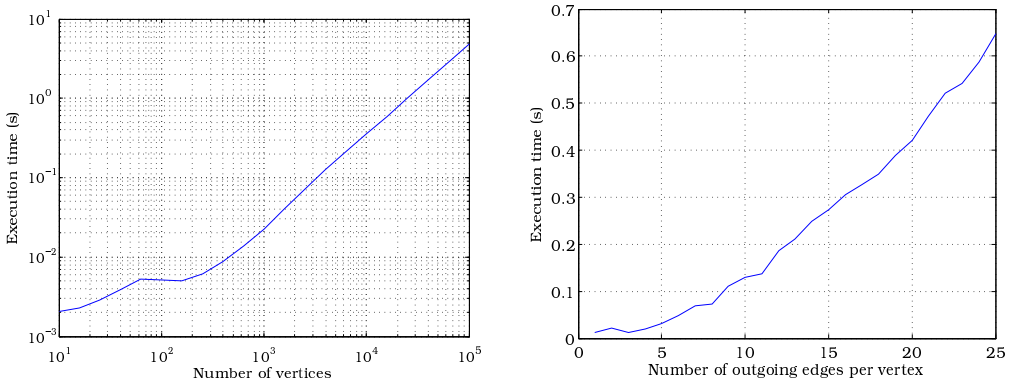


Figure 3.13: Edge routing to external ports



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

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

Figure 3.14: Execution times of hierarchical layout

Here we see that the execution time highly depends on the average vertex degree, since layout for a graph with 2 000 vertices and 2 000 edges is 8 times faster than layout for 100 vertices and 2 000 edges. One reason for this is that for vertices with a lot of incident edges the number of long edges that stretch over multiple layers is likely to be high, so that dummy vertices must be inserted to obtain a proper layering. The consequence is that the problem size rises with regard to the total number of vertices.

Chapter 4

Automatic Layout in Ptolemy

Ptolemy is a graphical modeling suite developed by the Center for Hybrid and Embedded Software Systems (CHESS) of the University of California, Berkeley, USA, under the lead of Edward A. Lee [7].

Ptolemy supports *actor oriented* system design, where the building blocks of a system are *actors*—small software components that consume data tokens and produce new data tokens like functions. Actors can be interconnected to form a whole network of data flow. A *director* is a software component that is responsible to organize the execution orders of the actors. Ptolemy provides a wide variety of director implementations that execute actor models in different ways, usually following specific formal semantics, which are also known as *models of computation* in the Ptolemy notion. Directors have to follow only a few rules and implement a certain interface and hence a model developer is able to create custom directors that execute actor models in any way.

Actors can be implemented directly in some host language, most likely Java, or be composed of another Ptolemy model, *i. e.* another actor network. Each composite actor has its own director to control the model execution of this single actor contents. So a key concept in Ptolemy is to use different composite actors, each using a different director in one model yielding heterogeneous models that comprise multiple models of computation.

One use case could be to model a software controller, with discrete events semantics, and a simulation plant model of the mechanical parts of the system or its environment, with a continuous time semantics that reflects its nonlinearity. This way a whole system can be simulated in Ptolemy prior to its physical integration in the target.

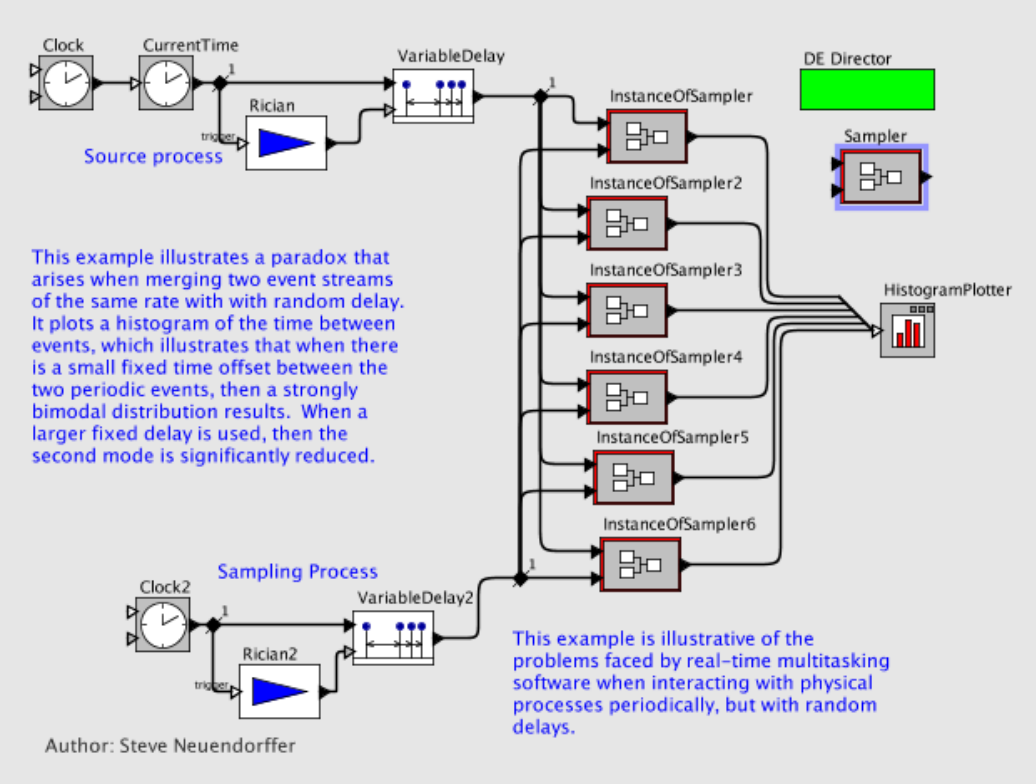


Figure 4.1: A graphical representation of a Ptolemy actor model

4.1 The Ptolemy Layout Problem

Ptolemy models can be created either programmatically by its Java API, manually by writing the model specification in an XML language called Model Markup Language (MoML) or by drawing a graphical representation of a Ptolemy model in a diagram editor called *Vergil*.

A typical graphical Ptolemy representation is shown in Fig. 4.1. Actors are represented as rectangles with some actor-specific icon. Actors produce or consume data on *ports*, either inputs or outputs or both, represented as a small triangle at the border of an actor icon. Rectilinear polylines connect ports with each other where usually a simple port may only be connected to exactly one connection, semantically called a *relation*. A connection can be branched by introducing an explicit *relation vertex*, a small black diamond icon, to which multiple connections may be attached.

As Fig. 4.1 also shows, there can be other components in a Ptolemy model that do not need to be explicitly connected to some other components. The

most prominent one is the *director*, represented by an unconnected labeled box. Documentation blocks or more generally *text attributes* can be placed in a model at arbitrary positions. There are quite a few others like these in the Ptolemy libraries available.

4.1.1 Node Placing

In Ptolemy all aforementioned nodes can be placed manually, *e. g.* actors, directors, relation vertices and text attributes. The horizontal and vertical coordinates of all nodes are persistently stored and are part of the Ptolemy model. These locations can be adapted programatically and hence can be used by the layout algorithm.

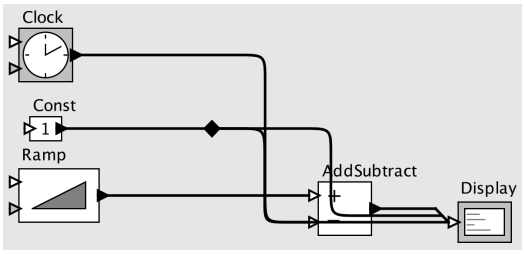
Most iconed boxes such actors, vertices, and directors have a fixed size, which can be changed neither manually nor programmatically. The size is set once for the node by its specification from its icon and settings which might be stored in the Ptolemy node library. The size of text attributes is variable by the length and wrapping of the text. The user can influence the shape of text boxes by the text's length, its wrapping (which must be manually set) and font size and style. Automatic wrapping to a given text box width is not intended. Actors also have a label, its name, which can be arbitrarily customized by the user. It can be arbitrarily wide and high by using line breaks. An outer bounding box for nodes is given by Ptolemy including all elements of the node, *e. g.* an actor's ports and its label. Therefore in general the size of an actor is variable but given by a specific model.

Structural hierarchy is an important concept in Ptolemy but is not expressed directly in the graphical representation like in Sec. 2.2.2. Contents of composite actors like the *Sampler* in Fig. 4.1 is not shown in the same diagram but can be opened in a completely new canvas. So layout of hierarchy is not in the scope of layout in the Ptolemy Vergil editor.

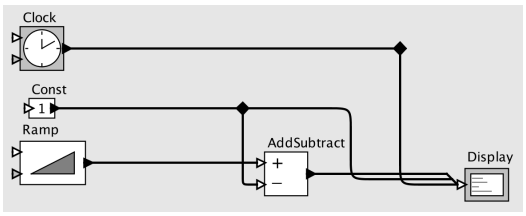
4.1.2 Connection Bend Point Placing

Placing of bend points is an issue in Vergil, because bend points are *not* part of the Ptolemy model and hence are not persistently stored, and for a user it is not possible to directly influence the locations of connection bend points, neither manually nor programmatically.

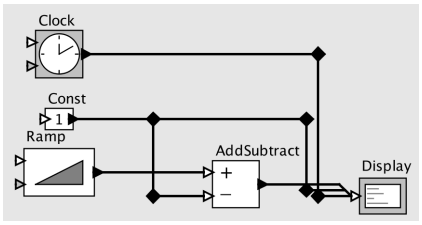
Ptolemy uses an internal connection router that dynamically computes bend points for a connection between two endpoints. This router is a rectilinear style Manhattan router without obstruction avoidance. Hence given an arbitrary node placing, it is likely to have an arbitrary number of overlappings of connections with nodes and also with other connections.



(a) Arbitrary model with node and connection overlappings



(b) Optimized layout by introducing an additional relation vertex and moving nodes



(c) Inserting one relation vertex per bend point results in full control over connection routing

Figure 4.2: Connection Routing in Ptolemy

However, there are some options for influencing the routing shown in Fig. 4.2. Depicted in Fig. 4.2a is a placement of nodes for which the Ptolemy router results in two overlappings of nodes and one overlapping of connections. The latter results in an ambiguous diagram, where it cannot be decided which connection enters the *AddSubtract* actor and which enters the *Display* actor.

Node Moving Routing is usually influenced manually by moving nodes. If this is not sufficient, new nodes can be inserted, which then can be placed according to the desired layout. Relation vertices are nodes that can be added within a connection without changing the semantics of the model. Hence insertion of a few relation vertices and optimizing the placement of all nodes is the usual way of manually creating collision-free connection routings as shown in Fig. 4.2b.

This has also been used by the original author of the Ptolemy model in Fig. 4.1, where the actor instances of the Sampler in the center are slightly offset in order to get an unambiguous connection routing to the HistogramPlotter. Additionally there are two routing slots created for the two inputs of all of the Sampler instances by adding a relation vertex and placing them also with a little distance at the horizontal coordinate.

Vertex Insertion To get full control over the connection routing, the Ptolemy Manhattan router can be handed trivial routing tasks by creating only straight connection pieces. This can be achieved by insertion of relation vertices for every bend point in the diagram. An example is shown in Fig. 4.2c.

The advantage is that one gets full control over the bend point placement as there are no real bend points but only vertices which can be placed. In spite of that the drawbacks are obvious. The most visible is that the diagram gets less appealing by this crowded view. The other is that only for the goals of layout the underlying semantic model gets heavily changed by insertion of new semantic objects. Even while this has no semantic implications, the semantic model gets crowded.

One way to make the graphical representation more appealing would be to simply hide the helper vertices such they do not get drawn at all. However, this would lead to an odd state of the diagram where following manual user interaction—like manually moving things—would be very difficult and unintuitive.

Improved Manhattan Router Another idea is to improve the functionality of the built-in Manhattan router of Ptolemy to also avoid obstacles and minimize edge crossings like the results of the algorithm presented in Sec. 3.4.6.

However, the problems compared here are totally different. The routing algorithm presented in this paper is highly interweaved with the placing problem of nodes and therefore can exploit the already prepared routing slots of the earlier layout phases. This way the routing problem can be coped with well, while a general connection router that is separated from the node placing steps is much more complex: Given an arbitrary node placement, the calculation of bend points has to take into account all other nodes, while space everywhere between the nodes is limited. Hence the general routing problem is much more complex than the combined routing problem.

Developing a general stand-alone router is out of the scope of this work and might be approached in the future in another context.

Bend Point Router The last alternative would be to replace or upgrade the Ptolemy Manhattan router in order to graphically apply a given precomputed set of bend points to the connections. Then the calculated bend points of the algorithm presented here could also be applied to

the Ptolemy diagram. This would be the simplest and most appealing way to get the desired result.

Technically, however, this would imply a few basic changes in the Ptolemy infrastructure. First, this would require means to persistently store bend points related to a connection. For example the MoML would have to save the bendpoints related to connection pieces, which themselves are also not really explicit parts of the Ptolemy model and would need to be introduced there.

Then, after a layout run of the auto layouter, the automatically generated bend points should be used while there would need to be a fallback to the old Manhattan router whenever the user manually starts to drag single items around again, unless manual changes in an auto-layouted diagram should be prohibited at all.

We have implemented a two-level approach for the KIELER and Ptolemy interface. The first simple and non-invasive mode simply places nodes and does not touch connections. Hence the internal Ptolemy router is used to route the edges which is likely to produce overlappings. The second mode also routes edges by the VERTEX INSERTION method to showcase the routing capabilities. This involves a lot of hard-coded model transformations that introduce new relations and removes others while trying to keep the semantics as before. Methods to remove or hide unnecessary relation vertices help to avoid a cluttered mess of relation vertices.

4.2 Mapping the KIELER Layout Problem to Ptolemy

Employing the KIELER layout algorithms to Ptolemy seems to be a straightforward mapping. Unfortunately it is not due to some subtle differences in the two layout problems. These will be discussed in the following.

4.2.1 Abstraction

An issue in modeling tools like Ptolemy is that in their implementation they try to follow some abstraction rules to separate concerns and to hide implementation details of lower levels in the higher ones. In general this is a good idea as long as the intended application on higher levels does not demand any information or API of lower levels.

In graphical modeling the main user interaction mechanism still is WYSIWYG drag-and-drop editing and so most tools are designed only for this purpose. There are high-level interfaces to manually move graphical items one by one and the feedback is given by the graphical representation to the developers eyes directly. Usually—and this also holds for Ptolemy—the details are not available in the public APIs. For example it is not always trivial to obtain the actual structure and layout of a diagram programmatically in actual coordinates, because some location functions are not stated public.

In Ptolemy there is such abstraction between the underlying graphical drawing framework called *Diva*[1] and the more specific Ptolemy II Vergil editor. Some information was originally hidden in protected APIs, such as the orientation of ports, and had to be made public for this work. Other information is not consistent and needs many special case handling in the implementation for reading and reapplying the layout to the diagram. For example locations in horizontal and vertical coordinates are handled differently in Diva and Ptolemy and even within Ptolemy differently for actors and text attributes. So bounds in Diva give the location by their top left corner (just like in KIELER) while Ptolemy uses the center point for most items, except for text attributes where it is again the top left corner.

Usually in a drag-and-drop editor the user does not need to care about such issues because movements are done relative to current coordinates. But for automatic layout this makes the interface code more verbose and error-prone.

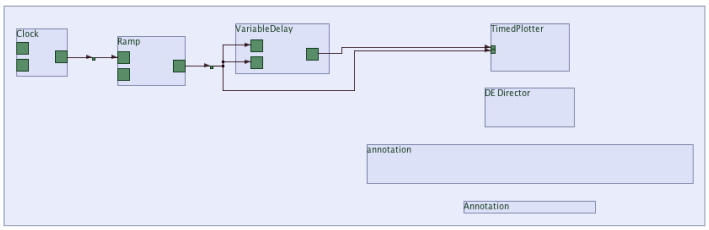
4.2.2 Nodes

Nodes in Ptolemy can comprise multiple elements like an icon, text, and ports, where the icon usually does not fill the whole bounds of the actor. Its text, *e. g.* the actor name, and the ports extend the bounds, sometimes by a significant amount. Hence to reserve enough space in the layout for placing the nodes, the overall outer bounds are used for the sizes of nodes in the KIELER KGraph. This results in a KGraph where the ports are always fully covered by the KIELER node's bounds. This is no problem for the layout algorithm as long as it knows to which side of the port—North, East, South, or West—the port belongs. This information must be read from the Ptolemy diagram.

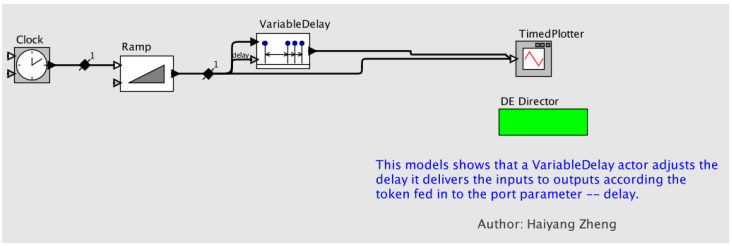
Next to actors with ports there are other nodes which are connected but have no ports at all. These are for example internal ports of composite actors.

It can be argued how to treat relation vertices. Following the VERTEX

[1] `http://embedded.eecs.berkeley.edu/diva/`



(a) Internally computed layout



(b) Ptolemy diagram

Figure 4.3: Ptolemy hierarchical layout including unconnected nodes results in unappealing stacked views.

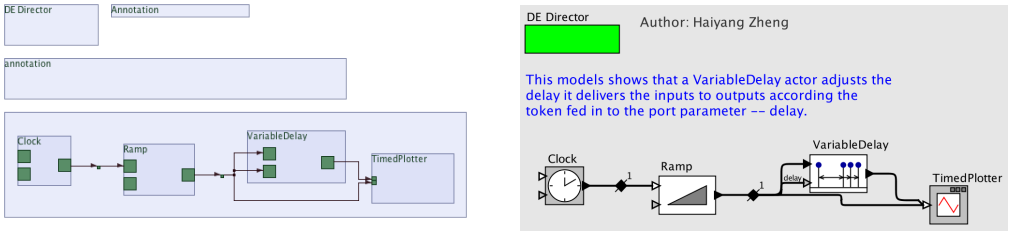
INSERTION strategy of above would regard relation vertices as connection bend points and not as nodes. For the first approach to Ptolemy, no connection routing is applied at all and the internal Ptolemy Manhattan router does the job. Here, relation vertices are treated as usual nodes with a size of zero for simplicity.

4.2.3 Block Layout

A Ptolemy diagram not only consists of connected nodes but also comprises unconnected decorating nodes such as text boxes for documentation, attributes like a director, and other such items.

Especially text boxes for documentation tend to get quite large as Fig. 4.1 shows. Processing them in the presented hierarchical layouter places all unconnected items in the last layer in a large pile. If the texts are quite wide, the whole layer is stretched to the size of the largest item. This leads to unpleasing results like shown in Fig. 4.3.

We improved the situation by employing the structural hierarchy (composite nodes) mechanism of KIELER, which allows for every node in the graph to contain subgraphs. The feature that each subgraph can be laid out with a different layout algorithm allows us to handle connected and unconnected nodes differently.



(a) Internally computed layout (b) Ptolemy diagram

Figure 4.4: Separated hierarchichal layout for connected nodes and a simple block layout for others.

All connected nodes are put in one composite node, which is layouted with the hierarchical layout algorithm. All other nodes are layouted together with the composite node by a simple block layout heuristic. The result is shown in Fig. 4.4. The block layouter assumes that the composite node containing the connected nodes is the largest node and then arranges all other nodes on top of that next to each other and wrapping at the composite node in size increasing order. It allows to set priorities to nodes which is used to always position the director of the node in the top left corner. For many diagrams this simple heuristic reveals acceptable and especially much more appealing results than the hierarchical layouter alone.

The simple block layouter could be improved to solve the block layout problem more generally, *i. e.* place a set of unconnected nodes with fixed sizes optimally. To some extent this relates to Harel's Blob layout [17]; however, Harel assumes that the sizes of the nodes are variable, which is not the case here.

A drawback arises when modelers use *secondary notation* by expressing something with semanticless means. In textual languages different ways of indentation are secondary notation which in that case can show block scopes while it is ignored by a compiler. In Ptolemy it is common to use text attributes to give a short documentation to specific graphical parts on the canvas. Therefore the text attribute is usually placed close to the nodes that are to be explained by the text; this placement is also a form of secondary notation. There is no semantic link that can be made between the text and the node and hence no automatic means can take this relationship into account.

For this case we implemented different versions of the layouter such that it can either layout all nodes with the above described combination of hierarchical and block layouter, or it can place only all nodes that are connected and all other nodes (*e. g.* director, text attributes, or parameters) stay un-

touched.

This way the developer can run the layout multiple times and manually place unconnected items for documentation in whatever way. After following layout runs these will stay at the respective locations.

4.2.4 Graph Direction

The hierarchical layout approach is designed for directed graphs only and uses the direction information of edges to place the nodes onto the different layers. This usually results in drawings with the major direction of data flow from left to right for horizontal layout.

Although in general Ptolemy is a data flow language that transfers data tokens between ports, the direction of flow is not necessarily unambiguous or enduring.

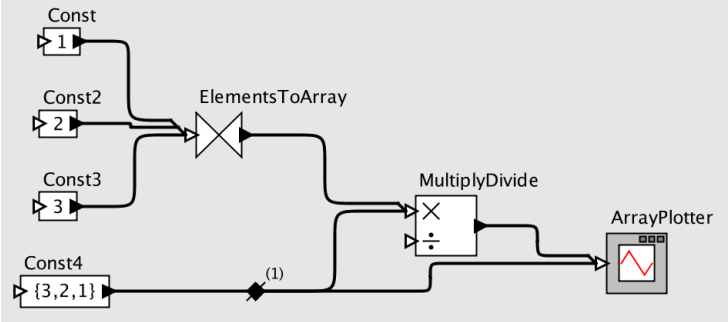
First, the low level graphical representations of single links between ports or relation vertices have directions not related at all to the Ptolemy flow of data. As connection figures themselves show no arrow heads, this is of no relevance but just implies that from the direction of a graphical edge the flow direction cannot be deduced. Hence one needs more information about a whole connected set of relations and what kind of ports—source or target—their endpoints are. Then the direction of such a relation set resp. low level edge set can be computed.

Second, the direction of dataflow can change in a Ptolemy diagram. Flow of data is done in single steps by passing single data tokens between ports. A port of an actor can be both, input and output port. While a port cannot produce and consume data at one port at the same time, it can do it interleaved. Hence connections between ports might be bidirectional.

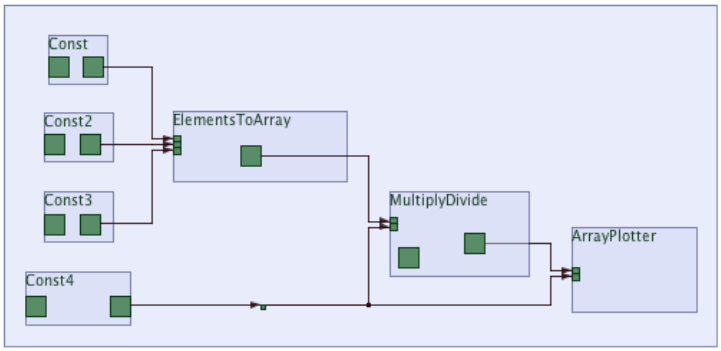
As this is not a very typical way to model in Ptolemy, the direction in a connected relation set is approximated by a simple heuristic that searches for the first source port in the set and uses this to determine the direction for the layout algorithm.

4.2.5 Multiports

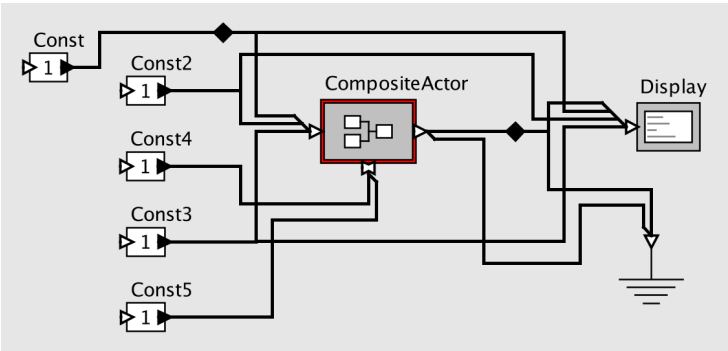
Ports in Ptolemy can be of different kinds. Simple ports just take or produce up to one connection while *multiports* (carrying a white port icon) allow multiple connections. The multiple connections get ordered and can be accessed by the owner actor by index of the so-called *channels* of the multiport. Order of the channels is determined by the temporal order the edges were connected to the port. Graphically the order is presented in Vergil as shown



(a) Multiports in Ptolemy take multiple incoming connection in a specific order.



(b) In the KIELER datastructure this gets mapped to a set of small ports with a small offset each.



(c) Examples of routing with hidden vertices to all directions of multiports

Figure 4.5: Multiports in Ptolemy get represented by sets of ports in KIELER to help avoiding additional connection crossings.

in Fig. 4.5a. Hence the order of the nodes in the layers might introduce additional connection crossings for nodes connected to the same multiport.

Like shown in Fig. 4.5b each Ptolemy multiport gets mapped to a set of multiple small ports in the KGraph datastructure. The small helper ports get shifted a bit according to how the Ptolemy connections are fanned out. This emulates the order of the connections, and the nature of the hierarchical layout approach will try to avoid additional edge crossings in the crossing reduction phase described in Sec. 3.4.4.

4.3 Experimental Results

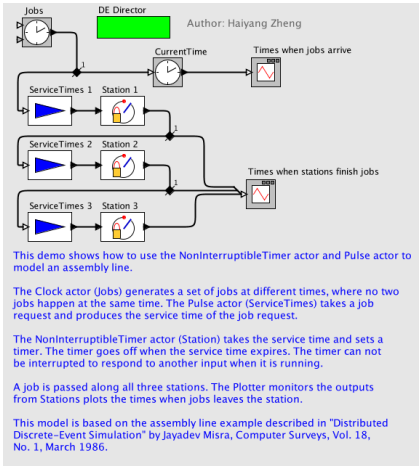
In the following we depict some example layouts. The models are mainly taken from the official list of Ptolemy demos accompanying the Ptolemy tool. We show the original layout and different results of the layout algorithm. This is the placing of all nodes (connected and unconnected but without routing) and a version of the connected part where routing was done by introducing helper vertices which get hidden afterwards. Sometimes we also show the internal KGraph datastructure which is the direct output of the layout algorithm.

In general the hand-made original output is expected to be the best, because the official Ptolemy demos are showcases caringly designed to be presented to public especially by expert Ptolemy users. So one can expect that enough time was spent to make the layout sound. This might be the biggest drawback that some developer has spent some considerable effort to create that layout while with automatic layout, it is only one click away.

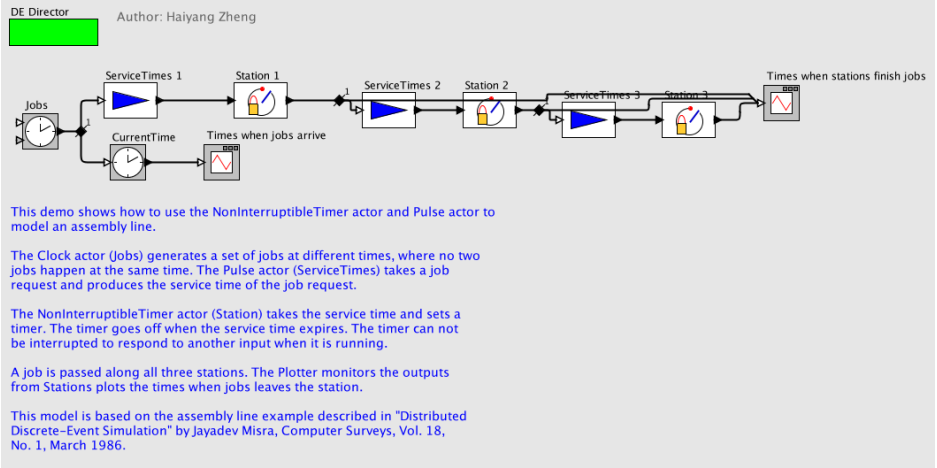
Unfortunately as discussed above, the routing of edges cannot be applied directly to Ptolemy. Hence the applied placing of nodes to Ptolemy usually looks much worse than the placed and routed images, because the internal Ptolemy Manhattan Router does not avoid any overlappings of connections with other connections or nodes. The routed images are transformed to a considerable amount by inserting new relation vertices for the routing. A developer has to decide whether this transformation is acceptable only for layout purposes. However, it best showcases the power of the layout algorithm itself.

Although the Ptolemy models have very different overall sizes, they use composite nodes to reduce the amount of nodes on every single canvas. Hence almost all models in the Ptolemy demos have about the same size on one hierarchy level. As Ptolemy does not yet support to directly visualize nested models, the compound graph feature of our algorithm cannot be used. Therefore very big examples would be quite artificial. However, Section 3.5 presents

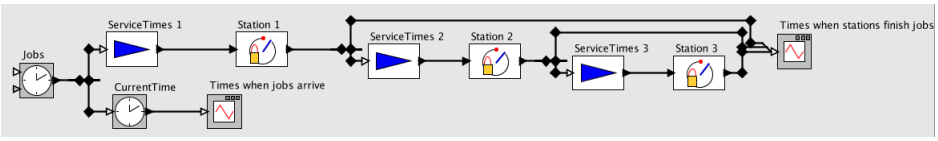
experimental results not specific to Ptolemy where we also consider arbitrarily large randomly generated diagrams.



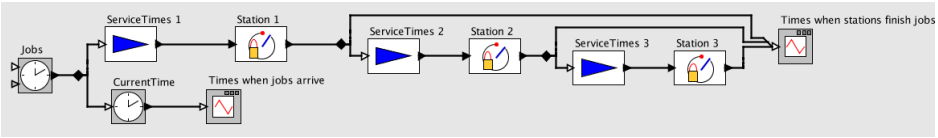
(a) Original



(b) Only placed all nodes

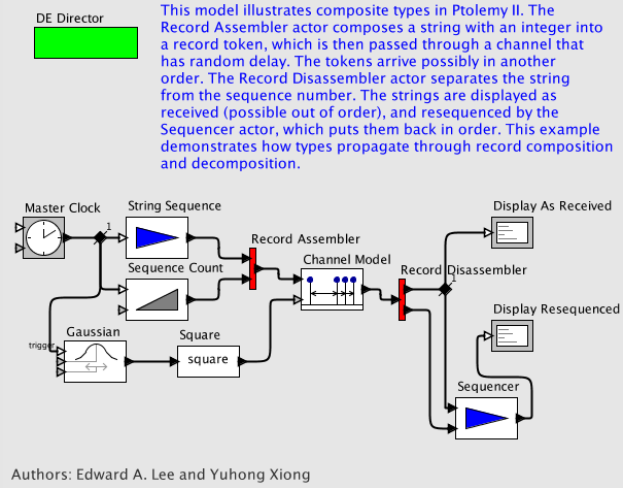


(c) Placed and routed connected nodes with helper vertices

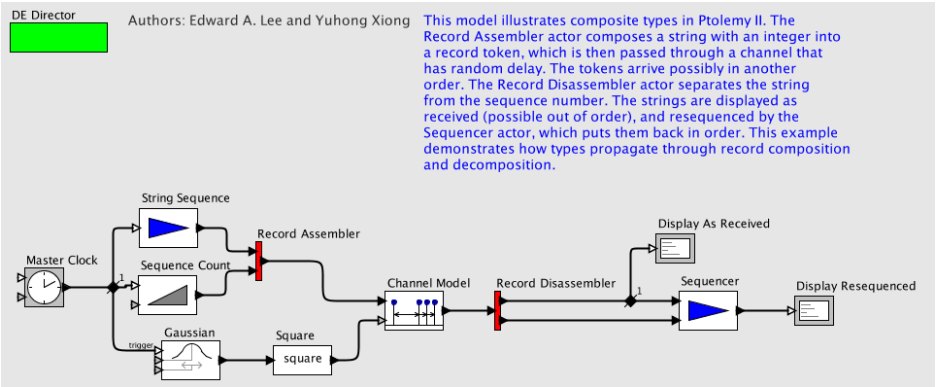


(d) Placed and routed with hidden vertices

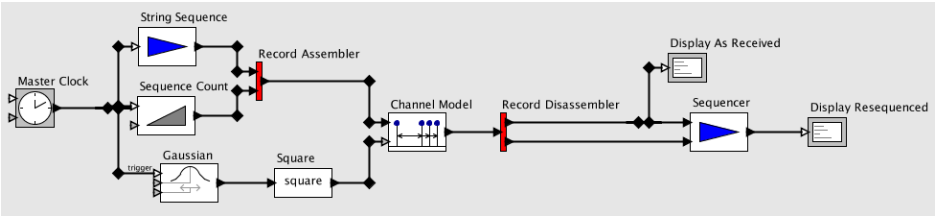
Figure 4.6: AssemblyLine: This is an acyclic, fairly sequential model that results in a wide horizontal span.



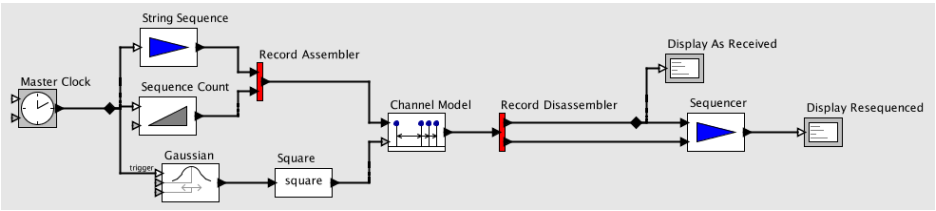
(a) Original



(b) Only placed all nodes

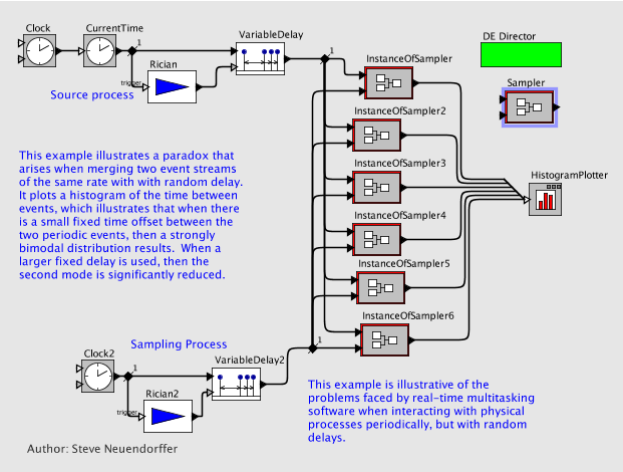


(c) Placed and routed connected nodes with helper vertices (director and notes omitted)

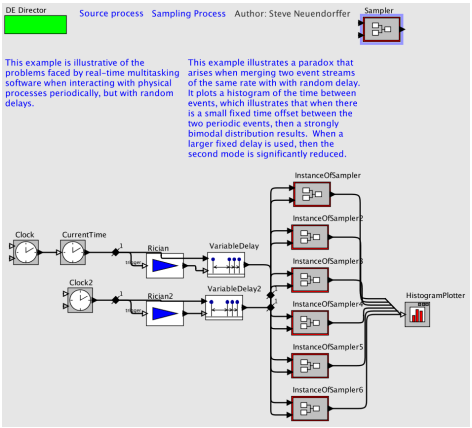


(d) Placed and routed with hidden vertices (director and notes omitted

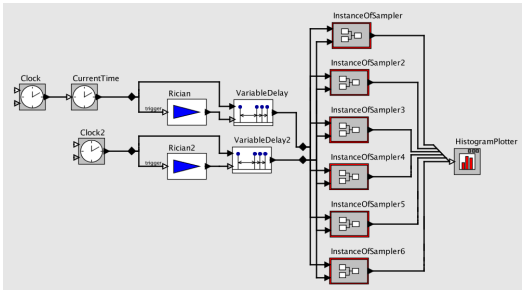
Figure 4.7: Router: A model where the manual layout is quite optimized and packed. In the generated layout some rather small nodes have long labels and hence use much space in their layer, *e. g.* the Record Assembler/Disassembler.



(a) Original

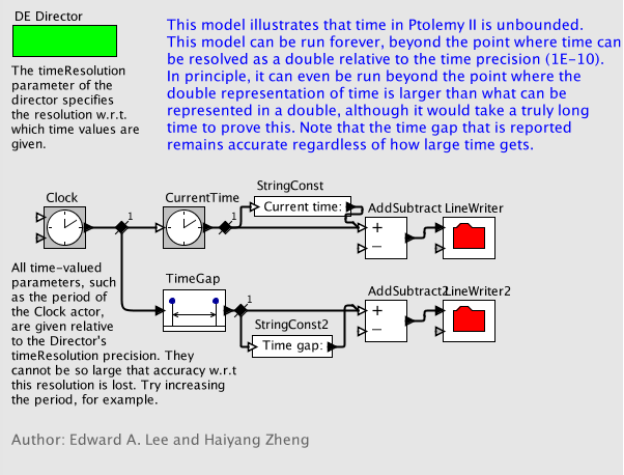


(b) Only placed all nodes

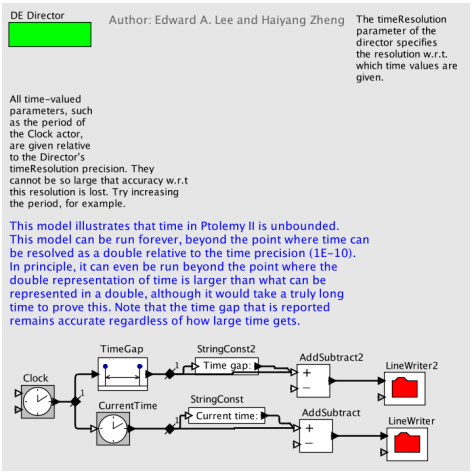


(c) Placed and routed with hidden vertices

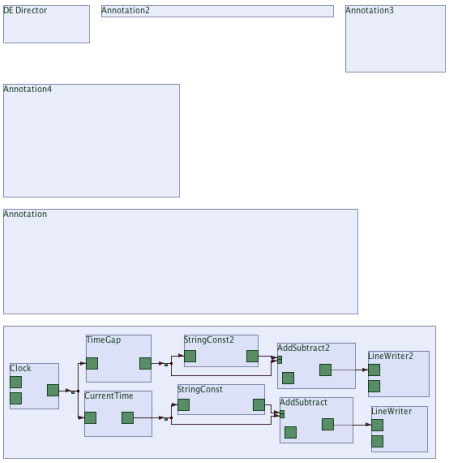
Figure 4.8: TimingParadox: In the original layout the author slightly moved all nodes to reveal a clean connection layout. Without setting bend points explicitly in the auto-layouted version, the result shows many connection overlaps. Setting bend points by relation vertices gives a clear routing. However, considering relation vertices as regular nodes results in a suboptimal vertex placement, because junction points of hyperedges are not shown correctly



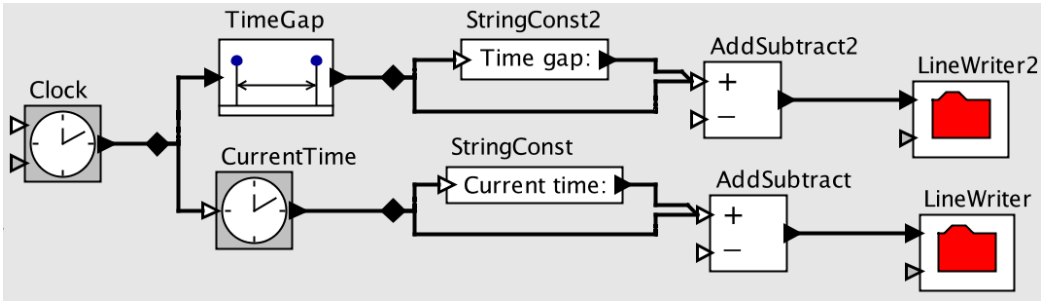
(a) Original



(b) Only placed all nodes

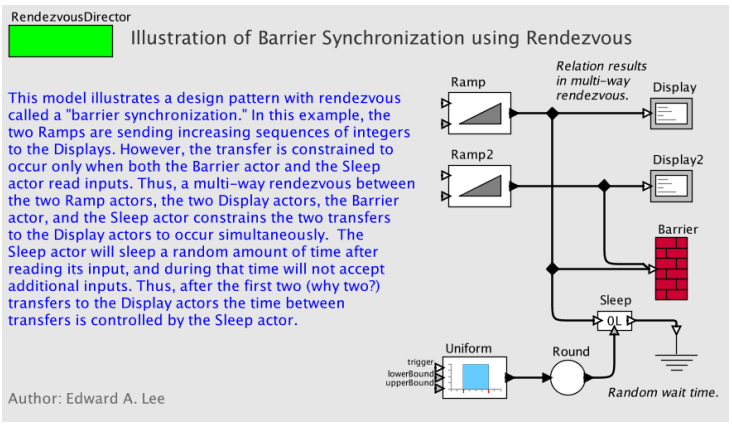


(c) Corresponding KGraph

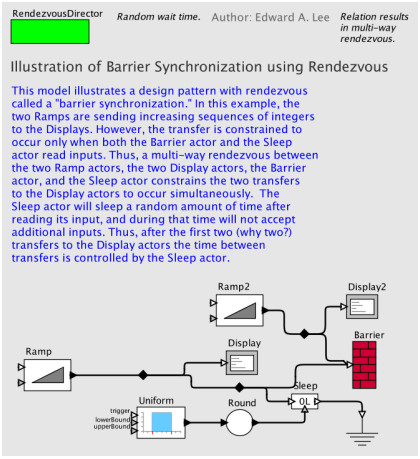


(d) Placed and routed with hidden vertices

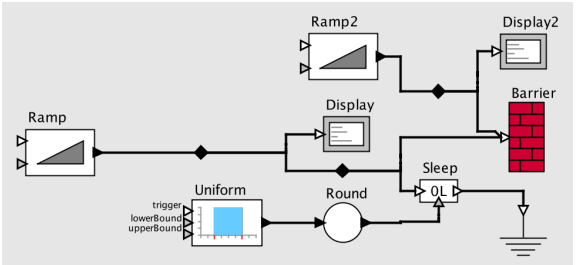
Figure 4.9: LongRuns: The box layout of the text annotations is suboptimal. In the original the lower left text box is wrapped to get a specific shape of the text to get a compact (overlapped) layout. Again, the explicit routing helps to avoid overlaps.



(a) Original

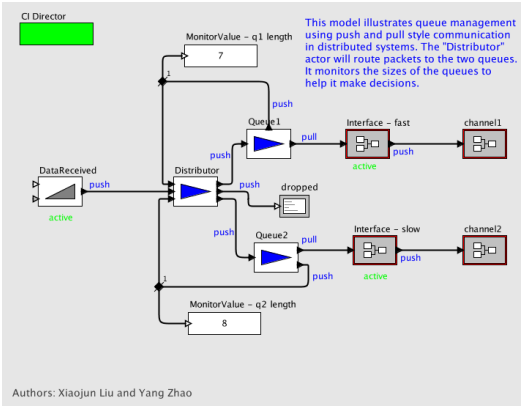


(b) Layouted

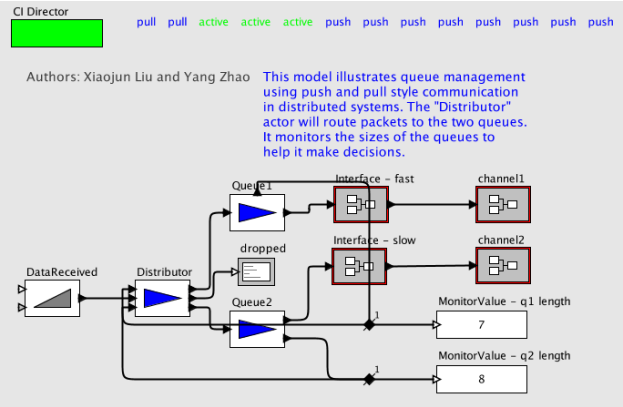


(c) Placed and routed with hidden vertices

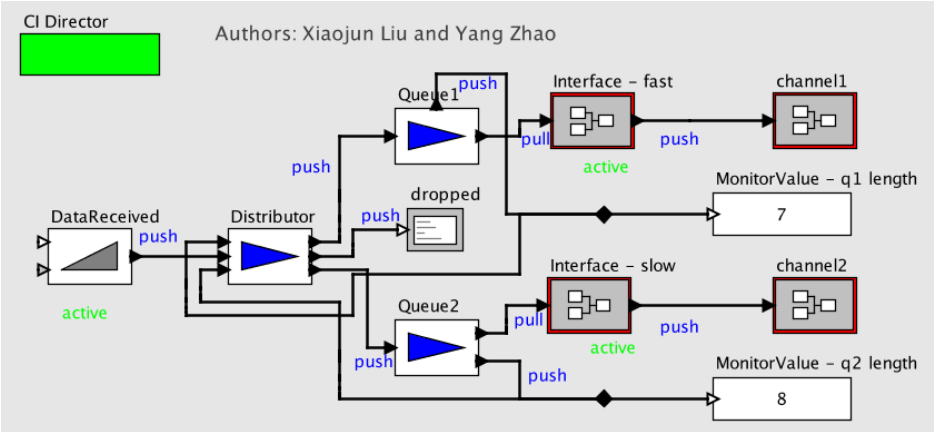
Figure 4.10: Barrier: The connected components are clearly laid out without overlappings. While the text box layout looks alright, the author of the original has placed text nodes next to connections, which give an information about the corresponding relation. In the structure of the diagram this implicit connection between the objects is not visible. Hence the layout algorithm cannot take that information into account while laying out items. Therefore, in the auto-layouted version the context sensitive text attributes loose their context.



(a) Original

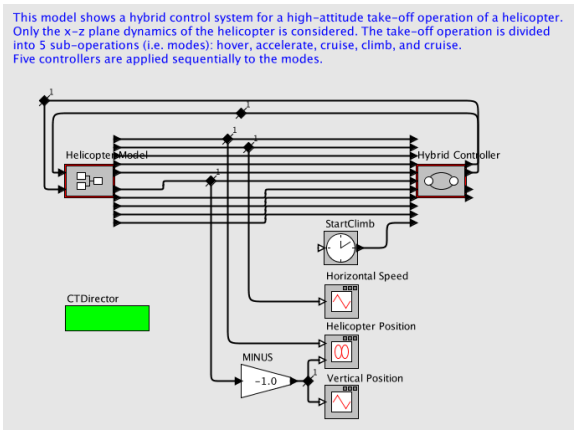


(b) Layouted

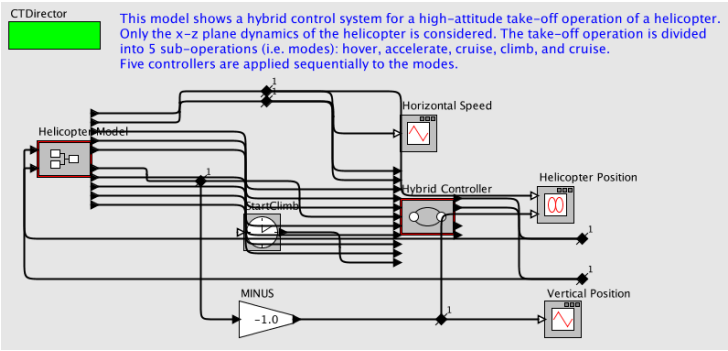


(c) Placed and routed with hidden vertices with manually placed text annotations

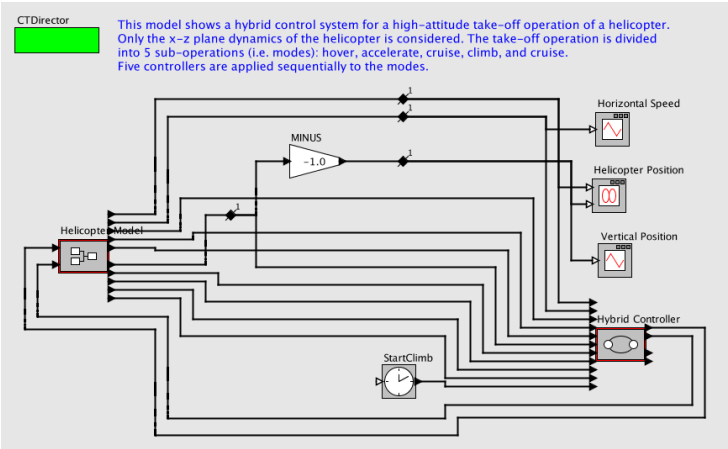
Figure 4.11: CI-Router: While the node placing is good, the Ptolemy Manhattan Router produces bad connection-node overlappings. Here, again, the text attributes placed next to graph items totally loose their context. Here it helps to only place (and route) connected nodes while unconnected such as text attributes are left untouched. This way the user can manually place text attributes to document special parts of the model.



(a) Original

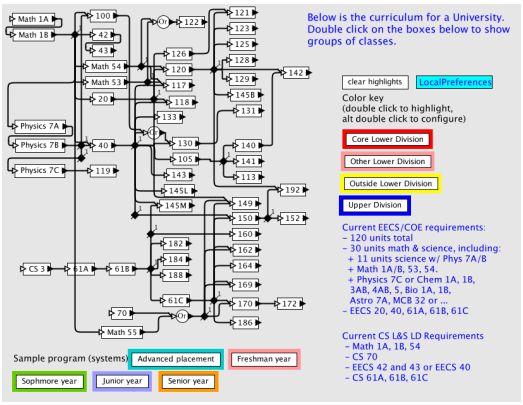


(b) Only placed all nodes without removing unnecessary vertices

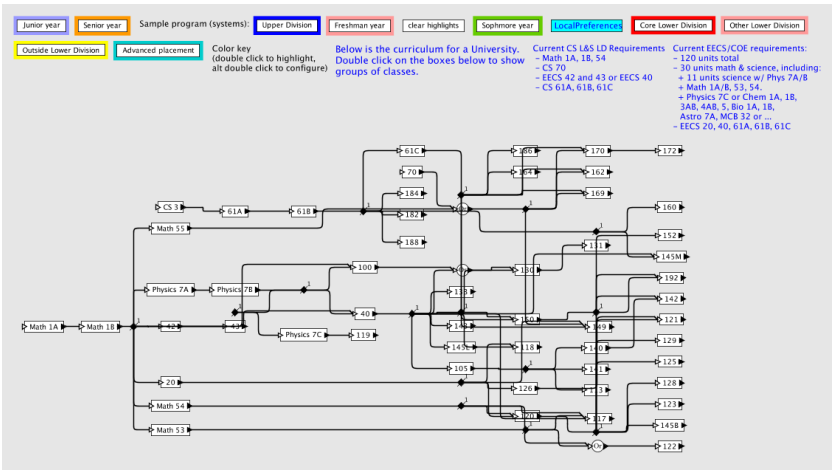


(c) Placed and routed with hidden vertices

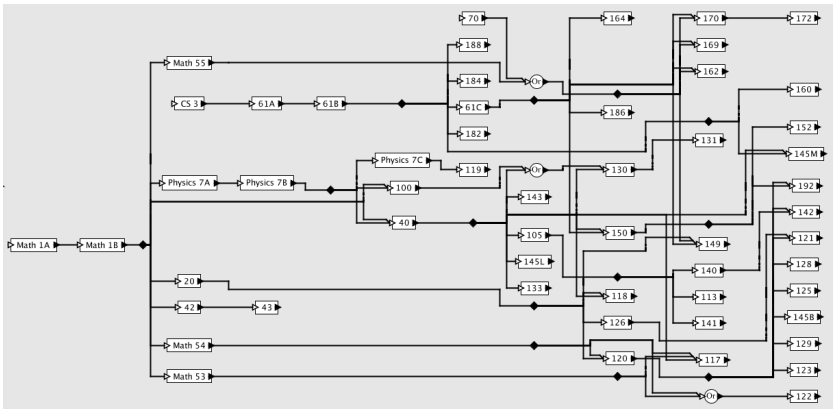
Figure 4.12: HelicopterControl: This is a model with many parallel connections. They get routed around some nodes, which the Ptolemy Manhattan router ignores and produces a tangled mess of wiring. Additionally, the original author used relation vertices to enhance the original layout. A simple placing without removing unnecessary relations keeps the vertices—although it is clear that they loose their original intention—and regards them as usual nodes. Relation vertices might become sinks when one of their incident edges is reversed in the cycle removal phase of our algorithm, which is suboptimal. Removing such vertices completely before layout and routing results in a clear drawing.



(a) Original



(b) Only placed all nodes



(c) Placed and routed with hidden vertices

Figure 4.13: Curriculum: This is a model with many small nodes. The placing looks quite good, although we ourselves can hardly judge whether the original is done following any semantic secondary notation. The diagram becomes wider, but better reflects the order of the nodes. However, while creating the KGraph took about 0.3 s, running the layout algorithm 0.06 s, applying the layout for these 88 graphical items took about 12 minutes using the official Ptolemy MoML change requests. This is likely some bug, as for all other models, the time for applying layout was in the same range as the time for creation of the KGraph.

Chapter 5

Conclusion

We have presented KIELER, a project on the pragmatics of graphical model-based design, with regard to its interface for automatic layout of diagrams. Automatic layout can be applied to diagrams of Eclipse GMF editors, which constitute a large set of graphical editors in Eclipse. Additionally, we implemented an interface for automatic layout in Ptolemy, a project on the design and simulation of embedded real-time systems with different models of computation. We introduced the layered approach for graph drawing with our extensions to handle the special constraints of data flow diagrams.

Experimental results show that our algorithm is able to yield well readable layouts of the data flow diagrams of Ptolemy and is very fast if the average vertex degree is not too high. However, the graphical user interface of Ptolemy does not allow the specification of bend points for edges, but relies on a separate edge routing algorithm. For this reason the output of our layout algorithm cannot be directly applied to Ptolemy diagrams yet. As a work-around for this problem, we enforce the proper routing of edges by using relation vertices.

To improve the layout functionality of KIELER, we plan to further optimize the hierarchical layout algorithm and to investigate other approaches of graph drawing. Furthermore, Ptolemy and its graphical editor Vergil could be extended for better support of automatic layout. Specifically, it would be desirable

1. to add direct support of bend points for edges, including their persistent storage in MoML files, and

2. to be able to link textual comments to specific elements of the diagram in order to place them near their corresponding element.

Alternatively, a new graphical editor for Ptolemy could be implemented in

Eclipse using GMF. In this way the editor would directly benefit from all available features of GMF, including automatic layout enabled by KIELER.

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. ACM, 1985.

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

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

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

[5] 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.

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

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

[8] Thomas Eschbach. *Visualisierungen im Schaltkreisentwurf*. PhD thesis, Institut für Informatik, Albert-Ludwigs-Universität Freiburg, June 2008.

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

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

[11] Hauke Fuhrmann and Reinhard von Hanxleden. On the pragmatics of model-based design. Technical Report 0913, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2009.

[12] Erich Gamma, Richard Helm, Ralph Johnson, and John M. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995.

[13] 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.

[14] 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.

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

[16] 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 *LNCS*, pages 167–182. Springer-Verlag, 1998.

[17] David Harel and Gregory Yashchin. An algorithm for blob hierarchy layout. *The Visual Computer*, 18:164–185, 2002.

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

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

[20] 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`.

[21] 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.

[22] 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.

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

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

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

[26] Georg Sander. Layout of compound directed graphs. Technical Report A/03/96, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, June 1996.

[27] Georg Sander. Layout of directed hypergraphs with orthogonal hyperedges. In *GD 2003: Proceedings of the 11th International Symposium on Graph Drawing*, volume 2912 of *LNCS*, pages 381–386. Springer-Verlag, 2004.

[28] Georg Sander and Adrian Vasiliu. The ILOG JViews graph layout module. In *GD 2001: Proceedings of the 9th International Symposium on Graph Drawing*, volume 2265 of *LNCS*, pages 469–475. Springer-Verlag, 2002.

[29] 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. SIAM, 1990.

[30] Miro Spönemann. On the automatic layout of data flow diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2009. `http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/msp-dt.pdf`.

[31] Miro Spönemann, Hauke Fuhrmann, Reinhard von Hanxleden, and Petra Mutzel. Port constraints in hierarchical layout of data flow diagrams. In *17th International Symposium on Graph Drawing (GD'09)*, LNCS, Chicago, September 2009. Springer.

[32] Kozo Sugiyama and Kazuo Misue. Visualization of structural information: automatic drawing of compound digraphs. *IEEE Transactions on Systems, Man and Cybernetics*, 21(4):876–892, Jul/Aug 1991.

[33] 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.

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

[35] 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.

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

[37] Roland Wiese, Markus Eiglsperger, and Michael Kaufmann. yFiles: Visualization and automatic layout of graphs. In *GD 2001: Proceedings of the 9th International Symposium on Graph Drawing*, volume 2265 of *LNCS*, pages 588–590. Springer-Verlag, 2001.