KLay Layered

Project Overview

Responsible:

- Unknown User (cds)

Key Publications:

- Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. Drawing layered graphs with port constraints. *Journal of Visual Languages and Computing, Special Issue on on Diagram Aesthetics and Layout*, 25(2):89–106, 2014. The original publication is available at www.sciencedirect.com. (pdf / bib)

Further Publications:

- Miro Spönemann, Hauke Fuhrmann, Reinhard von Hanxleden, and Petra Mutzel. Port constraints in hierarchical layout of data flow diagrams. In *Proceedings of the 17th International Symposium on Graph Drawing* (GD'09), volume 5849 of LNCS, pages 135–146, Springer, 2010. The original publication is available at link.springer.com. (pdf / bib)
- Lars Kristian Klauske, Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. Improved layout for data flow diagrams with port constraints. In *Proceedings of the 7th International Conference on Diagrammatic Representation and Inference* (DIAGRAMS'12), volume 7352 of LNAI, pages 65–79, Springer, 2012. The original publication is available at link.springer.com. (pdf / bib)
- Miro Spönemann, Christoph Daniel Schulze, Ulf Rüegg, and Reinhard von Hanxleden. Drawing layered hypergraphs. Technical Report 1404, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, April 2014. (pdf / bib)

Related Theses:

- Miro Spönemann, *On the automatic layout of data flow diagrams*, March 2009 (pdf)
- Philipp Döhring, *Algorithmen zur Layerzuweisung*, September 2010 (pdf)
- Christoph Daniel Schulze, *Optimizing automatic layout for data flow diagrams*, July 2011 (pdf)
- Insa Fuhrmann, *Layout of compound graphs*, February 2012 (pdf)
- John Julian Carstens, *Node and label placement in a layered layout algorithm*, September 2012 (pdf)
- Katja Petrat, *Erweiterung und Implementierung eines Knotenplatzierungsalgorithmus*, March 2014 (pdf)

KLay Layered is a layer-based layout algorithm mainly intended for diagrams with an inherent data flow direction. This page describes its general architecture, while child pages discuss the different phases, its intermediate processors, and other specialized topics.

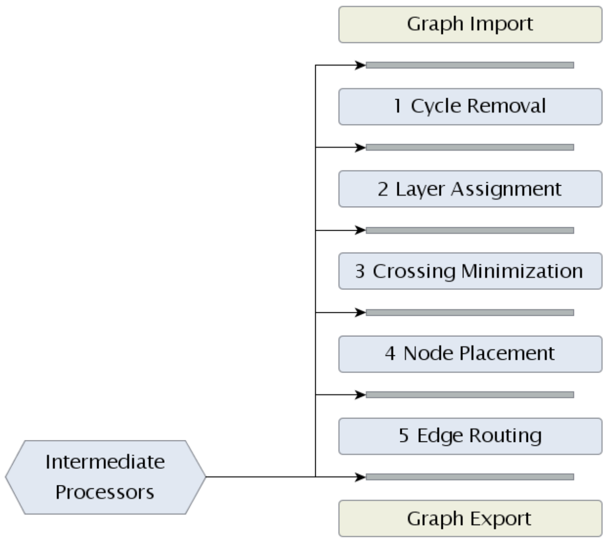
Contents

Error rendering macro 'excerpt-include'

No link could be created for 'Downloads - KIELER Layout Algorithms'.

Architecture

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



The backbone of KLay Layered are its five layout phases, of which each is performing a specific part of the work necessary to layout a graph. Three of the five phases (layer assignment, crossing minimization, and node placement) go back to a paper by Sugiyama et al. They are widely used as the basis for layout algorithms, and can be found in loads of papers on the topic. A detailed description of what each layout phase does can be found on this page.

Intermediate processors are less prevalent. In fact, they are one of our contributions to the world of layout algorithms. The idea here is that we want KLay Layered to be as generic as possible, supporting different kinds of diagrams, laid out in different kinds of ways (as long as the layout is based on layers). Thus, we are well motivated to keep the layout phases as simple as possible. To adapt the algorithm to different needs, we then introduced small processors between the main layout phases (the space between two layout phases is called a *slot*). One processor can appear in different slots, and one slot can be occupied by more than one processor. Processors usually modify the graph to be laid out in ways that allow the main phases to solve problems they wouldn't solve otherwise. That's an abstract enough explanation for it to mean anything and nothing at once, so let's take a look at a short example.

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

For graphs that are not connected it is possible to execute the algorithm, i.e. the five phases with intermediate processors, separately on each connected component. The connected components processor splits an unconnected graph into multiple connected graphs and rearranges them after the layout of each component has been computed. This helps to present the components more compactly and neatly.

The interface to the outside world finally allows us to plug our algorithm into the programs wanting to use it. While not strictly part of the actual algorithm, that interface allows us to lay out graphs people throw at us regardless of the data structures used to represent those graphs. Internally, KLay Layered uses a data structure called LGraph to represent graphs. (an LGraph can be thought of as a lightweight version of KGraph, with the concept of layers added) All a potential user has to do is write an import and export module to make his graph structure work with KLay Layered. Currently, there's only two module available for importing KGraphs, which make KLay Layered work with KIML: KGraphImporter, which imports a single hierarchy level, and CompoundKGraphImporter, which imports and flattens the whole hierarchy to enable compound graph layout.

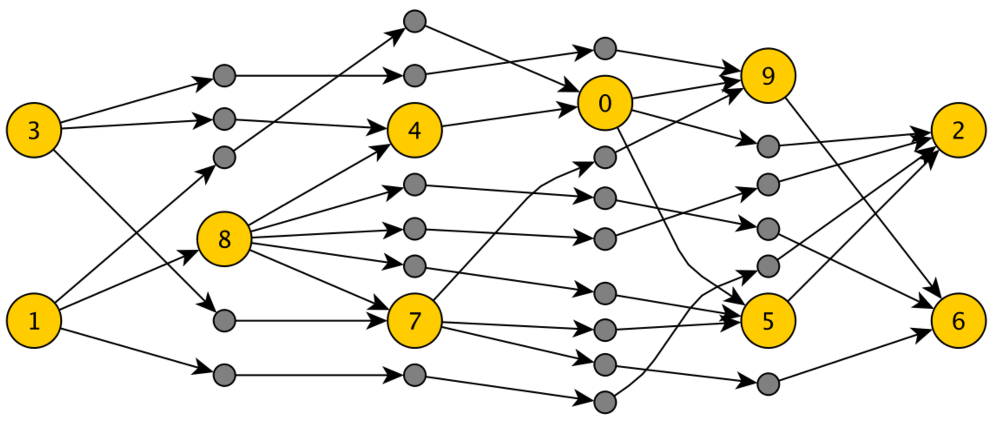
More details on the classes relevant to this architecture can be found below.

Dummy Nodes

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

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

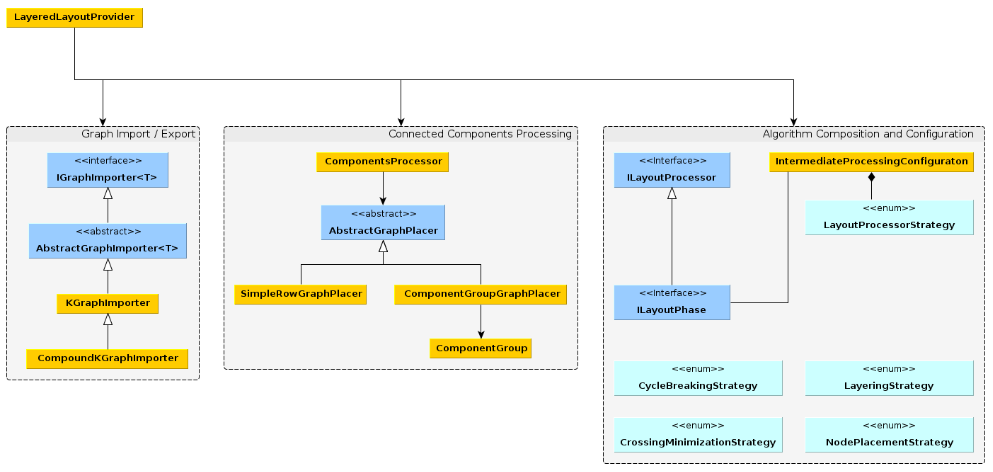
See the graph below for an example. The orange nodes were layered, but the layering was by no means a proper layering. Thus, gray dummy nodes were added.



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

Class Design

The class design of KLay Layered is probably best explained through the use of the following diagram, which you will now have to enlarge and study (don't worry, we'll explain the different parts in a minute):



Lets work our way through the diagram.

Graph Import and Export

Since KLay Layered's internal format is the LGraph, we need a way to import KGraphs (maybe also other external formats) into the LGraph format, and a way to apply the computed layout information back to the original graph. This is what IGraphImporter implementations are for. The AbstractGraphImporter predefines methods for handling common import situations, such as generating dummy nodes for hierarchical ports. The KGraphImporter knows how to import and export a single level of hierarchy, while the CompoundKGraphImporter knows how to import the whole hierarchy at once.

Connected Components Processing

Each connected component of a graph can be laid out separately and afterwards placed in a way as to minimize the area taken up by the drawing. This results in much more pleasing and space-efficient layouts than if we were to only consider a graph as a whole. Splitting and rejoining the graph is implemented in the ComponentsProcessor. Splitting is always the same, so the processor just does that itself. Joining connected components, however, can be done in different ways, implemented by different subclasses of AbstractGraphPlacer (which provides commonly used methods to move connected components around). The CompoundGroupGraphPlacer uses a sophisticated algorithm and uses ComponentGroup instances as the data structure to operate on.

Algorithm Composition and Configuration

KLay Layered can flexibly adapt itself to different layout requirements. To that end, the algorithm itself is basically just a list of modules (ILayoutProcessor) that the input graph is passed through. However, the algorithm is still based around the concept of five different layout phases, which implement the ILayoutPhase interface. The algorithm configuration happens like this:

1. The input graph is annotated with information as to which concrete implementations of the different layout phases to use. Each phase (except for the last one) has a corresponding strategy enumeration describing the different choices. (Strategy pattern, anyone?) The layout phases are instantiated and form the basic structure of the algorithm.
2. Each phase implementation may require a list of ILayoutProcessors to be executed in the different intermediate processing phases. These requirements are described using IntermediateProcessingConfigurations. The requirements of each phase are collected and instantiated.
3. From the phase and layout processor instances, the final list of modules is compiled that makes up the currently required incarnation of the algorithm.

Putting It All Together

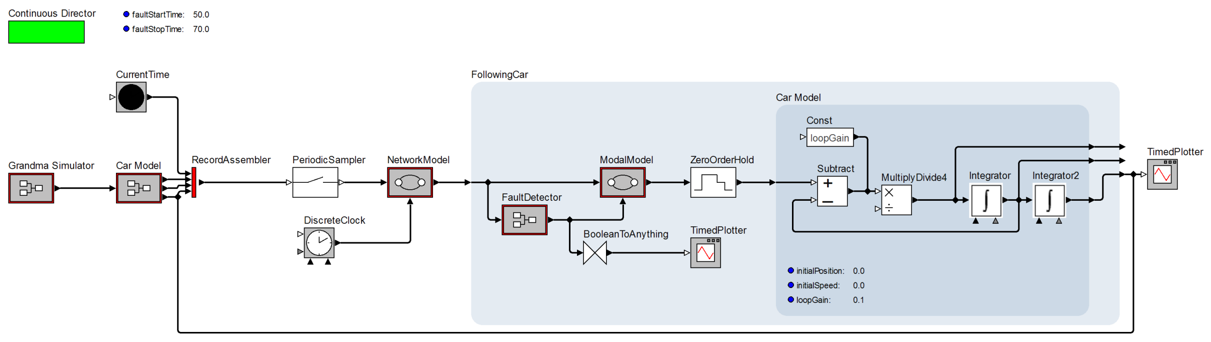
The central class is the LayeredLayoutProvider. It does the following:

1. Using one of the IGraphImporter implementations, it imports the KGraph to be laid out into KLay Layered's LGraph format. Which implementation is used depends on whether the algorithm has to compute a layout for all hierarchy levels at once (compound layout) or not.
2. Since the algorithm's configuration depends on the graph to be laid out, the LayeredLayoutProvider now has to decide which ILayoutPhase implementations to use for each of the five layout phases (this is specified with the different phase strategy enumerations). Once that is decided, each ILayoutPhase is queried for an IntermediateProcessingConfiguration, which describes – using the LayoutProcessorStrategy – the ILayoutProcessors it needs in each of the intermediate processing slots. The outcome of this step is a list of ILayoutProcessor instances that constitute the concrete layout algorithm.
3. Next, the imported graph is split into its connected components using the ComponentsProcessor. The processor does the work of splitting the graph into its connected components.
4. The main step: executing the algorithm (the list of ILayoutProcessor instances, as you will certainly remember) on each connected component.
5. With a layout computed for each connected component, the ComponentsProcessor is invoked again to join them together. The processor can use one of two implementations of the AbstractGraphPlacer class: SimpleRowGraphPlacer simply places the connected components in rows, trying to adhere to the required aspect ratio. The ComponentGroupGraphPlacer is more sophisticated and tries to work around problems with connected components processing in compound nodes.
6. Finally, it again uses one of the IGraphImporter implementations to apply the computed layout back to the KGraph.

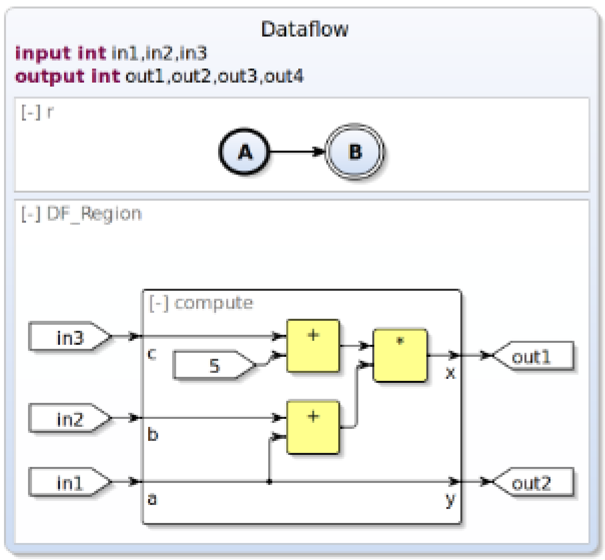
Example Layouts

Below some example diagrams that were layouted with the KLay Layered algorithm.

Ptolemy Diagram

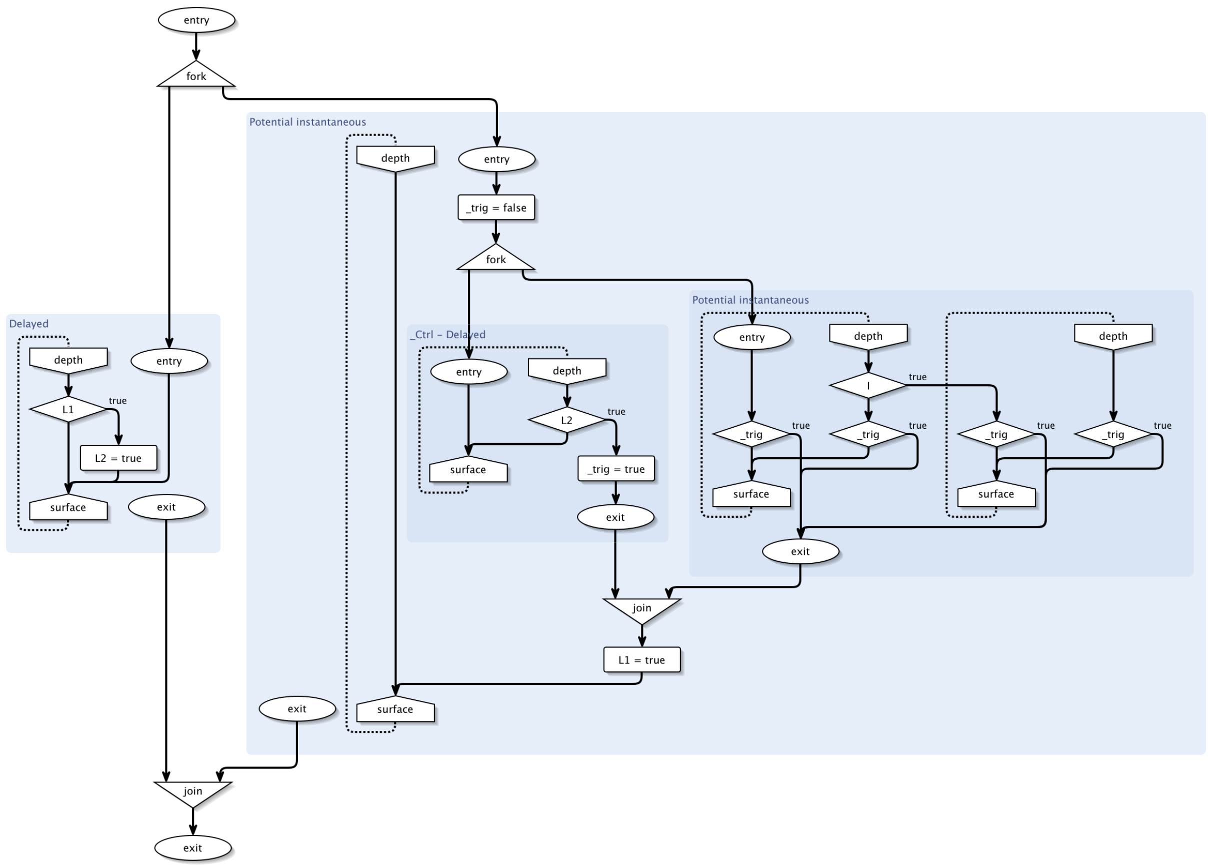


SCChart with Dataflow



```
input int in1,in2,in3
output int out1,out2,out3,out4
```

SCG


entry
fork
Potential instantaneous
depth
entry
_trig = false
fork
Delayed
depth
entry
L1
true
L2 = true
surface
exit
_Ctrl – Delayed
entry
depth
L2
true
surface
_trig = true
exit
Potential instantaneous
entry
depth
I
true
_trig
true
_trig
true
surface
depth
_trig
true
_trig
true
surface
exit
join
L1 = true
exit
surface
join
exit