Tool Support for Layout Algorithm Development with ELK

Yannic Borgfeld

Bachelor Thesis
March 2019

Real-Time and Embedded Systems
Prof. Dr. Reinhard von Hanxleden
Department of Computer Science
Kiel University

Advised by
Dipl.-Inf. Christoph Daniel Schulze

Selbstständigkeitserklärung

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

Kiel,

Abstract

ELK is a framework to connect diagram viewers and editors to layout algorithms. Besides, it provides the option to integrate new algorithms. This thesis identifies flaws and inconveniences in the process of developing those, proposes possible solutions and describes their implementation.

Such a process starts at the development setup. An essential flaw in ELK's setups for a specific target group was identified, which renders its debug tools rather useless. A proposition was made to adapt the setup to the target group without affecting others.

Setting up a layout algorithm executable with ELK is a tedious process, since it has to be done by hand and the steps include the handling of Eclipse's plug-in system. Executing these steps take some time, especially for a developer who is inexperienced with ELK. An automation was implemented in order to reduce the complexity and time spent.

When testing an algorithm, one has to start a second IDE. As a consequence the developer has to switch between windows to examine textual console output produced by the algorithm. By introducing the *layout log* view to ELK this minor inconvenience is taken care of.

ELK does not have a tool to view intermediate and auxiliary graphs. Therefore the already existing *layout graph* view was reworked to allow for individual logging of graphs. This might help developers to identify the cause of an algorithm's unexpected behavior faster and simpler. Furthermore the graph drawing has changed to also display graphs with elements outside of its boundaries including ones at negative coordinates.

Contents

List of Figures

Acronyms

API Application Programming Interface

DFS Depth First Search

ELK Eclipse Layout Kernel

ELKG ELK Graph Format

ELKT ELK Text Format

IDE Integrated Development Environment

JSON JavaScript Object Notation

KIELER Kiel Integrated Environment for Layout Eclipse RichClient

KlighD KIELER Lightweight Diagrams

PNG Portable Network Graphics

RTSYS Real-Time and Embedded Systems

SCCharts Sequentially Constructive Statecharts

UI User Interface

URL Uniform Resource Locator

XML eXtensible Markup Language

Chapter 1

Introduction

The presentation of data often requires the drawing of *graphs*. Such data can be urban rail tracks, a mechanical process, or users in a social network. In case of the urban rail tracks a map with a lucid layout of stations and their connections is crucial to make navigation as simple as possible for every passenger. Mechanical processes are often represented in flow chart diagrams. Diagrams of relationships between user of a social network might require to focus on a subset of data. This may cause certain users to show up in more detail than others. All these examples share one characteristic: their graphical representation consists of *nodes* and *edges*. These are the ingredients used to build graphs. In theory, a graph consists of a set of nodes and a set of edges. In regards to the examples the stations, states, and users are represented by nodes. The tracks, transitions between states, and relations of user are symbolized by edges.

A proper arrangement of the graph's elements is important for the diagram's comprehensibility. This is easily done for a few nodes and edges by hand, but gets more difficult with an increasing number of elements. Designing a map for a city's public transportation system by hand is a justifiable time investment considering the number of people examining it. This may also apply to flow charts, but these sometimes tend to be larger for more complex processes. The number of users in a social network nowadays is rather large. Regardless of its purpose this data set cannot be managed by a human in an justifiable time span. In this case algorithms can give assistance to the creators of graph drawings and automate the process. These are called *graph drawing algorithms* [DET+94]. They are developed in order to save the developers time, but also to apply certain aesthetic criteria. For example, the number of edge crossings is supposed to be held low and graph elements should not come too close to each other. Over time many types of these algorithms were developed for different kinds of graphs. For instance, the nature of relationships between social network users requires a different layout than flow charts. The foundation of aesthetic criteria has been researched to find some common ground on what these are [Har98; Pur02; BRS+07].

A framework to perform layouts on graphs is the Eclipse Layout Kernel (ELK). It offers interfaces to connect diagram viewers and editors through its own graph structure to its layout algorithms, but it also allows for the integration of new ones, so developers can incorporate algorithms on their own. ELK's arguably most important algorithm is based on Sugiyama's *layered approach* [STT81]. In its original form it features five phases, which already indicates that the topic around laying out graphs is in fact complex and therefore developing such algorithms is difficult. I experienced this myself when taking a course on *Automatic Graph Drawing* at Kiel University. We were faced with tasks of implementing layout algorithms with ELK.

The first task was to implement the *force-based approach* proposed by Fruchterman and Reingold [FR91]. It was originally based on a model where nodes are imagined as metal rings and edges as springs. Fruchterman and Reingold modified this approach to use an analogy for natural forces between elements attracting and repulsing each other. Our first challenge even before implementing the actual algorithm was to setup a layout algorithm project. Even though ELK offers a concise documentation about how to do so, we ran into problems because the process is rather error-prone and not necessarily trivial for someone who has just learned about the framework. After this hurdle was

overcome, we shifted our attention to the algorithm's implementation. The process of programming did not take long, but the debugging process did. Executing the algorithm only gave us feedback in the form of console output and the resulting graph. The impact of changes we were making to the algorithm's parameters were not really traceable through the development tools ELK offered. These only allowed for viewing the final graph and measuring the execution time. To view intermediate steps, in order to see when exactly our algorithm behaved unexpected, we manually limited the algorithm's number of iterations. Thus, we were be able to view the state of the graph during the execution. The effort this method took was very high and therefore really time-consuming.

At a later stage of the lecture we were supposed to implement the aforementioned layered approach. It relies more on custom data structures. The algorithm places the nodes in layers and sorts them with a special criterion to minimize crossings. Based on the layers' orders the nodes are placed and the edges are routed. When we ran into unexpected behavior with this, we were again investigating. This time we printed the node names in regards to their arrangement in the layers to the console to inspect how the implemented sorting fails. We tested the algorithm on different graphs to narrow down the error's cause and because of that we came across a major inconvenience. To test an ELK algorithm one must start a *target platform* which is in this case an instance of the Eclipse Integrated Development Environment (IDE). This comes due to the fact that layout algorithms integrate with ELK via a plug-in system, hence they are executed in an Eclipse instance. Thus, a second IDE must be started in order to test a layout algorithm. This causes console output to appear in another Eclipse instance than the algorithm is executed in. In our case it was a tedious process to constantly switch between two windows in order to inspect output.

1.1 Problem Statement

Layout algorithm development is a difficult undertaking. ELK already has some debug tools, but further additions to its repertoire could aid developers in certain scenarios as described above. This thesis aims to get rid of inconveniences including the issues of manual project setup, tedious textual debugging, and viewing intermediate graphs by providing appropriate tools for layout algorithm development in ELK. These should aid developers in every stage of creating such algorithms. For that I have to identify specific issues in the whole process starting at the development setup. I have to investigate solutions to simplify the setup of layout algorithm projects and explore possible tools for supporting the actual programming and debugging process.

1.2 Related Work

Not a lot of related work exists in the field of supporting layout algorithm development. Still, in the Real-Time and Embedded Systems (RTSYS) working group at Kiel University some tools were built. One of those is a *random graph creator* [Sch15] which was implemented to create a large number of graphs for testing algorithms. It allowed to specify requirements for the generated graphs to examine specific test cases with a large sample size. Another one is GrAna [Rie10], a graph analysis tool to examine structural graph properties such as node and edge count. It also allows to analyze graph drawings and provides number of crossed edges or bendpoints.

1.3 Outline

This section describes the remainder of this thesis. Chapter 2 provides the necessary knowledge. First it will introduce layout algorithms in more detail (Section 2.1). It will then dive into the topic of the Eclipse IDE and its extensibility (Section 2.2). Afterwards the technical details of ELK are presented (Section 2.3) and a very small introduction into the programming language Xtend is given as well as into KIELER. Chapter 3 examines the current state of developing layout algorithms for ELK and identifies some problems. Chapter 4 presents possible solutions to those problems and Chapter 5 is about the implementation of those. In Chapter 6 implemented concepts are evaluated and Chapter 7 will draw a conclusion.

Chapter 2

Preliminaries

This section introduces the basic knowledge the thesis is built on. It will dive into all necessary topics around layout algorithms and technological basics surrounding the Eclipse IDE and ELK.

2.1 Layout Algorithms

As one can imagine there is an infinite number of possible drawings for a graph which also come with many possibilities of bad drawings. A poorly drawn graph can make the content, despite great quality, worthless. Many characteristics cause a graph to be hardly comprehensible. For instance, these are closely drawn vertices, crossed edges, or overlapping labels. Thus the creator of such graph has not only to care about the content but also about the layout. Often it is tedious work to design a proper graph layout by hand and it sometimes takes as much time as the content's creation. In those cases *layout algorithms* may save a great amount of work by controlling the graph's space requirements, node placement, as well as its edge routing.

This brings up the question: What qualifies a layout as good? The answer to this question may have a very subjective notion to it. Still, some objective points can be made, such as: Nodes should have a proper spacing between them and should not overlap, and having the least number of edge crossing.

A layout algorithm usually consists of several algorithms. Those may be divided in different phases as there are multiple matters to care about while laying out a graph. Node placement and edge routing are obvious ones that come to mind. A take on laying out graphs is the aforementioned *Sugiyama approach*. It features five phases that build on each other by serving a data structure required by the following phase. The first one is the cycle breaking. It reverses edges in order to eliminate any cycle in the graph. This is done to assign layers to every node in the following phase, such that all edges point to a node that is in a subsequent layer. Afterwards the nodes get reordered within their layer to minimize edge crossings. Then the nodes properly get placed with respect to their layers. In the end edges get routed and previously reversed edges are brought back in their actual direction. Finally the graph's layout is finished.

2.2 Eclipse Integrated Development Environment

Eclipse is commonly known as IDE mostly used for Java Development, but that is by far not its only use. Its extensibility makes it useful in many scenarios and in fact it supports many more programming languages. Eclipse can be extended through *plug-ins* and it provides an open-source platform for creating just these. One can also make a whole new application by creating them and using the Eclipse run-time.

The Eclipse application itself contains only a small run-time engine called Equinox[1], which does

[1] http://www.eclipse.org/equinox/

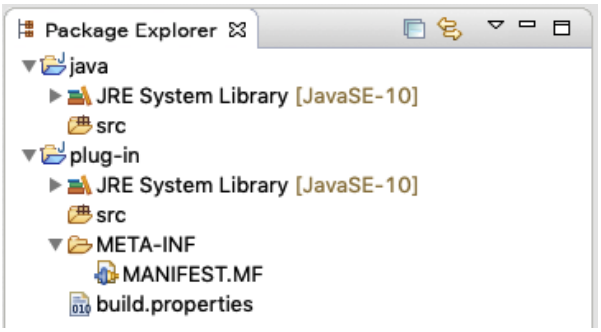


Figure 2.1. Difference between newly created java and plug-in projects.

not offer many end user functionalities. It is a reference implementation of the OSGi-Framework[2] and it allows for easy integration of so-called OSGi bundles and plug-ins for Eclipse are just that. Thus additional functionalities can easily be added to an already installed Eclipse application.

2.2.1 Plug-ins

A plug-in is a structured component, which can be run by the Eclipse run-time engine. It can provide additional functionalities as well as extend or customize other plug-ins' functionality. Such a plug-in is written in Java. When creating a new plug-in project using the designated wizard, one can notice a difference compared to a plain Java project. Besides the source package it contains two additional files as shown in Figure 2.1. One of them is located in the `META-INF` directory and is named `MANIFEST.MF`. This file contains various meta data about the plug-in such as the identifier, bundle and module name, the execution environment, and dependencies to other plug-ins or packages. The new project also has a `build.properties` file containing information about the compilation process of the plug-in.

In order to add any new functionality to the Eclipse platform one has to use an *extension point*. These can be provided by plug-ins to allow other plug-ins to register as an *extension*. For instance, if one has the desire to add a view to the Eclipse platform the newly created plug-in has to register as an extension to the view. This is done by choosing the `org.eclipse.ui.views` extension point for the plug-in to extend. If doing so, one most likely has to enter extra information depending on the extension point's requirements. Looking at the example of adding a view, this is a path to an icon, a class that implements the interface `org.eclipse.ui.part.ViewPart`, and some more meta data. If provided by the extension point, a new Application Programming Interface (API) might be available in the plug-in. All extensions and extension points are registered within the project's `plugin.xml` file. This file only has to be created if dealing with extensions in a plug-in.

The mentioned files can be modified by using a specific editor provided by Eclipse itself.

Some base plug-ins already exist and are provided by Eclipse. One of the two most used base plug-ins is the *workbench*. With the workbench Extension Point, User Interface (UI) elements of the application can be altered or other custom ones can be added. This includes menus, toolbars, dialogs, wizards, and other custom views and editors. The other one is the *workspace* (also called *resource management* in the documentation). It allows for access to the application's set of files, which also include Eclipse projects, and provides tools to manipulate those files.

There are other default plug-ins having extension points such as one for building a debugger, one for implementing a Version Control System (VCS), as well as one for providing context sensitive

[2] `https://www.osgi.org/developer/what-is-osgi/`

```
1  @Override
2  public void createPartControl(Composite parent) {
3     parent.setLayout(new GridLayout(2, false));
4     ListViewer listViewer = new ListViewer(parent);
5     listViewer.setContentProvider(
6        ArrayContentProvider.getInstance());
7     String[] elements = {"These", "are",
8        "some", "elements"};
9     listViewer.setInput(elements);
10    Text textViewer = new Text(parent, SWT.NONE);
11    textViewer.setText("This is some text.");
12 }
```

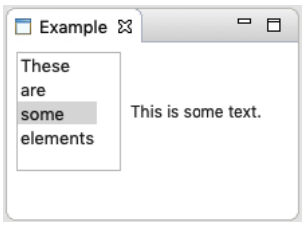


Figure 2.2. Some example code producing a simple view.

information and online documentation.

There are more plug-ins used to implement some features during this thesis' work. Those are built on top of the workbench and workspace to help with certain application parts implemented very often. For example there is the JFace UI Toolkit, which simplifies some common implementations in UI design for Eclipse such as dialogs, wizards, user preferences, actions, and widget management.

Plug-ins cannot be run within the same Eclipse instance they have been implemented in. For that reason they have to be compiled and run by a *target platform*. This is just another Eclipse instance that can be set up and started from the development platform to test plug-ins.

2.2.2 Views

Introducing a view to Eclipse requires a plug-in. Within its manifest the dependencies `org.eclipse.ui` and `org.eclipse.core.runtime` need to be added. Then a class extending `org.eclipse.ui.part.ViewPart` has to be created and registered as a new extension of `org.eclipse.ui.views`, which only requires a name, an id, and the path to the class. Two methods must be implemented: `viewPartControl(Composite parent)` and `setFocus()`. The latter is called when the view comes back into focus and is supposed to take care of notifying every view element to be switch to their focused appearance. The former is the more interesting method. It is executed on the view's start to set up all of its view elements. The parameter `parent` is a *composite* used for structuring view. These can contain other composites or actual view elements. *Layouts* can be assigned to them in order to determine how its contained elements are arranged. These elements also can be given *layout data*, which decides how they are placed within their assigned space of the containing composite. Any kind of view element taking some structured input, such as arrays, lists, and tree structures, must have a *content provider* assigned in order to adapt the given type of input. This provides the dynamic of being able to fit any custom data structure into a structured view element. These content providers are already available for common data structures, but have to be created for more complex ones. They also need a *label provider* which takes care of displaying icons and text for single elements. In case the elements are supposed to be selectable, a listener can be registered which can extract some data from the chosen element and display it in another view element for example.

A simple example of code constructing a view is demonstrated in Figure 2.2. One can see a grid layout with two columns is set on the parent composite in order to place the views beside each other. A list viewer is created, which a simple array content provider gets passed to, and a string array is

allocated as input. A second element in form of a text view is created and a string to be displayed is set as input. The result after starting the view is seen on the right.

It is also possible to add so-called *actions* to the view. These are buttons located in the grey area right to the the view's title. They can be registered for a view via a *tool bar manager*. For creating an action one needs a class extending the abstract class `action`.

2.3 Eclipse Layout Kernel

The Eclipse Layout Kernel (ELK) is an Eclipse Incubation project[3] developed by the Real-Time and Embedded Systems (RTSYS) working group at Kiel University. It offers an infrastructure for graphs and diagrams from another application to be connected to automatic layout algorithms. Some are included in the project. They are highly configurable, which allows users to adjust them to their personal preferences. An example of this is the layered layout algorithm, which is inspired by the Sugiyama approach [STT81]. It also offers developers to extend the framework by integrating custom layout algorithms. These will integrate into ELK just like any other algorithm. The reader should note that ELK does not give one the option to render graphs directly but only compute automatic layouts of these.

2.3.1 ELK Architecture

ELK is composed of multiple Eclipse plug-ins. These plug-ins can be separated in two categories: *algorithms* and *core*. While the algorithm category contains all layout algorithms provided by ELK, the core consists of all necessary components to model graphs and execute layout algorithms, as well as connecting those to an editor or external viewer. It also holds implementations for tools built for debugging and monitoring executions, which this thesis will heavily touch upon.

If one wants to connect their own applications diagrams or graphs to ELK's algorithms, they have to provide a diagram layout connector. Its task is to transform some data structure into an ELK graph, which a layout can be performed on. When the algorithm was run, the diagram layout connector must apply the layout to the original data structure.

2.3.2 ELK Graph

In general a *graph* consists of sets containing nodes and edges. Other elements such as ports and labels can be identified as well. A summary of all graph elements and their terminology can be found in Figure 2.3. Nodes can be either simple or hierarchical. A *simple node* does not contain children while a *hierarchical node* does. The same categorization can be used for edges. A *simple edge* stays within its parent node and a *hierarchical edge* connects nodes contained by different parents. Nodes can have *ports* which serve as docking points for edges.

These elements can be found in the ELK graph meta model, which is shown in form of a class diagram in Figure 2.4. These also incorporate abstract graph elements offering further extensibility. The most commonly used ELK components to assemble a graph are `ElkNode` and `ElkEdge`. The top level element of an ELK graph is an `ElkNode` containing other nodes and edges in lists.

A graph can be instantiated by either code or parsing from a specific format. One of these is the ELK Text Format (ELKT) which uses the `.elkt` extension. It provides a simple method to define parse-able

[3] `https://www.eclipse.org/elk/`

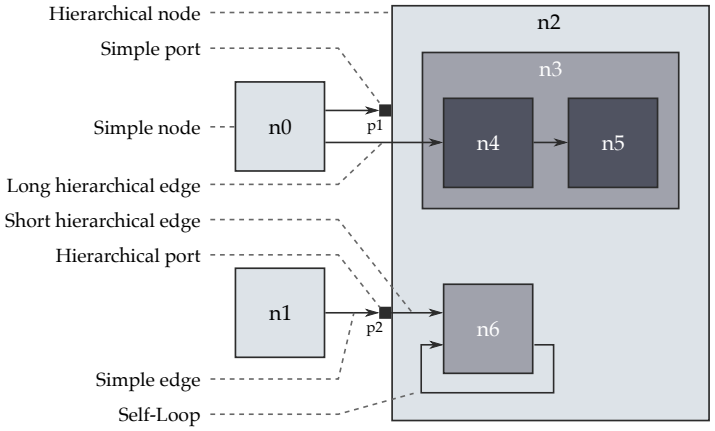


Figure 2.3. Graph Terminology. Source: [Sch]

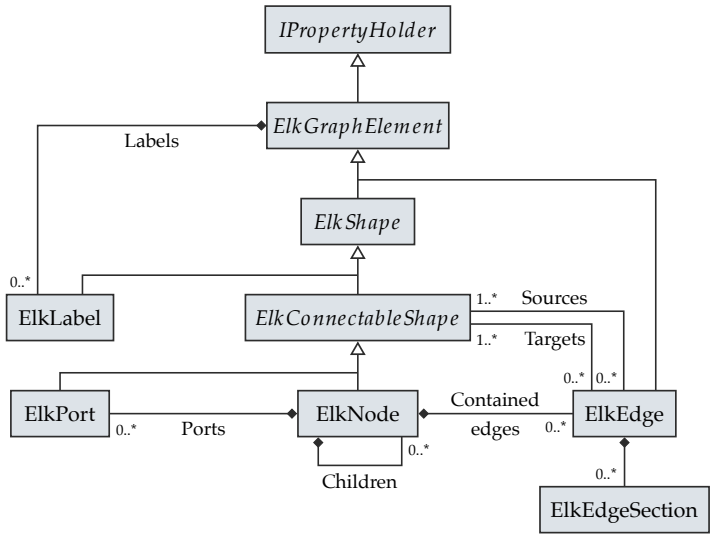


Figure 2.4. ELK meta model. Source: [Sch]

graphs. The following code snipped shows how a graph with two nodes and a connecting edge is modeled:

```
1 node n1
2 node n2
3 edge n1->2
```

Properties can be attached to the elements like so:

```
1 node n1 {
2    layout [ size: 50, 50 ]
3    label "n1"
4    port p1
5 }
```

Note that the port inside also could have properties attached within curly brackets. The same works with the JavaScript Object Notation (JSON) format. The graph from the former snippet can modeled just like so:

```
{
   id: "root",
   children: [
      { id: "n1"},
      { id: "n2"}
   ],
   edges: [{
      id: "e1", sources: [ "n1" ], targets: [ "n2" ]
   }]
}
```

Another format is the ELK Graph Format (ELKG) which is an eXtensible Markup Language (XML) based language that is not as simple as the ELKT format and rather generated by the framework than written by humans.

2.3.3 IElkProgressMonitor Interface

The *IElkProgressMonitor* interface is an important component for this thesis. Classes implementing this interface are used to track progress of a layout algorithm as well as measuring its timely performance. The progress information can be visualized in a progress bar and the measured times can be observed in the *layout time view*, which will be introduced in Chapter 3.

When an instance of a monitor is created, it is not started yet. It explicitly has to be started and stopped. The time between those two actions is measured. When instantiating a monitor it can be named and given a workload as integer. It often makes sense to hand the monitor a workload greater than 1, so progress can be displayed in a more meaningful way in a progress bar. When a chunk of work is done, one can simply make a call on the monitor passing it a number of how much work has been completed. A layout algorithm having three phases serves as an example in the following. Initially the progress monitor is created with a workload of 3. Every time a phase comes to an end the progress monitor receives a notification of one work unit being done.

One problem arises if one wants measuring performance of each phase individually. As there is only one monitor measuring time from start to end it only shows the elapsed time of the whole execution. To get rid of this problem one is able to create *sub monitors*. These can be created from the parent monitor and work exactly like its ancestors. Hence a sub monitor also can instantiate its own sub monitor. This allows for creation of a tree structure and therefore it can have infinite depth. This way developers can evaluate the performance of each phase individually. A visualization with an example can be seen in Figure 2.5. The time progresses from left to right. The top monitor is the root of the monitor structure. It has 3 child monitors named after its three phases. *Phase 2* is divided into two other sub phases. The top monitor *Layout Algorithm* measures time for the whole execution while each sub monitor only measures time for its individual time slot. The phases are drawn with some distance between them on purpose as layout algorithm tend to have some intermediate processing phases. This is often done to adapt data structure from one phase to another.

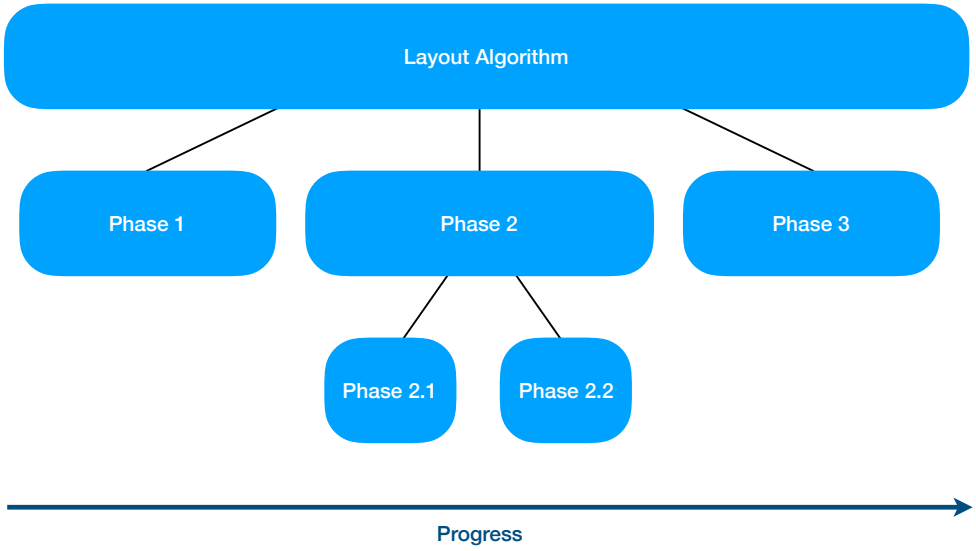


Figure 2.5. Visualization of the tree structure of progress monitors in ELK.

2.3.4 Creating a new Layout Algorithm in ELK

Some people may want to create new layout algorithms for ELK. For this matter ELK offers the `org.eclipse.elk.core.layoutProviders` extension point. This indicates that a layout algorithm is implemented in an Eclipse plug-in project. The following explanation assumes that a general plug-in project is already created.

In the beginning some dependencies need to be added: `org.eclipse.elk.core` and `org.eclipse.elk.graph`. Afterwards one creates a Java class extending `AbstractLayoutProvider` and implements the `layout` method. This way the class can be referenced later. In addition, a file with the `.melk` extension has to be placed in the base package of the project. *Melk* stands for ELK Metadata. The project must have the *Xtext*[4] nature, so some source code can be generated from the file. In order to register the algorithm there are some entries needed, just like so:

```
1  package yab.layout.thesis
2  import yab.layout.thesis.ThesisLayoutProvider
3
4  bundle {
5     metadataClass options.ThesisMetaDataProvider
6     idPrefix yab.layout.thesis
7  }
8
9  algorithm Thesis(ThesisLayoutProvider) {
10    label "Thesis"
11    metadataClass options.ThesisOptions
12 }
```

When saving, two files are generated. One is a general meta data file `ThesisMetaData Provider` and the other is an algorithm-specific meta data file `ThesisOptions`. Those files provide information needed

[4] `www.eclipse.org/Xtext/index.html`

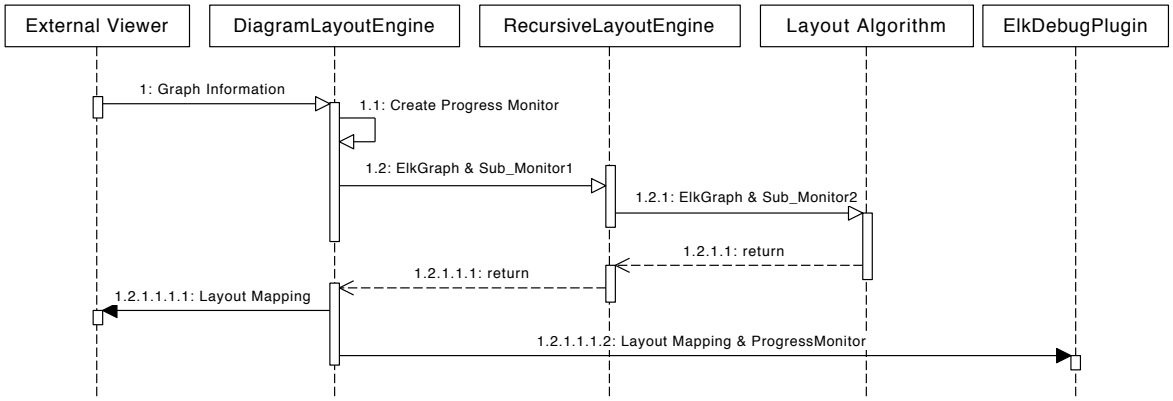


Figure 2.6. Visualization of the data flow while layouting a graph.

by the ELK framework in order to integrate the algorithm. Now the general meta data file must be registered at the previously mentioned `org.eclipse.elk.core.layoutProviders` extension point. This can be done via the plug-in manifest editor in the *extension* tab. Here one only has to enter the path to the general meta data provider, in the example above this is the `ThesisMetaDataProvider`. After doing so the algorithm is finally ready to be executed.

2.3.5 Data Flow of Debug Information

This subsection introduces the flow of data starting at an external viewer and ending at ELK's debug plug-in, which is showcased in Figure 2.6. The explanation is heavily simplified, since a lot of details are not relevant in the context of this thesis. In the following the actions around the layout engines are described very roughly. When the external viewer has a diagram supposed to be laid out, it sends the graph information to the diagram layout engine. There an ELK graph is built and a progress monitor created and started. It then creates a sub monitor and passes this on to the *recursive layout engine* together with the ELK graph. The recursive layout engine then uses an algorithm to perform a new layout by passing it the ELK graph and another sub monitor created from its own one. While the algorithm does its work on the graph, it can log information with the monitor given to it. After the algorithm finished as well as the recursive layout engine, the diagram layout engine returns the new layout mapping to the external viewer, which displays it. While doing so the ELK debug plug-in is notified. Upon start, the plug-in registers a layout listener, which is triggered by the diagram layout engine, when a layout is done. Thus it receives the final layout mapping and the progress monitor, both of which it can delegate to its views in order to display any information contained.

2.4 Xtend

Xtend is a high-level programming language that is designed for Java Virtual Machine (JVM)-based systems. It does not run on the JVM, instead java source code gets compiled from it which makes it interchangeable with Java. The syntax is similar but less verbose. It has some useful features such as template expressions which are used in this thesis. The following code showcases this:

```
1 package yab.some.interesting.package
2
```

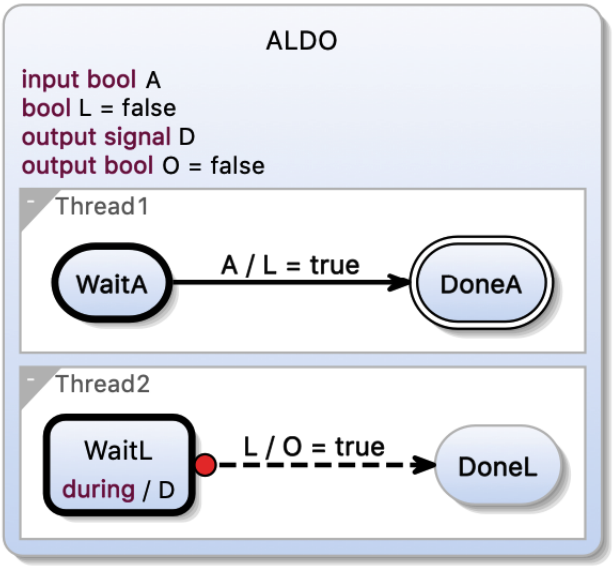


Figure 2.7. A visualization of an SCChart.

```
3 class SimpleTemplate {
4    static def String fillGap(String gap) '''
5       The following «gap» will be filled.
6    '''
7 }
```

The generated Java class then can be called in the code like this:

```
1 ...
2 import yab.some.interesting.package.SimpleTemplate;
3 ...
4    void foo() {
5       System.Out.println(SimpleTemplate.fillGap("something"));
6    }
```

Calling `foo()` will print `The following something will be filled.` to the console. This can be used to create templates for whole files.

2.5 KIELER

Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) [HFS11] is a research project about enhancing the graphical model-based design of complex systems. It is developed by the RTSYS working group at Kiel University. It is a modeling software used, for example, for Sequentially Constructive Statecharts (SCCharts) [HDM+13] which is a state-chart modeling language. It is design for specifying safety-critical reactive systems. A visualized SCChart can be seen in Figure 2.7. The visualization is laid out with ELK's layered algorithm. The SCChart's code, which the visualization was generated from, can be compiled to either C or Java code, which has intermediate steps which can be viewed with a specific compiler view.

Chapter 3

Current State of Development

This chapter describes the current state of how developers build layout algorithms in ELK. This is divided into three steps: First comes the development setup which shows multiple ways of setting up an Eclipse installation for different target groups. Then the project setup of an layout algorithm is presented, and finally everything about executing and debugging an algorithm is introduced.

3.1 Development Setup

Currently there are two different setups for developers to work on algorithms mentioned in the ELK documentation. One is for developers who only want to implement layout algorithms for ELK and the other one is for ELK contributors. Two other non-public ways are also presented and in the end a small comparison between those with regards to their target groups is made.

3.1.1 Algorithm Developer Setup

For those wanting to develop layout algorithms, the ELK documentation provides a guide[1] for doing so. It assumes that an Eclipse for Java development is already installed. To install ELK one has to find the menu item *Install new Software...* below the menu bar item *Help*. The dialog box illustrated in Figure 3.1 shows up. Then the Uniform Resource Locator (URL) of the nightly[2] or release[3] update site of ELK has to be copied and pasted into the *Work with:* field causing the list view below to update. In this updated view one has to expand the *Eclipse Layout Kernel* item and tick the check boxes next to the *SDK* and the *SDK (Sources)* features. The rest of the installation wizard can be completed without further user input, except accepting the license agreements. After a restart Eclipse is now ready for algorithm development. It now has access to some new views: the *layout time* view and the *layout graph* view, which are part of the framework's debug plug-in.

3.1.2 ELK Contributor Setup

For people wanting to contribute to ELK there is another setup which clones the whole content of ELK's repository into the workspace. This is done by following the ELK documentation's guide[4]. For that one has to download the Eclipse installer *Oomph*. In the first dialog box of the installer one has to enter the *advanced mode*, which can be found in the menu opening by clicking on the top right hamburger button. Multiple Eclipse setups now could be chosen in the following window, but one should select the version for Java development and hit *Next*. Afterwards *Eclipse Layout Kernel* must be selected in the project list. Hitting *Next* brings up the installation variables. These should be modified

[1] https://www.eclipse.org/elk/documentation/algorithmdevelopers/gettingeclipseready.html

[2] https://build.eclipse.org/modeling/elk/updates/nightly/

[3] https://download.eclipse.org/elk/updates/releases/0.4.1/

[4] https://www.eclipse.org/elk/documentation/contributors/developmentworkflow/installingwithoomph.html

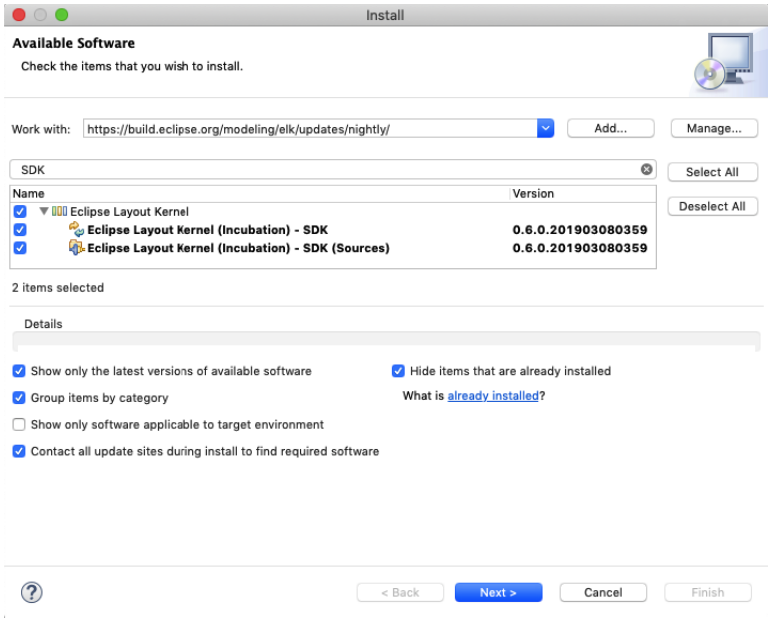


Figure 3.1. Eclipse *Install new software* dialog box.

to one's individual need. The next installation pages can be stepped through by accepting license agreements and hitting *Next*. In the end the wizard downloads the required sources and sets up the IDE for development. Now all features of ELK are available, along with their source code.

3.1.3 Other Setups

Besides the officially documented ones, there are two more non-public ways of setting up Eclipse having some additional content to offer, which benefits algorithm developers.

The first one is the *Graph Drawing* setup from the elective subject *Automatic Graph Drawing* at Kiel University. It is made for students to have an easy setup of an Eclipse platform to implement layout algorithms. The setup is similar to the one of ELK contributors, but when selecting a project one has to add a new resource on the top right with the green plus icon and enter an URL adding a new item for selection. The URL to this resource is only given out in the lecture. All ELK components will be installed without providing a clone of the source code, but in addition to that the KIELER Lightweight Diagrams (KlighD) view is installed, which is very useful for pure algorithm development, because it allows to utilize ELK's debug tools. It can be used to view ELK graphs. A deeper introduction to the view in given in Section 3.3.1.

The second one is the contributor setup for KIELER. This setup follows the guide[5] from the working group's website, which is basically the same as the graph drawing setup, only with another resource. Then the new KIELER item in the list of sources can be expanded to reveal the choice of *semantics* and *pragmatics*. The pragmatics package should be chosen and the setup proceeded as described above. These steps set up the pragmatics project and install ELK's features in Eclipse, since the project has dependencies to them. That makes the features available for developing layout algorithms. This includes being able to create new algorithms and using the layout graph view, layout time view and

[5] `https://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/Getting+Eclipse`

the KlighD view, which is part of the pragmatics project.

3.1.4 Evaluating these Setups

The officially documented way of setting up Eclipse for algorithm development does not include the KlighD view, which can be quite helpful in pure algorithm development. Without it one is unable to utilize the layout time view as it retrieves its information from the progress monitors, which it only has access to when the diagram layout engine is run. However, if developers have an own application using the Eclipse platform in combination with ELK, the diagram layout engine is run to perform layouts on their own diagrams by transforming them into an ELK graph and calling an algorithm. Thus the official setup only make sense when using an own application requiring automatic graph layouts. The ELK contributor setup might help if one wants to examine the code of other algorithms included in ELK, otherwise one should go with the former setup. For pure algorithm developers the graph drawing setup for students is the desired setup as it makes it possible to utilize the layout time view through the KlighD view. The KIELER setup does not make sense for pure layout algorithm developers as it installs a lot of features never used for the development. So what matters for algorithm development is that one has to somehow execute the diagram layout engine in order to be able to use ELK's debugging tools. Unfortunately the setup guide in the ELK documentation do not touch upon this issue and new algorithm developers may not be able to utilize the frameworks full debug possibilities.

3.2 Project Setup

For creating new layout algorithms a new plug-in project and some additional files need to be created and some modifications to the plug-in's files must be made. Currently this has to be done by hand and there is a guide[6] in ELK's documentation about how to do so. The steps were already mentioned in Section 2.3.4. The time spent executing these steps may vary heavily depending on the developer's experience and would be better spent concentrating on layout algorithm development. They also should not be bothered with Eclipse's plug-in system just for developing layout algorithms.

3.3 Developing an Algorithm

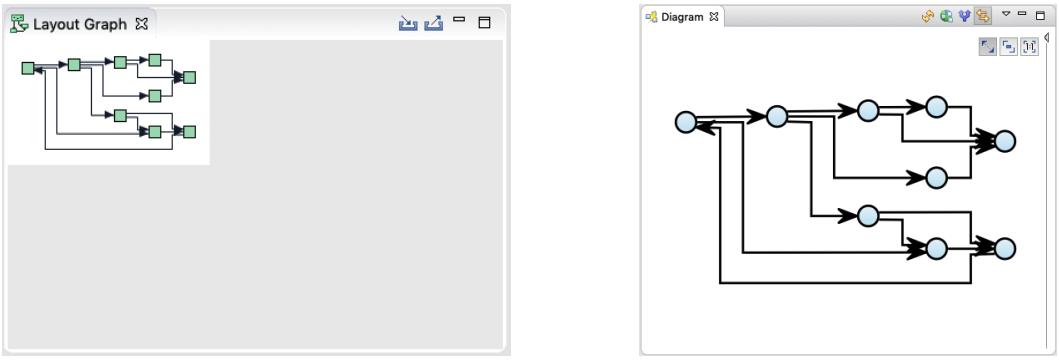
This section dives deeper into the topic of algorithm development with regards to testing and debugging. Plug-ins, and thus ELK algorithms, cannot be executed in the Eclipse platform they are developed in. A target platform has to be started in order to execute the algorithm. In the target platform a graph has to be created and the easiest way is to use the ELKT format. Once created the algorithm supposed to be executed must be annotated via the `algorithm` tag. Now some views can be opened.

3.3.1 Viewing a Graph

To view the graph there are two options. One either opens the layout graph view or the KlighD view, both shown in Figure 3.2.

The layout graph view consists of a graph canvas and two action buttons in the top right as shown in Figure 3.2a. The right button exports an image of the drawn graph as Portable Network Graphics (PNG) file, whereas the left button allows the user to load a graph into the canvas from a file.

[6] `https://www.eclipse.org/elk/documentation/algorithmdevelopers/creatinganewproject.html`



(a) Layout Graph view **(b)** KlighD view

Figure 3.2. Both graph views with the same graph layouted by the layered algorithm.

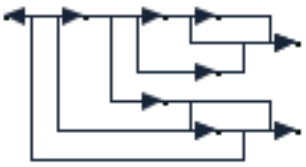


Figure 3.3. The graph's nodes do not have a size assigned.

For instance, this file could be an ELKT file. One characteristic of the layout graph view is the accuracy with which it draws the graphs. For example, if the graph only denotes simple nodes without any property such as size, they will show up as dots in the view, which can be seen in Figure 3.3. Note that this is the same graph as seen in Figure 3.2. The view also does not show anything of the graph if it has no size assigned, which a proper layout algorithm usually takes care of. If the algorithm does not assign a size to the graph, meaning its size is zero, the view shows that by staying empty.

On the other hand the KlighD view handles the subject of loading graphs very differently. It always scans any editor which the Eclipse platform has in focus. In case it holds a file containing data which can be transformed to a graph, it will display that. This includes ELKT files which can be opened in a text editor in Eclipse. Every time a change is registered, it will immediately parse the file and display the graph. Between parsing and displaying, the graph is checked for missing properties such as the node size, and fills them with default values before performing the layout. If an edge is not routed properly, missing edge sections will be inserted leading directly to their target. The graph's size is not taken into account when drawing. Nodes outside the graph's boundaries are still drawn.

One important thing to note is that if both views are opened and the focus of the Eclipse instance is on an editor containing a parseable graph, the KlighD view will be triggered to display the graph. By doing so it will run the diagram layout engine that notifies a listener, which also updates the layout graph view as shown in Section 2.3.5. That makes viewing a resulting graph in the layout graph view simpler and faster, because otherwise one would have to load the graph's file via the aforementioned load action, which might include tedious searching within the file explorer. However, it might not be an optimal representation, because the node sizes may be influenced by the assigned default values. In order to debug algorithms these may come in handy, but it is a characteristic developers should keep in mind.

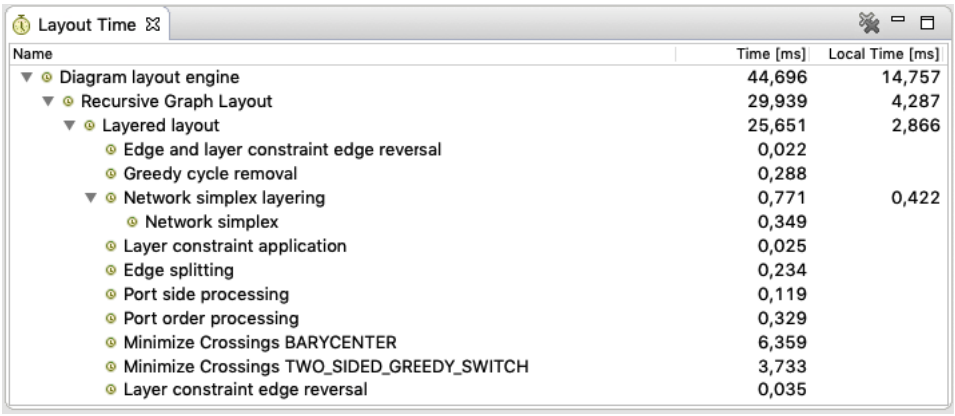


Figure 3.4. Layout Time View showing results of one execution of the layered algorithm.

3.3.2 Measuring Performance

The *layout time* view allows the developer to inspect an algorithm's execution times. An example of a measurement can be seen in Figure 3.4. One can see the results of the execution from the layered algorithm. The view contains a table with three columns. First there is the *Name* column containing expandable elements which represent progress monitors. An indentation represents a layer of sub monitors. The *Time* column displays the total time elapsed in milliseconds for the monitor's starting and ending time while the *Local Time* column takes sub monitors into consideration. The time it took a sub monitor's task to finish is subtracted from the time of the parent monitor.

This view allows the developer to identify inefficient parts of an algorithm. Measuring the execution time of either the whole algorithm or a specific phase is held simple. First, time measurement must be enabled in the ELK options located in the preferences of the target platform. There the check box *Execution time measurement* must be ticked. The following code snippet shows how to start and end the progress monitor, as well as how to create a sub monitor for it. The snippet displays the relevant code from the `layout` method of the layout provider class.

```
1 progressMonitor.begin("Thesis Algorithm", 2);
2 // do something...
3 IElkProgressMonitor subMonitor = progressMonitor.subTask(1);
4 subMonitor.begin("Some sub task", 1);
5 // do sub task...
6 subMonitor.done();
7 // do something else...
8 progressMonitor.done();
```

In the first line the `progressMonitor` will be started with the name *Thesis Algorithm*. After some work a `subMonitor` is created from the `progressMonitor` in line 3 and started in line 4. When the work is done it is stopped in line 6 and later on the same is done to its parent in line 8. This way the time of the execution of the whole algorithm was measured as well as the time of the sub task, both of which show up in the layout time view accordingly. Thus, by properly using this view together with progress monitors, the developer is able to identify inefficient parts of the algorithm easily.

3.3.3 Other Forms of Debugging

ELK offers barely any support for textual debugging. When developers want to print some output they have to use Java's console printing methods besides the possibility of creating output files. As a consequence, executing the algorithm in the target platform will lead to the output being printed into the console of the development platform. Every other debug information is shown in the target platform, though. Thus one has to switch the focus from one application to the other one frequently. The output might also be lost in between any other executed code's outputs.

Further tools for debugging are implemented in ELK and its algorithms. The check box *debug graph output*, which can be enabled in the framework's preferences, controls output generated by the diagram layout engine. Every time it lays out a graph and the check box is ticked, an ELKG file is created. The output then can be found in the user's home directory under `elk/diagram_layout_engine`. The information contained by the file are about the graph's final layout.

Other ways of debugging are utilized in ELK. To be precise these are not implemented by its debug plug-in, but rather in its algorithms. For example the layered algorithm has its own debug utility class allowing developers to output specific information about the execution in form of *JSON* and *dot* files. The JSON files describe the graph in a similar fashion such as the general XML files that are generated by the framework, except that one execution causes more than one file to appear, since intermediate steps are recorded. They cannot be parsed in order to create an ELK graph from them, because they contain properties unknown to the parser. The *dot* format is commonly used by *GraphViz*[7] and is a less verbose way of defining graphs. Every intermediate step is recorded in both formats. Those files are located in the `elk` directory as well, but this time they can be found under `layered`. They are properly named according to the algorithm's phase they were recorded in. Fortunately ELK has a boolean layout option *debug mode* implemented, which algorithms can adopt. The option can be used in a conditional branch to control whether debug output is generated. Having the option enabled makes sense when debugging an execution's results, but it does not when measuring performance with the layout time view. That would falsify the result of the measurement, since debug operation might be expensive. Thus having the option to turn a debug mode on and off is useful in some cases. In the ELKT format a simple `debugMode: true` enables it for the executing algorithm. If not specified the option is turned off.

[7] `http://www.graphviz.org`

Chapter 4

Conceptional Ideas

The previous chapter gave an overview about the experience of developing layout algorithms for ELK. Some major inconveniences could be spotted:

- Setups do not always provide the best tools for their purpose.
- There is no automatism for layout algorithm project creation.
- Debug output requires to change focus to another Eclipse instance and cannot be structured.
- The layout graph view can only show the result of an algorithm, but no intermediate steps.
- Drawn graphs are only shown when they have a width and height assigned and elements outside its boundaries are not drawn at all in the layout graph view.

This chapter will dive into the possible improvements to these issues and some approaches to those. Initially optimizations to the development setup will be discussed. Then the problem of the project setup is taken care of and afterwards a solution for the debug output is figured out. Finally some improvements for the viewing of graphs will be considered.

4.1 Development Setup

There are currently four different ways of setting up Eclipse for algorithm development as mentioned in Section 3.1. The setup for algorithm developers from the documentation installs ELK and its debug tools. For new developers wanting to learn about layout algorithms and come across ELK, this setup seems to be the most fitting one. This setup, however, does not provide every possibility to utilize ELK's debug tools, because they most likely do not have an external viewer at hand. The graph drawing setup would be the more desired one, because it actually comes with an external viewer, the KlighD view. There might be some people wanting to use ELK to develop algorithms for their own diagram viewers, while benefiting from the useful features the ELK graph model does provide. They have no need for the KlighD view, hence the original algorithm development setup is the right choice for them. I propose to combine those two setups, so that the user has the option to install the KlighD view. Some kind of explanation for why he may wish to do so should be included next to a check box determining the view's inclusion in the installation. The same might apply for the ELK contributor setup as well, because some features might only be tested in combination with an external viewer. For example, this includes the diagram layout engine. However, one has to note that most—if not all—contributors so far encounter the project through the RTSYS working group. That means they most likely have KIELER installed and with that the KlighD view.

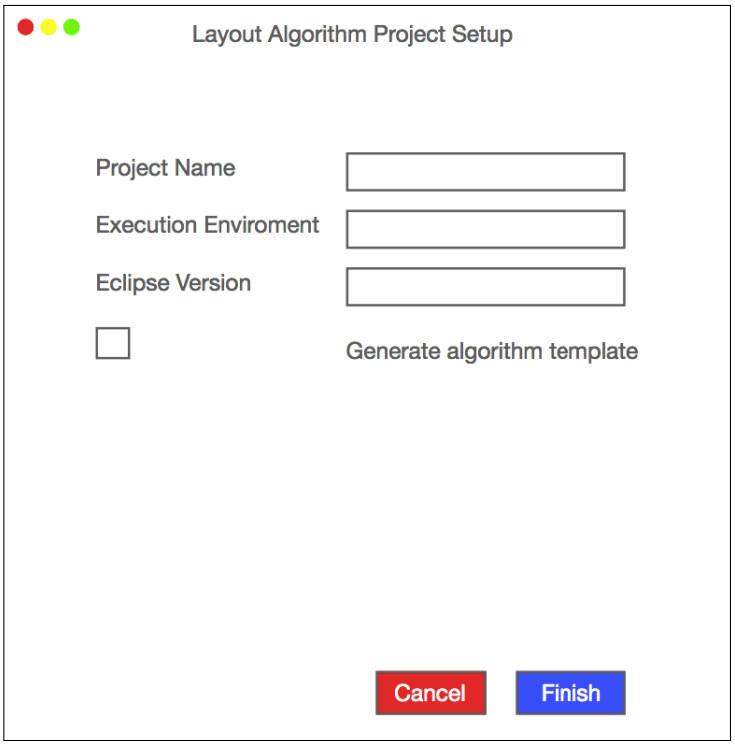


Figure 4.1. First wizard page design.

4.2 Project Setup

The project setup currently requires the user to execute all steps to create a layout algorithm project by hand. Inexperienced users might need some time to do so and they also might do some mistakes along the way, because the process is error-prone. An automatism in form of a wizard, which creates a whole layout algorithm project ready to run, would be a huge improvement over the current state. It should be simple to use and the generated project should already provide some implemented features as guideline for new developers. The first design of such a wizard is shown in Figure 4.1. It does not have a lot of user inputs. It only takes the project name, the execution environment (basically the Java version), the Eclipse version the plug-in is run on, and an option for generating an example algorithm. The wizard's dialog box can be canceled or finished. The second and third input are supposed to make it compatible for future iterations of Java and Eclipse. The generated algorithm template could, for instance, showcase some simple mechanisms such as node placement, edge routing, and layout options to provide simple examples for new developers.

Users would not have to care about the Eclipse-specific topic around extensions anymore and could not make any mistakes along the way, while setting every single file up by hand. The time spent on setting up a project would be heavily reduced and can be used for actually creating the algorithm. Besides, the wizard would make layout algorithm development in ELK slightly more accessible. It could prevent users from giving up the creation of new layout algorithms, because of a tedious setup.

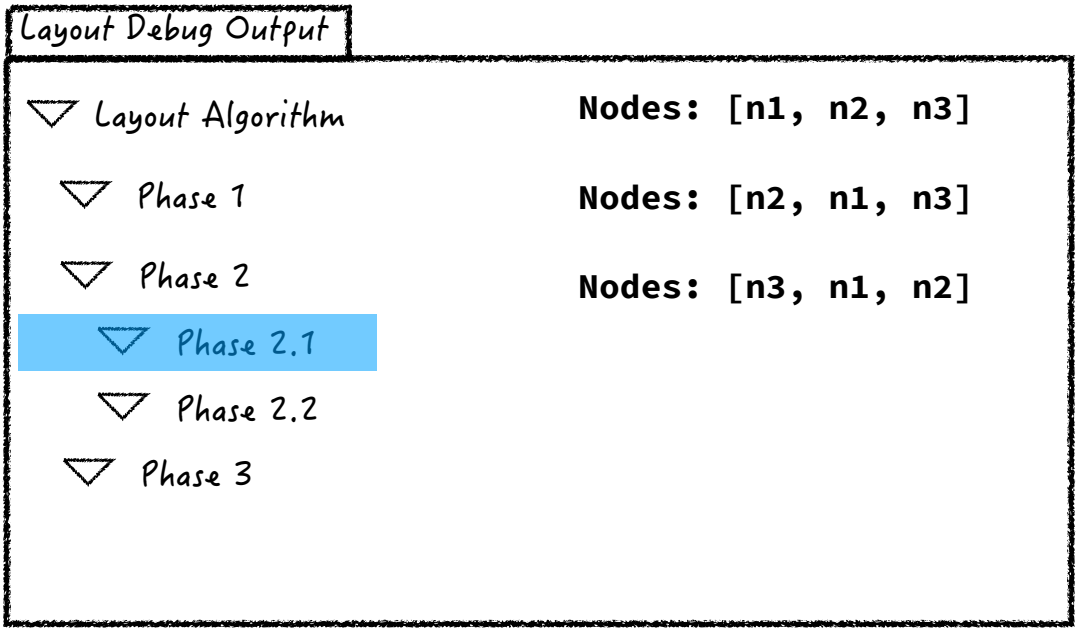


Figure 4.2. Design of a view for debug output.

4.3 Textual Debugging

Currently developers have to switch between the development Eclipse platform and the target platform in order to observe any debug output, which might appear between other output in the console. A view displaying it within the target platform could get rid of this inconvenience. It might also provide some mechanism to structure the output. This could be achieved by somehow using the progress monitors. The first rough design of such a view is displayed in Figure 4.2. To the left are the progress monitors with regard to their tree structure whose elements are supposed to be expandable and selectable. When selecting, the corresponding output is shown on the right. Some kind of indication, whether an element contains input, could be useful for providing greater clarity. Furthermore an option to filter out empty monitors could improve clarity even more.

The framework should provide a method which takes any object as output and display it as string within the view. An idea was to make additions to the progress monitor and let the view retrieve the output. It already provides debug information by measuring performance in time thus only more content would be added to it. The monitor could have a `log(Object object)` method converting any object to a string. The view would receive the information in a similar fashion to the layout time view.

Such a view would reduce the annoyance of switching focus between two windows over and over again. It also would bring in greater clarity to the output by having the possibility to utilize the monitor structure. In fact, it would be totally up to the user how he organizes the progress monitor for greater clarity of their algorithm's output.

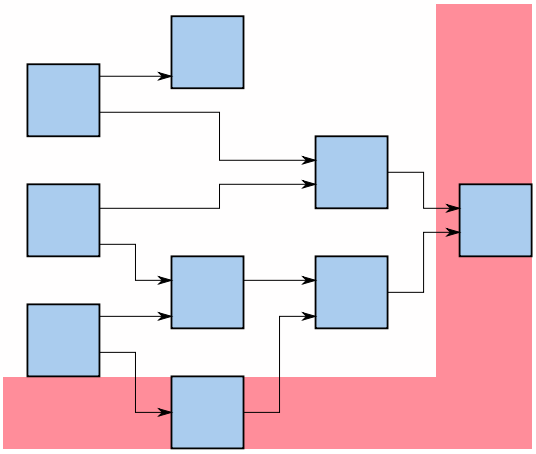


Figure 4.3. Concept of a graph with nodes out of its boundaries underlaid with red.

4.4 Graph Viewing

The layout graph view has the characteristic of showing graphs with high accuracy, but there is still room for further improvements. For instance, graph elements could be drawn even though they do not lay within the dimensions assigned to the graph. The area outside of the graph then could be underlaid with red to highlight those elements being misplaced. Figure 4.3 displays a simple example of this. Graph elements, which lay on an area having a white background, are within the boundaries of the graph. In case the graph is properly sized no red area is supposed to be displayed. So the red area marks the minimum size required for the graph to display all of its elements. Another useful feature would be the drawing of graph elements placed at negative coordinates. For that the original graph would need to be offset in order to include negative coordinates on the canvas. This additional space should also be colored in red to indicate a misplacement of contained graph elements.

Another helpful feature would be the possibility to view intermediate states of a graph. The layout of the view would be similar to the previously proposed one as seen in Figure 4.4. On the left side is a representation of the progress monitor structure. Graphs are assigned to monitors allowing developers to organize their graphs freely, similarly to the output in the previous proposition. Because a single progress monitor could hold multiple graphs, a list viewer could be put between the two view elements to give the developer the option to choose a specific graph. On selecting a monitor, its contained graphs are listed in the list viewer with a tag to identify them. When a graph is selected, it should show up on the canvas. Another approach was to provide two buttons to cycle through all graphs of a progress monitor. It also came to my mind to show the graphs for selection below their progress monitors within the tree viewer, which then might be filled up quite a lot. These approaches were abandoned, because the list view provides more clarity about which graph is shown. While the view has a tree viewer similar to the aforementioned one, it could use the same features such having an indicator for monitors containing graphs and excluding those not holding graphs through filtering.

Graphs are supposed to be logged with progress monitors. The first idea was to utilize the same method `log()`, but instead a new method `logGraph()` is supposed to be introduced. Thus a proper distinction between plain text strings and those encoding graphs can be made by using either `log()` or `logGraph()`. A parameter should be passed stating the type of the graph, so the string can be parsed correctly. Furthermore a parameter `tag` should be given to the method in order to identify the logged

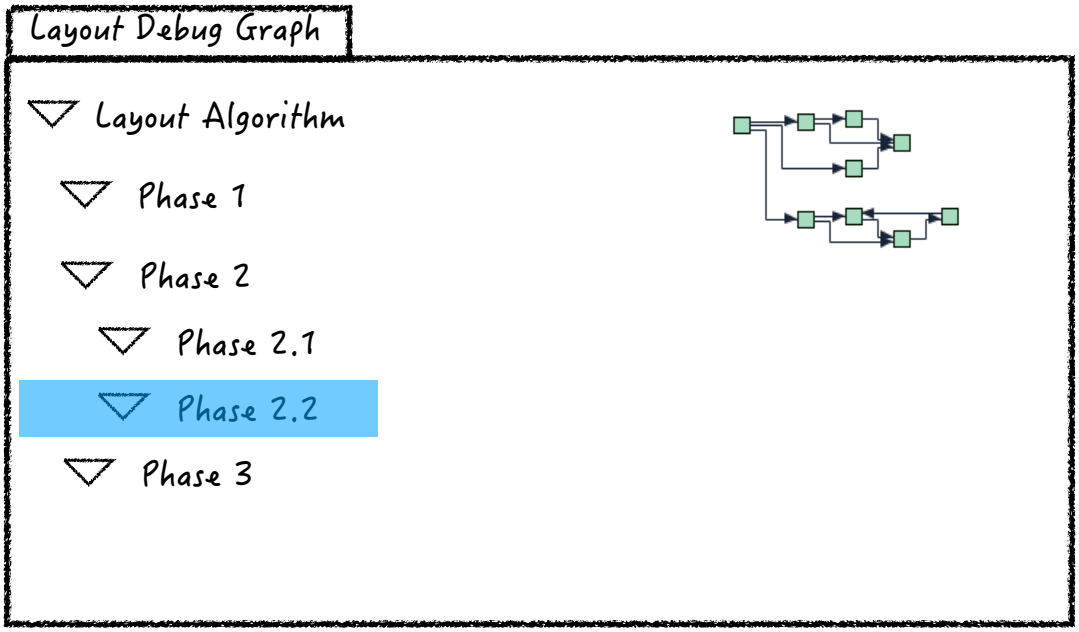


Figure 4.4. Design of the new layout graph view.

graph in the list viewer.

Having these features implemented for the layout graph view would greatly improve its debugging capabilities. Viewing intermediate graphs might make it possible to identify flaws more precisely in an algorithm. One would also be able to extract simple graphs from a hierarchical one and examine them isolated. The proposed way of displaying graphs would make it possible to view the result of an algorithm without it having assigned width and height to the graph yet. The view also makes it easy to spot, when an element misplaced elements outside of the graph.

4.5 An Early Design

In the beginning of this thesis's work a design for a debugging tool was presented, which had all functionalities within one view. The design can be seen in Figure 4.5. The progress monitors were supposed to be placed on the left and upon selection of a so-called breakpoint the logged graph should show up in the middle of the view. On the right were multiple tabs of which only one could be seen at a time. The debug output would have its own tab within the view. The design also had some other tabs being placeholders for further ideas, which might have come up during working on the thesis. In the end the whole idea of including all features in a single view was abandoned, because it would be a very static construct. I decided to make separate views to better utilize the space for each view and give the developer better control over the arrangement of debug information within their workbench.

4.6 Comparison to KIELER's compiler view

While working on the designs it was suggested to reuse KIELER's compiler view. In combination with the KlighD view it serves the same purpose than the previously proposed design from Figure 4.4. In

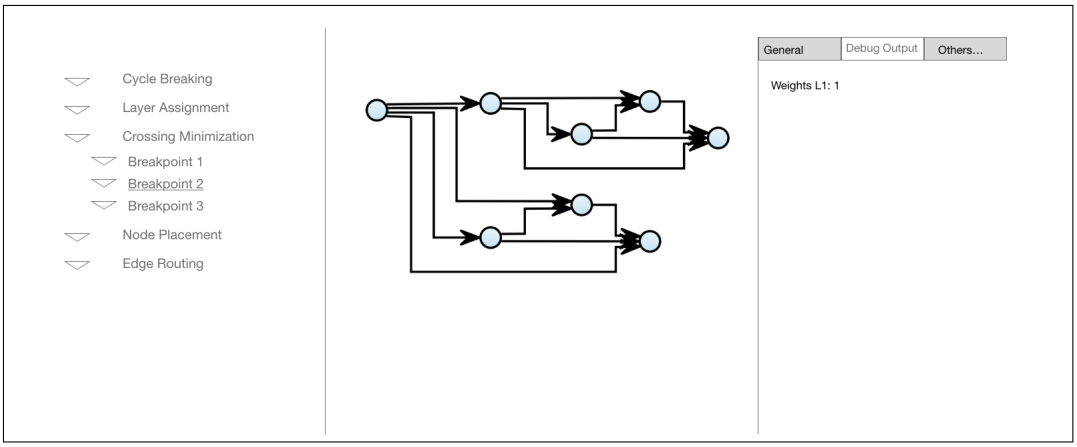


Figure 4.5. First design of a new debug tool.

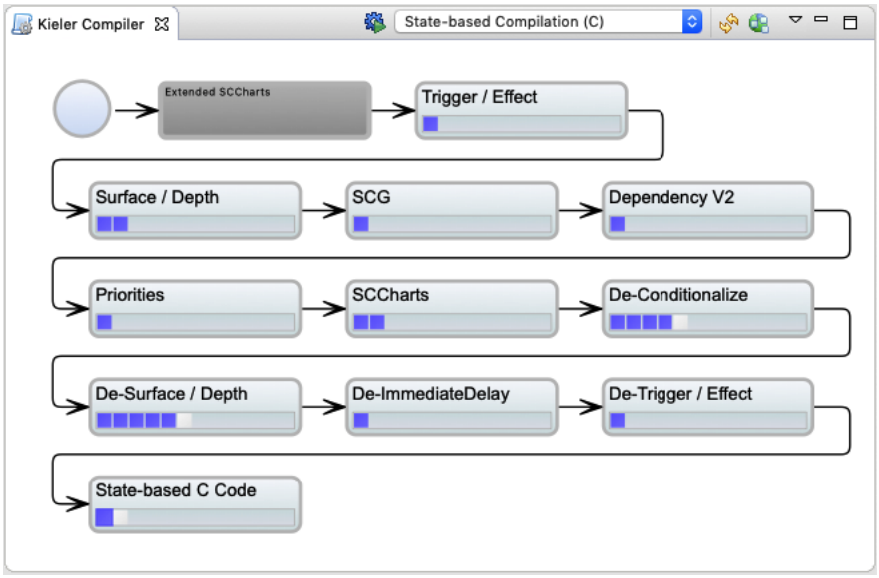


Figure 4.6. KIELER compiler view

the view, shown in Figure 4.6, one can see multiple elements aligned in five rows. These are states the compilation of an SCChart goes through when being transformed to code. The element in the top left corner describes the start of the compilation. The second element is an expandable area which reveals its contained compilation states when double clicked. All other elements have no hierarchy and contain clickable intermediate steps within their state. When clicking on one of them, the step is visualized in the KlighD view.

This whole concept of viewing intermediate states is applicable to ELK. Such an expandable state could be a monitor containing sub monitors for example. Still, it was decided against using this way of displaying the intermediate states of a graph. First of all, it does not fit the general theme of how ELK's debug tools look and feel. The layout time view existed beforehand and displays the algorithm's monitors in a tree viewer within a table. There is no reason to have an inconsistent user

guidance throughout the views. Another minor disadvantage is that graphs are only identifiable by their containing monitor and position in them. A list viewer displaying the tags logged along with the graph gives more clarity about which graph is displayed. Also the compiler view uses layout algorithms to arrange the states as they are basically a graph which is supposed to fit in the available space of the KlighD view. This is especially bad when one is debugging the algorithm the compiler view uses to arrange the compilation states.

Chapter 5

Implementing Concepts

This chapter is about the implementations of some of the concepts presented in Chapter 4. Note that this thesis's work does not improve anything in regards to the development setup as this rather is a problem of distribution than one of implementation. Before diving into the main topics of improvements some general additions are presented.

5.0.1 Logging Preference

A new preference was introduced to the ELK framework. It got added in order to give developers a tool to control whether output is logged. For that it got added within the preference page of ELK, which can be found in the Eclipse preferences as demonstrated in Figure 5.1. Thus some additions have been made to the diagram layout engine located in ELK's service plug-in. A string identifying the preference was added and the creation of progress monitors now includes configuring them according to the new preference. The preference in general was added to ELK's UI plug-in. A preference initializer takes care of setting up the preferences with default value, which has been set on `false` for the logging preference.

5.0.2 Logging Preference Info Bar

A general info bar view element, which can be utilized by any debug view, has been added in a new package called `org.eclipse.elk.core.debug.views.util` within the debug plug-in. It can be used by

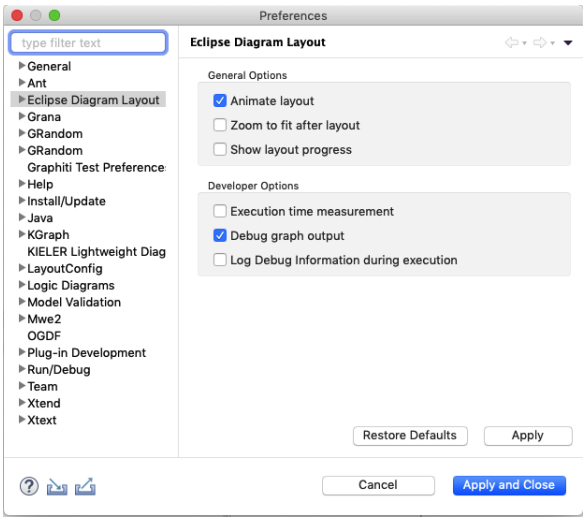


Figure 5.1. ELK preference page displayed inside of the Eclipse preferences.

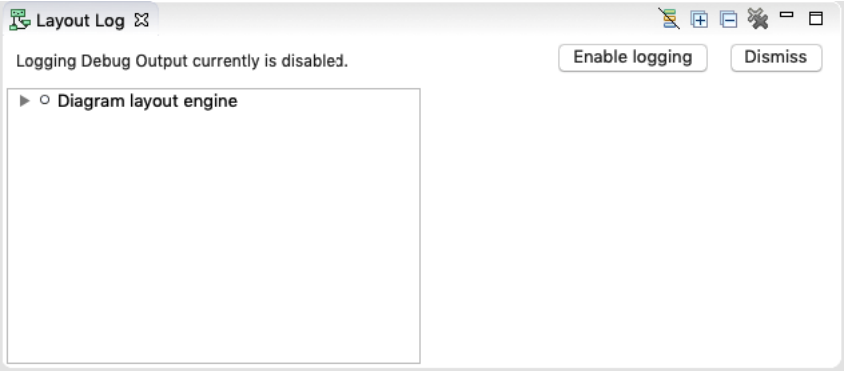


Figure 5.2. Layout log view displaying the info bar right after opening.

a view to notify the user that the logging preference is disabled as shown in Figure 5.2. The new class `LoggingInfoBar` provides a method, that takes the composite in which the info bar is supposed to be placed in. Before setting it up, the state of the logging preference is checked and if it is disabled, the info bar's creation process is dismissed. It offers the possibility to directly enable the preference without entering ELK's preference page or to dismiss it by clicking the buttons on the right. Either hides the view while the *Enable Logging* button does what is says.

5.1 Project Setup

This section revolves around setting up *layout algorithm projects* for ELK and the realization of suggested improvements. Chapter 2 introduced layout algorithms, as well as their setup for ELK. Such a setup can be quite tedious as it involves a considerable number of manual steps.

Within a general plug-in project the developer has to perform following steps in order to create a runnable layout algorithm:

1. Add ELK's dependencies.
2. Create a base package.
3. Create a layout provider class extending `AbstractLayoutProvider`.
4. Create a `melk` file within the base package and add the Xtext nature when prompted.
5. Add an extension to the project.

Developers that are not familiar with the procedure and the framework might be overwhelmed. Many people who set up new algorithm projects are new to ELK, for example students confronted with the task while taking the lecture *Automatic Graph Drawing*. Time spent on setting up a working project is time not spent actually creating the algorithm.

As the whole development cycle also includes the project setup, I introduce a new tool in order to simplify the process and make developing layout algorithms for ELK more accessible.

5.1.1 Layout Algorithm Project Wizard

The proposed tool is the new Layout Algorithm Project Wizard. Wizards in general are sequences of dialog boxes leading the user through complex tasks. An example is a setup wizard that often

occurs when installing programs on a computer. They are also common in the Eclipse IDE for resource creation. To be precise, the new tool is only an extension of the Eclipse plug-in project wizard. It adds another template and an additional wizard page.

To set up a new algorithm project, the *Plug-In Project Wizard* that already exists in Eclipse must be started. That can be found under `File > New > Project`. In the opened dialog box select *Plug-in Project* and hit *Next*. The first page of the plug-in project wizard opens (Figure 5.3a). There one only has to enter a *project name* and hit *Next* again.

On the second page (Figure 5.3b) the user must make sure that the check box *This plug-in will make contributions to the UI* is unchecked. A layout algorithm also is no rich client application, so the selection should be on *No*.

Hit *Next* and a list with plug-in templates shows up (Figure 5.3c). Choose *Layout Algorithm* and hit *Next* again.

A rather blank page shows up (Figure 5.3d). It only contains a text input for the algorithm name. The wizard tries to derive it from the project name, but it could be set manually as well. When clicking *Finish* a new plug-in project with the selected template is generated and can be found in the workspace.

A moment later a `src-gen` folder also is generated. Unfortunately the setup of the project is not finished yet. In the—now most likely opened—manifest editor one must enable the check box *This plug-in is a singleton*, because ELK requires it.

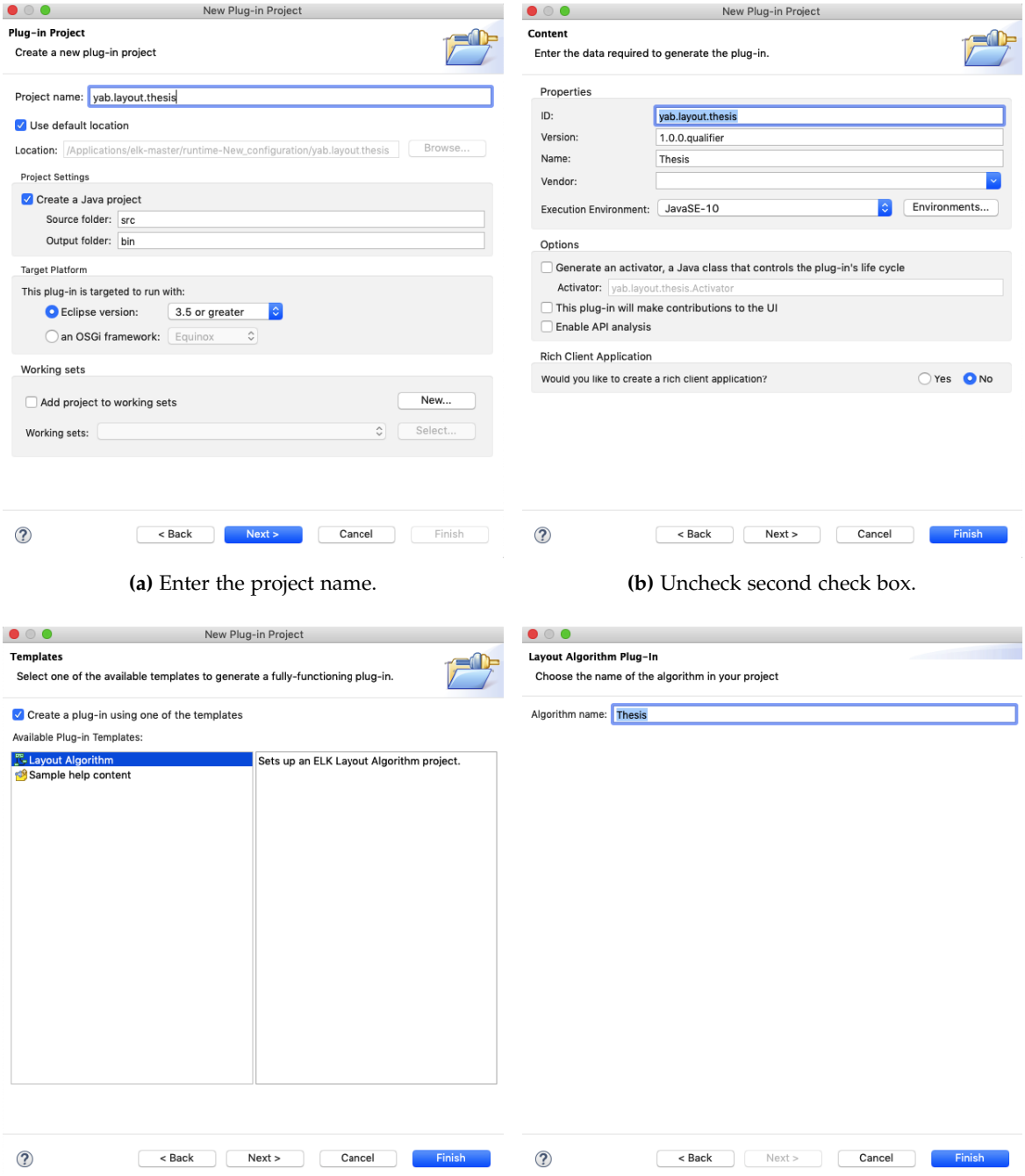
The layout algorithm is now ready to execute. The generated layout provider already contains some code that serves as a tutorial, including basic placement of nodes, routing of edges and usage of progress monitors.

```
1  public class ThesisLayoutProvider extends AbstractLayoutProvider {
2
3     @Override
4     public void layout(ElkNode layoutGraph, IElkProgressMonitor progressMonitor) {
5        // Start the progress monitor
6        progressMonitor.begin("Thesis", 2);
7
8        ...
9     }
10 }
```

It was taken from the documentation's section about algorithm development and was slightly modified in order to show the use of ELK's debug tools.

The same goes for the `melk` file that already has some options registered:

```
1  package yab.layout.thesis
2
3  import yab.layout.thesis.ThesisLayoutProvider
4  import org.eclipse.elk.core.math.ElkPadding
5
6  bundle {
7     metadataClass ThesisMetaDataProvider
8     idPrefix yab.layout.thesis
9  }
10
11 option reverseInput : boolean {
12    label "Reverse Input"
```



(a) Enter the project name.

(b) Uncheck second check box.

(c) Choose the layout algorithm template.

(d) Change the algorithm name.

Figure 5.3. Usage of the new layout algorithm wizard step by step.

```
13    description
14       "True if nodes should be placed in reverse order of their
15       appearance in the graph."
16    default = false
17    targets parents
18 }
19
20 algorithm Thesis(ThesisLayoutProvider) {
21    label "Thesis"
22    description "Please insert a short but informative description here"
23    metadataClass options.ThesisOptions
24    supports reverseInput
25    supports org.eclipse.elk.padding = new ElkPadding(10)
26    supports org.eclipse.elk.spacing.edgeEdge = 5
27    supports org.eclipse.elk.spacing.edgeNode = 10
28    supports org.eclipse.elk.spacing.nodeNode = 10
29 }
```

5.1.2 Implementation

While implementing this tool, two different approaches were considered. The first approach was about registering a whole new wizard using the `org.eclipse.ui.newWizard` extension point which might be more flexible but also adds a lot of complexity in terms of its development. The whole plug-in project creation would have to be implemented which is rather complex. Thus a better way of implementing a proper project setup was found. The extension point `org.eclipse.pde.ui.pluginContent` is offered by the Plug-in Development Environment (PDE) of Eclipse which provides tools for the creation of plug-ins. Extending this extension point coupled with implementing the `IPluginContentWizard` interface lets one add templates to Eclipse's plug-in project wizard. It allows to execute code after the original plug-in project wizard has set up the base of the plug-in and thus makes it possible to add additional content to it afterwards. For that the interface provides an API that allows almost every aspect of the plug-in project to be customized. The interface requires the implementing class to also extend the abstract class `org.eclipse.jface.wizard.Wizard`. A schematic in Figure 5.4 shows the correlations between the wizards and the project and file creation.

The modifications to the base plug-in project happen in multiple steps according to the user's interaction. When hitting *Next* on the template selection dialog box (Figure 5.3c) the wizard will be initialized and is able to retrieve information from the previous wizard pages. Using these information it derives a proposal name of the algorithm for its own wizard page from the plug-in's name by picking the string after the last dot and capitalizing it. After this, wizard pages can be added. As not much information is required only one page is added with only one single text input for changing the algorithm's name, if desired. When then user hits *Finish* multiple actions happen.

New dependencies are introduced to the plug-in project including the necessary ones from ELK. This is done by providing an array holding the IDs of the dependencies to a method provided by the interface. Afterwards any other modification to the project is done as the project's file system can be accessed. A base package is created with the correct directory structure according to the plug-in's name. The Java class for the layout provider gets created. This is done by using the template feature of *Xtend*. The Java class generated from the Xtend file contains a method that takes string inputs and enters them into the fitting spots within the file content. The returned string then can be entered in the

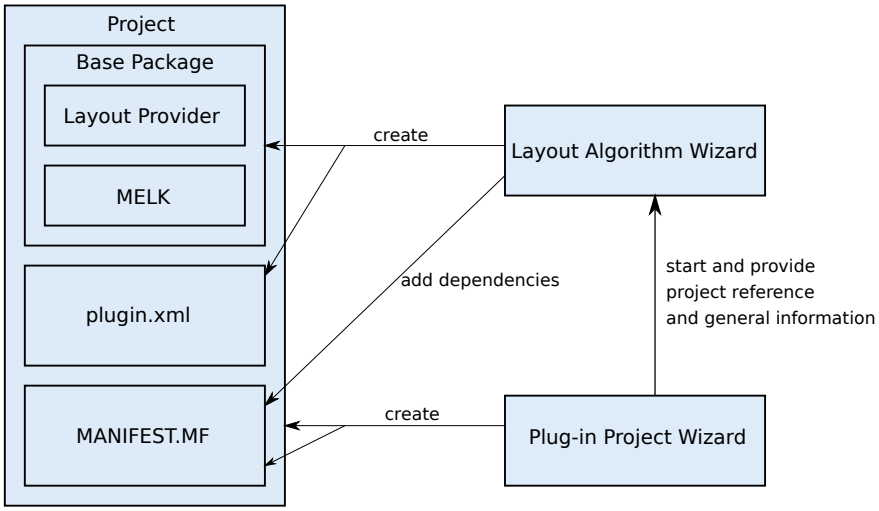


Figure 5.4. Wizard Correlations.

created file. The same procedure is performed for the `melk` and `plugin.xml` file. In the end the Xtext nature needs to be added to the project in order to generate the meta data provider classes later.

The wizard was added to the `org.eclipse.elk.core.debug` plug-in and got its own package named `wizard`. Also introduced is a sub package `templates` containing the Xtend templates for file creation.

Unfortunately the API offers no possibility to add an extension to the plug-in. Therefore the `plugin.xml` had to be created and appropriately filled manually in order to extend the necessary extension point properly. This may also be the reason why the plug-in does not get set as singleton because this depends on the extended extension points. When adding the extension by hand via the manifest editor the flag gets set automatically. In case a way might be found to add an extension via the API it might solve the singleton problem as well.

If the way layout algorithms integrate with ELK changes in the future by dropping the use of the infrastructure around extensions, one only has to remove the `createPluginXML` method within the `performFinish` method of the `AlgorithmProjectWizard` class and replace it with an implementation generating whatever is required then. In case anything in the files created needs to be changed, the Xtend templates are easy to adapt as shown in Section 2.4. The order of the operations after hitting *Finish* on the last wizard page does not matter as the results of each are not needed for any other action. However, some changes made in one process might entail the need of changing something in another one because some created components have references to each other. For example if some changes to the layout provider class are made such as the class name or package, the corresponding changes have to be made in the `melk` file as well.

5.2 Textual Debugging

Some inconveniences in the ways textual debugging currently is done in ELK layout algorithm development were detected. The console output is not visible in the same window causing it to appear and cannot be structured properly and isolated from other output. This is subject to change and therefore I introduce a new tool to the framework and show how it retrieves and displays information under the hood.

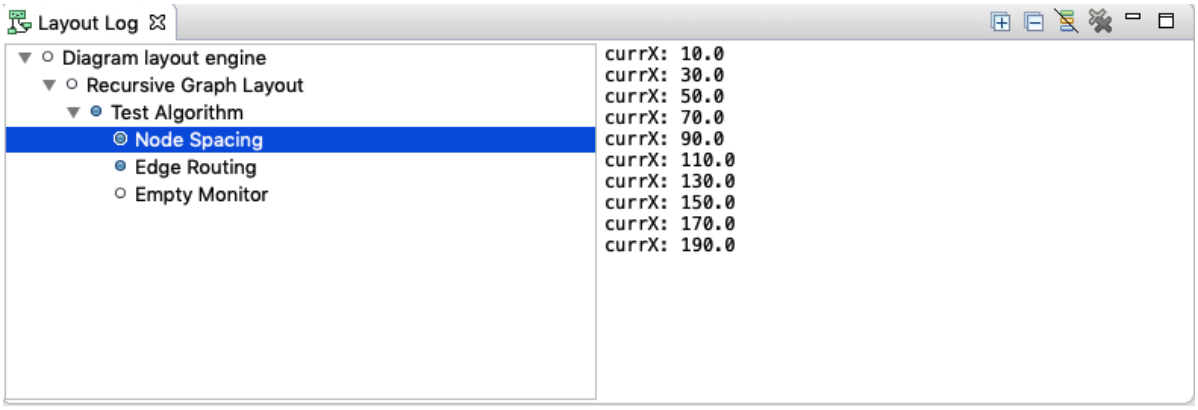


Figure 5.5. Layout Log View containing some output that is being made in the tutorial code.

5.2.1 Layout Log View

The *layout log* view is supposed to solve problems around displaying and viewing textual debugging output during layout algorithm development. It runs in the target platform and is able to show output generated by an executed algorithm. The view can be seen in Figure 5.5. Its design is inspired by the layout time view, which was introduced in Chapter 3. The tree viewer on the left hand side serves as selection input, where elements representing monitors can be expanded in order to view their children. The indentation stems from the elements' parent-child relationship. Every element is annotated with either a blue or a white dot indicating whether the corresponding monitor contains output. When a monitor is selected its output shows up within the blank space on the right. Multiple monitors can be selected while holding the shift key on the keyboard, which causes the view to display the input of all selected monitors at once.

Furthermore the view offers some actions on the top right. From left to right those are: Expand all elements in the tree viewer, collapse all elements in the tree viewer, filter the output, and remove all input from the view. The interesting one is the filter action, that excludes every monitor, which neither holds output nor has children of any depth holding any, from showing up in the selection screen. This might turn out to be useful, if the observed algorithm uses a deep progress monitor structure. When right-clicking on an element in the tree viewer, a small context menu pops up having a single option *delete*. It lets one remove a monitor with its children monitors from its parent. If a parent is chosen, the whole structure is removed from the list of monitors.

Within the code the output can be given directly to the progress monitor and for that purpose the method `log(Object object)` was added. This allows the monitor to take any input regardless of its type by converting it to a string before storing.

The whole functionality can be enabled by ticking the newly introduced check box *Log debug information during execution* in the ELK preferences introduced in Section 5.0.1. In case the layout log view gets opened while the preference is disabled, the aforementioned info bar shows up.

5.2.2 Implementation

The view is an addition to ELK's debug plug-in, thus it got placed in the package `org.eclipse.elk.core.debug.views.logs`, which also contains the package `action`. Besides the view components, a data structure *log container* was added that holds the required information retrieved from the progress monitors, which have been modified as well. To be precise, changes to the `IElkProgressMonitor`

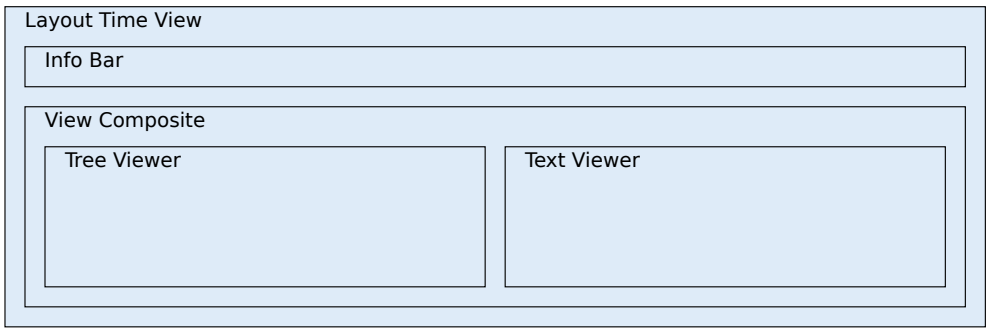


Figure 5.6. Structure of view components of the layout log view.

interface and its implementation `BasicProgressMonitor` were made. The interface now requires the implementing class to have the `log(Object object)` method and a getter `getLogs()` returning a list, hence the `BasicProgressMonitor` class now implements them. The former method calls `toString()` on the given object and adds the result to a list of strings, which is returned by the latter method. The logging is only done when the option for that is enabled in the preferences.

Following the data flow, the progress monitor structure ends up being disassembled by the views of ELK's debug plug-in including the new layout log view. Each time the data is received a static method of it is called, which starts an asynchronous operation checking, if the view is currently active. In case it is, the data will be given to the active view, otherwise it is dismissed.

The layout log view creates *log containers* from the progress monitors. These mirror the tree structure of them by going through recursively while only retrieving the logs. So for every progress monitor one log container gets created, even if it does not contain any logs. The class cannot be instantiated with a public constructor, but it offers a static method `fromProgressMonitor(IElkProgressMonitor monitor)` creating the whole tree structure and returning its root node. This is done twice so the view can hold a filtered and an unfiltered version of the new log container structure to quickly change between them. The filtering gets done after creation. A Depth First Search (DFS) algorithm will go through and delete all leafs not containing logs or parents leading to ones doing so. In detail it goes through a branch until finding a leaf and removes it, if it does not contain any logs. If all children of a parent are removed, itself will be checked on its logs and removed, if it does not contain any. Each of those will be added to a list holding either filtered or unfiltered logs. To clarify, these lists hold the roots of all of those tree structures. These lists are the input for the tree viewer and depending on the state of the filter button, one of these is set as input for the tree viewer. Every time the lists get an addition or the shown list changes the tree viewer has to be refreshed in order to display the change. A selection listener is registered on the tree viewer appending all strings of every selected monitor and setting it on the text view.

The view has a simple hierarchical structure for its view elements placed inside as shown in Figure 5.6. The top level element is a composite having a grid layout with a single column. Thus elements get placed on top of each other, such as the *Info Bar* and the *View Composite*. The latter has a fill layout, which causes its children to inflate the whole view. Thus the tree and text viewer in the view composite always stay at the maximal possible size when resizing the view. The info bar only shows up when the logging option is disabled on start of the view. If it gets switched off while the view is opened, the user is not notified again. Both viewers supports scrolling in horizontal and vertical direction in case their content span outside their boundaries.

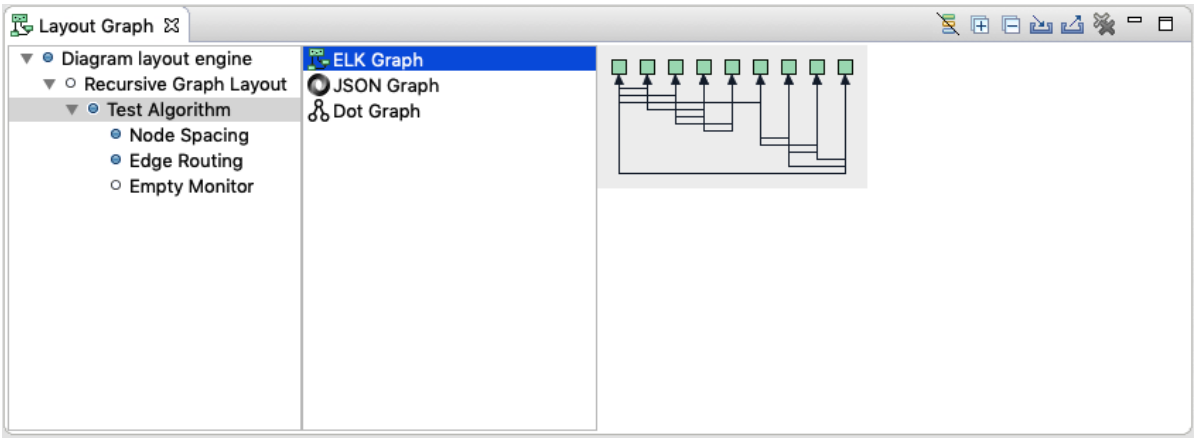


Figure 5.7. The new layout graph view.

5.3 Graph Viewing

The *layout graph* view is a tool for viewing graph drawings after an algorithm's execution has finished. When digging through the source code, one might come across the class `GraphRenderer`. This is the component which draws the graph onto the canvas. Amongst other things its class comment says: "This is primarily a debug tool". In this section I introduce the changes to the layout graph view which were made in order to transform it from a plain graph viewing tool into an actual debug tool.

5.3.1 Layout Graph View

On the surface the new layout graph view has three main view elements instead of just one. In addition to the graph canvas a tree viewer and a list viewer got added, as shown in Figure 5.7. It is now able to display multiple graphs from one single algorithm execution. Those can be either intermediate states of graphs, such as the ones after each iteration of the force layout algorithm [FR91], or any other auxiliary graph that might be used within an algorithm. The final graph is still automatically logged and is found in the top level monitor.

There is a significant change to the way the view works. Earlier the view received the final layout of the graph and displayed it. Now it does not receive the graph data directly, but the progress monitor holding it. To log a graph with the monitor, one has to use the `logGraph(Object graph, String tag, GraphType graphType)` method. The graph can be of any type but will be interpreted according to the assigned graph type. Currently there are three supported graph types: *ELK*, *JSON*, and *dot*. Since JSON and *dot* are textual formats, both are handled as strings. This made it necessary to distinguish between them by adding the enumeration `GraphType` in order to tell the receiver to parse them correctly. The `tag` parameter serves a possibility to identify the graph later on. In the view it is used to name the graphs in the list viewer. If developers want to log an ELK graph, they can use the convenience method `logGraph(ElkNode graph, String tag)`.

When an algorithm execution happened the progress monitor shows up in the tree viewer, similar to the other ELK views. Corresponding to the layout log view's tree viewer, the elements' dots indicate whether the monitor holds any graphs. On selecting an element, its graphs are listed in the list view and the first graph is set on the graph canvas automatically. The element's name is the tag logged along with the graph and the graph's type is represented by the icon as shown in Figure 5.7. Selecting

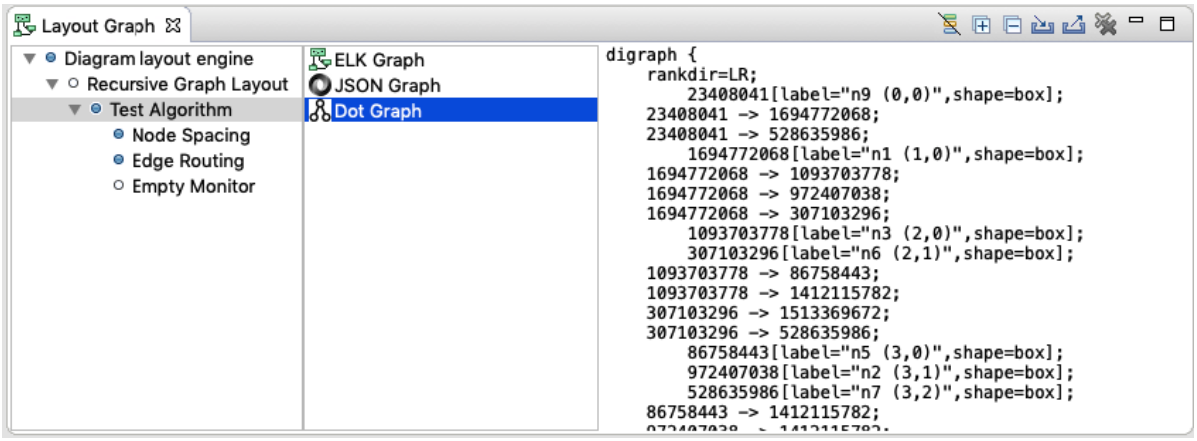


Figure 5.8. A textual graph representation.

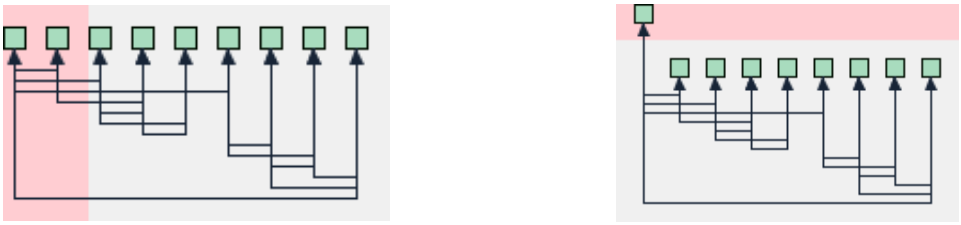
(a) A graph drawn with no dimension assigned. **(b)** A graph drawn with dimensions being too small.

Figure 5.9. Graphs with erroneous sizes assigned.

a list item causes the chosen graph to appear on the graph canvas to the right. Currently only ELK and JSON graphs are visualized as drawings while a *dot* graph is represented textually as demonstrated in Figure 5.8. If the graph type is not applicable to the actual graph object an error message highlighted in red shows up in the canvas.

Actions such as importing a graph and exporting its drawing as an image were kept from the old view. They were adjusted to the new structure of the view and work as before. When loading a graph, a new entry in the tree viewer is created and named “Loaded from file”. Every graph shows up in this container and takes its name from the file it was loaded from. Besides those actions, others were added which are basically the same as for the layout log view in Section 5.2: Filter the monitors, expand and collapse all elements of the tree viewer, and clear the view from all data. The context menu for deleting single monitors is also available.

As for displaying a graph, it is now fully drawn regardless of the dimensions assigned to it, as shown in Figure 5.9. The light grey area shows the graph’s assigned dimensions, while the red area represents the additional space needed to include all graph elements within the graph’s boundaries. Figure 5.9a shows a graph having no height and width at all; thus the whole area is highlighted in red. Figure 5.9b shows a graph with its boundaries not wide enough to include all of its elements. Another new feature is to draw graphs with elements located at negative coordinates, which is demonstrated in Figure 5.10. Under special circumstances the coordinate system’s origin will be marked. If the graph has no size assigned and contains graph elements with negative coordinates, a small corner marking $(0|0)$ is drawn, which can be seen in Figure 5.11. This is done to help the developer to spot problems in the algorithm by viewing intermediate graphs, which will have no dimensions assigned as it is



(a) Graph elements at negative horizontal coordinates. **(b)** Graph element at a negative vertical coordinate.

Figure 5.10. Graph with elements at negative coordinates.

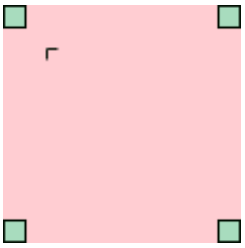


Figure 5.11. The origin of the coordinates is marked.

usually done when the layout is finished.

This view also supports the aforementioned info bar notifying the user if the logging preference is disabled.

5.3.2 Implementation

The view structure is more complex than the structure of layout log view. An overview of the composites and view elements is shown in Figure 5.12. The top level view contains the *Info Bar* and the *View Composite* and has a grid layout with one column to show the former above the latter. If the info bar is not shown, the view composite takes the whole space. It is divided into two sections, represented by the *Selection Composite* and the *Graph Viewing Composite*. The selection composite takes one half of the view and contains a tree viewer as well as a list viewer. Since a usual list viewer does not allow for icons to be placed next to its elements, a table viewer was used instead. Only one column of its table is filled in order to still look like a list viewer, but with icons. The graph viewing composite claims the other half and holds two other view elements, which are not being drawn next to each other. Instead only the view supposed to be seen is drawn while the other is hidden. This is achieved by the use of a stack layout which allows to dynamically change the view element shown in its composite. Thus it is possible to switch between the scrolled composite containing the graph drawing canvas and the textual graph canvas by defining them as top of the stack. The latter is just a text view displaying characters, while the former allows the contained drawing canvas to be as big as required and further provides scrolling if necessary.

Furthermore I made additions to the interface `IElkProgressMonitor` and its implementation `BasicProgressMonitor`. The interface now requires the above mentioned `logGraph(...)` methods as well as the getter `getLoggedGraphs()`. It returns a list containing `GraphStore` objects, which is a simple data class holding a graph, its tag, and graph type. It was added to the package `org.eclipse.elk.core.util`, which also contains the progress monitor files and the new enumeration `GraphType`. When `logGraph(...)` is called and its parameter `graph` is not `null`, a graph store is created and added to a list held by the

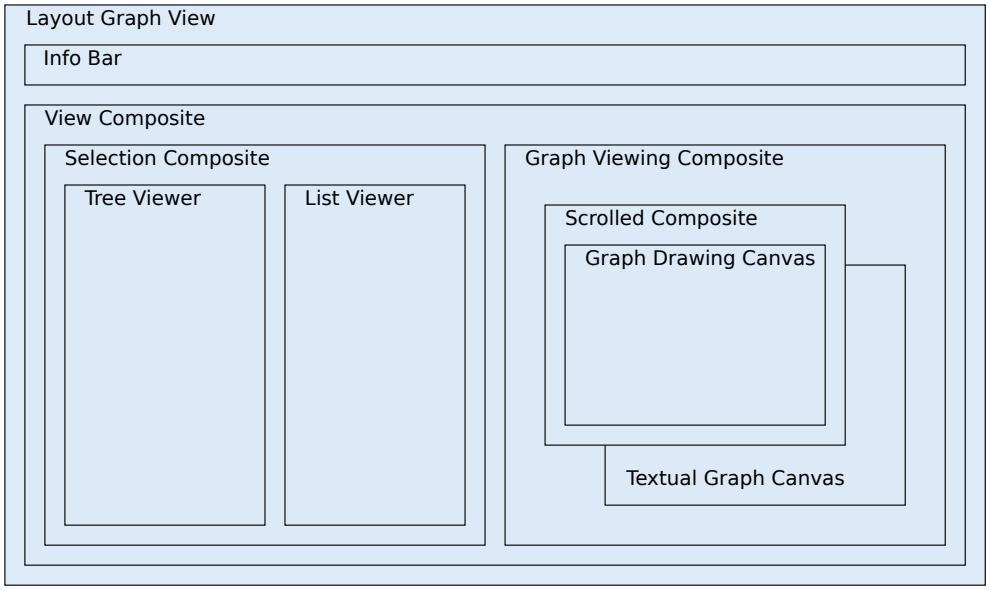


Figure 5.12. The view element structure of the layout graph view.

basic progress monitor.

Some modifications to the information retrieval were made in order to make the view able to receive multiple graphs from one algorithm execution. Earlier the view received a layout mapping from which a single ELK graph was derived. Now it gathers the information from progress monitors. A new data class called `GraphContainer` was added, which holds a list of graph stores and can form a tree structure. It is created from the progress monitor structure which it mirrors while only retrieving its name and graphs. For this matter it has a static method `fromProgressMonitor(IElkProgressMonitor monitor)` iterates over the structure recursively while recreating it with graph containers. The graph container is the counterpart to the log container introduced in Section 5.2.

When ELK's debug plug-in receives the results of an algorithm, it calls a static method `addGraphs(IElkProgressMonitor monitor)`. It starts an asynchronous task finding the opened view. In case of success two graph container structures are created of which one is filtered with the aforementioned DFS algorithm. Afterwards they are added to the corresponding list and the tree viewer gets updated in order to show the change. It shows either the filtered or unfiltered graph container list. When a graph is loaded by the user via the load action, a separate log container is created which only contains on single graph store with the given graph. It is added to both lists.

The tree elements can be selected, which triggers a listener causing the associated graphs from the progress monitor to appear in the list view. The input for the list viewer is a list of graph stores. It also has a listener attached tasked with initiating the displaying of the graph. The type of display is determined by the graph type. It extracts and casts the graph to the correct data type and forwards it to a method taking care of placing the right canvas on top and drawing it. As for an ELK graph no further computation after casting is needed and it can be passed directly to the visual graph canvas in order to be drawn. If a string got casted and is supposed to be a JSON object, ELK's JSON utility class `ElkGraphJson` is used for parsing. The result is an ELK graph which is then displayed on the graph canvas. In case the wrong graph type was logged, the cast fails, or the string is not parseable, the

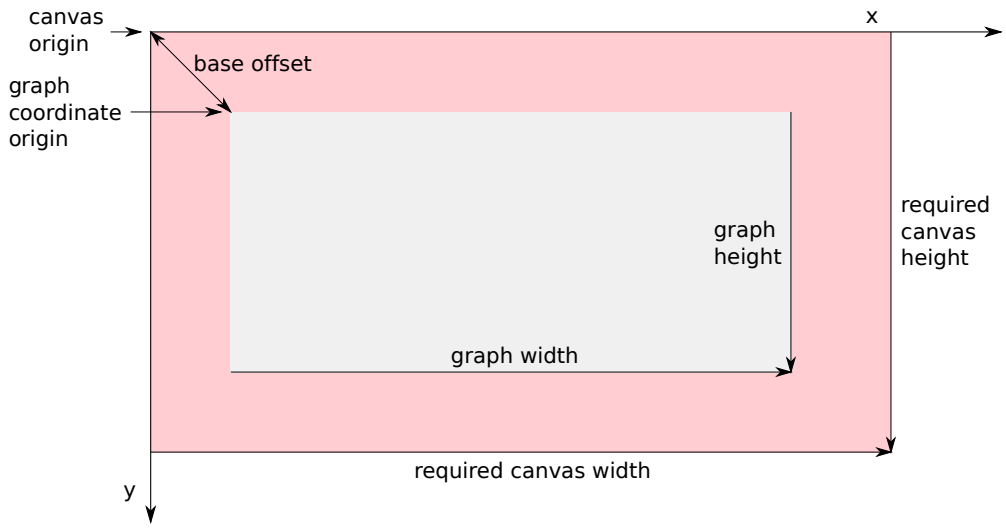


Figure 5.13. Utilization of the graph drawing canvas.

textual canvas displays an error message. If at a later point in time one wants to add a visualization for the *dot* format one could reuse code from the pragmatics plug-in `de.cau.cs.kieler.klay.debugview`. An image canvas would need to be added and used within the stack layout. If one wants to introduce new graph types displayable by the view, one must only add a new entry to the enumeration and add a new case for handling the corresponding type of graph. The list label provider will work regardless of an unknown graph type, but only provides icons for known types. So a new icon image must be placed in the plug-in's `icon` directory and loaded in the label provider.

For drawing graphs three classes are relevant: `GraphRenderingCanvas`, `GraphRendering Configurator`, and `GraphRenderer`. These existed beforehand but have seen some additions in this thesis work. Earlier the graph rendering canvas derived its width and height from the graph's assigned dimensions. For graphs without a dimension this resulted in an empty view. Determining the canvas's size now takes graph elements into consideration rather than the graph's width and height. A simple algorithm iterates over all graph elements and tries to find the maximum and minimum vertical and horizontal coordinates. When looking at the maximum ones the graph element's own width and height is taken into consideration. Note that the top-level graph's dimensions are also examined. This is done in order to still assign the canvas the graph's width and height if it is properly sized and thus contains all children. When the final minimum and maximum coordinates are found, the difference between them is calculated. The results are now the width and height of the drawing canvas. While having the minimal coordinates at hand, the algorithm uses these to set a base offset for the graph in the renderer. For example, if the minimal x coordinate is -10, the graph's drawing needs to start at a horizontal coordinate of $+10$ in order to include its elements located at negative horizontal coordinates relative to its origin coordinate. A sketch to clarify the whole process is demonstrated in Figure 5.13.

The graph rendering configurator holds font sizes and all color values for graph elements. Upon instantiation the graph renderer is handed such a configurator to derive those values from it. Two colors were added for the new backgrounds. While the actual graph's background has a slight gray shade to it, the area outside a graph's dimension is colored red.

The graph's actual rendering remained unchanged. However, some actions now take place before it.

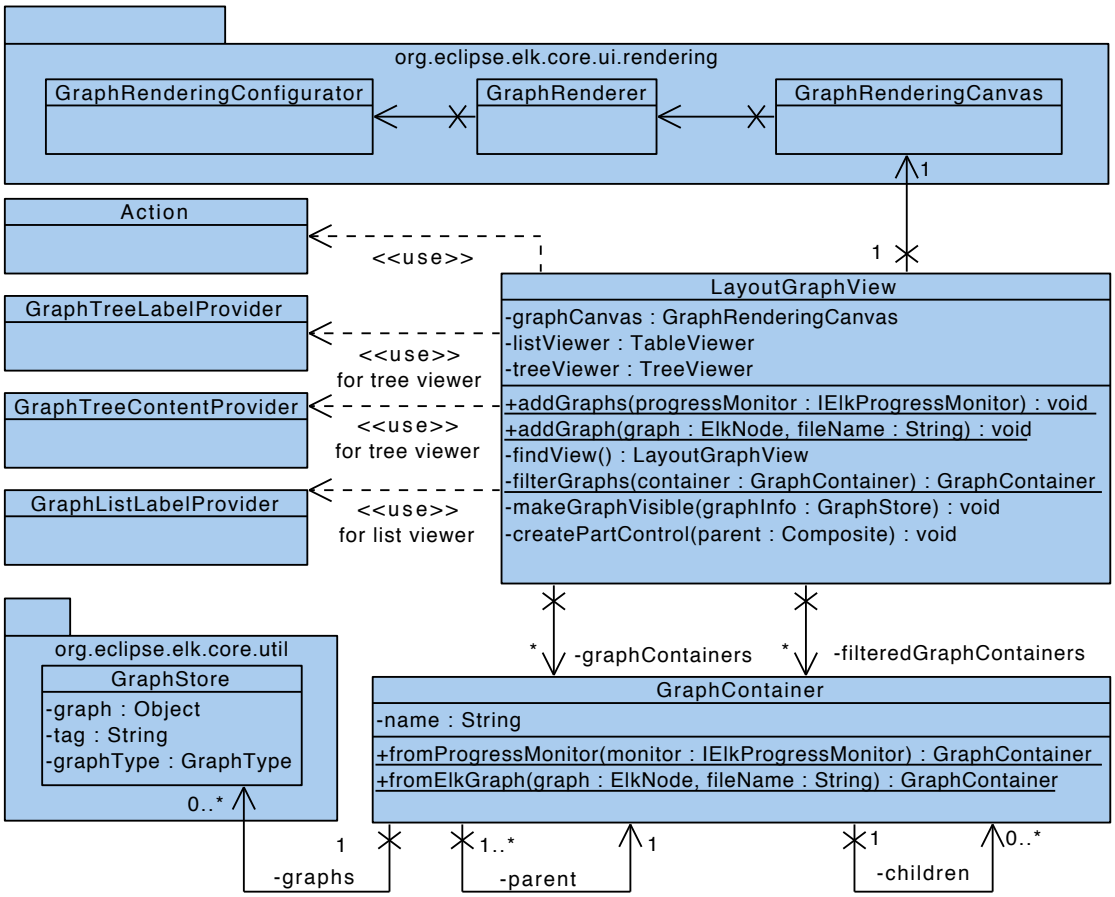


Figure 5.14. Class diagram for the layout graph view.

First of all the whole canvas is colored red. Afterwards the graph's dimension is derived and its area painted on top of the red coloring. The vector describing this offset was available prior to my changes but always initialized at the origin's coordinate. The graph renderer already took it into account when drawing. Thus there was no need for further changes to properly place the graph within the canvas, since the correct base offset is already set by the graph rendering canvas. After drawing the graph, it is checked, whether the graph has no dimensions and the base offset is greater than zero. The mark indicating the graph's origin coordinate is then drawn if necessary.

A class diagram containing all added and modified classes except the progress monitor's can be seen in Figure 5.14. The two packages are from other ELK plug-ins, which the debug one has dependencies to. Classes outside of them are within the `org.eclipse.elk.core.debug.views.graph` package. For greater clarity the *Action* class is a placeholder for the six classes implementing the actions in the tool bar. These are located in a seperate `action` package. A new dependency `org.eclipse.elk.graph.json` was added to the debug plug-in in order to parse graphs from JSON files.

Chapter 6

Evaluation

In this chapter I evaluate my implementations with regards to improvements compared to the prior state, usability, and their possible use cases. First and foremost I need to mention that this evaluation mostly bases on hypothetical scenarios, because no user study could be conducted. A study can be done in future by letting algorithm developers use these tools and filling out a survey for example. Possible participants could be found in the *Automatic Graph Drawing* lecture, where they sooner or later are confronted with the task of implementing layout algorithms. They can be surveyed towards the end of the lecture and data can be collected about the frequency of usage of specific debug tools, their usability, and ideas to further improve the developing experience. Some diagnostic data might be collected and sent to a server, which can be only done if people agree with it.

6.1 Project Setup

The goal of the layout algorithm project wizard's implementation was to speed up the project setup as well as making it simpler and more accessible. I started a very simple experiment where I gave a computer science student the task of setting up a layout algorithm project for ELK. He was not very familiar with it, so he used the ELK documentation along the way. The time was taken from when he first started the *new project* wizard to when he finished setting up a runnable algorithm. The algorithm was not supposed to do anything, meaning the `layout` method could be left blank. The only criterion was an algorithm executable in the target platform. The student made some mistakes when naming the meta data class within the `melk` file and registering it in the manifest. Noticing the mistakes took some time, so he finished the project setup in roughly 23 minutes. I also did the test myself to simulate the time a more experienced ELK developer might need. Without needing to look up any instructions, it took me 4 minutes and 3 seconds. The measured times are not necessarily very precise and the sample size is small, but they provide an idea about how long the setup might take.

With the new wizard the time needed to setup a runnable algorithm project is heavily reduced, as it is created within a minute. When calculating the time-savings with that value, the needed time reduces by approximately 95% for inexperienced developers and 75% for ones being experienced with the project setup.

This is obviously not the only advantage of the new wizard. Developers do not need to handle anything related to Eclipse plug-ins anymore besides clicking a check box in the manifest editor, which opens automatically right after the wizard has finished. This reduces the number of steps as well as their complexity and makes developing layout algorithms for ELK more accessible.

6.2 Textual Debugging

With the introduction of the new layout log view, some inconveniences during usage of textual debug output were taken care of. The output appeared in a different window than the algorithm was executed

in. It could be found within the console of the development platform, which also may display output from other code or stack traces from errors. I fixed these issues by introducing this new tool.

To log something one has to call `log(...)` on a monitor, which causes output to appear in the view. Logging is possible without starting and stopping the monitor, but it will show up as *Unnamed* in the view. So in case of larger progress monitor structures, they should be at least started, which requires the developer to assign a name to them. One example of such a large structure is ELK's layered algorithm which creates a monitor for each of the 20 layout processors used in its standard configuration. How the developer manages the algorithm's output is up to them, because the monitor structure can be arbitrarily expanded. This way the algorithm's output or even its specific phases are isolated from any other text in the console. The clarity this provides can be further enhanced by filtering the monitors. If only a few monitors hold output, the verbosity of the whole structure is cut down significantly in the tree viewer. The comprehensibility in the tree viewer might become worse with every additional execution. In case one wants to clear the view from all monitors, one can use the *remove all input* action or simply delete a selected element through the context menu accessible by right-clicking. This way the developer always can restore the view's clarity.

Another advantage is that the output is easier to use in combination with other tools of ELK such as the layout time view or the layout graph view, since there is no need to switch between the development and the target platform anymore. This may come in handy, especially in scenarios where the developer has little screen space available or has to observe and compare output throughout multiple algorithm executions.

The new view still has a small disadvantage in comparison to the traditional way of textual debugging. On the console the output shows up immediately during the execution of the algorithm, whereas in the layout log view only the monitor's element within the tree viewer shows up immediately. Usually it has to be expanded twice before the actual algorithm's monitor shows up, because the diagram layout engine and recursive layout engine have their own monitors. Only then a monitor can be selected and its output displayed. The *expand all tree viewer elements* action reduces the clicks needed to find an element in the in tree structure. Either way some clicks are required in order to inspect some output which is not the case in the console. Still, I consider this disadvantage as outweighed by the advantages provided by the view.

6.3 Viewing Graphs

The advantages of the new layout time view in comparison to the old one are rather obvious. From a user perspective the functionality of the view has not changed. The final graph layout is still immediately shown after the algorithm's execution and the same applies to loaded graphs. By logging the final layout automatically it is assured that the developer can inspect it without using the progress monitors, just like before.

The ability to log graphs individually makes it possible to log their intermediate states during execution. For example, this might be useful when looking at the different iterations of the force algorithm. One could use the number of iterations to tag the graphs, so the state after every iteration of the algorithm could be picked and examined individually. An example can be seen in Figure 6.1. ELK's force algorithm uses a custom graph data structure which cannot be logged with a monitor. In order to be able to log its intermediate steps an ELK graph is needed. Thus I put the original ELK graph in scope of the algorithm's iteration as well as the graph transformer used to create the algorithm's data structure. After every iteration the change is applied through the graph transformer to the ELK graph, which is then logged along with the number of iterations already done. This is not an optimal usage within the code, but it works for presentational purposes. Since the force approach does not feature

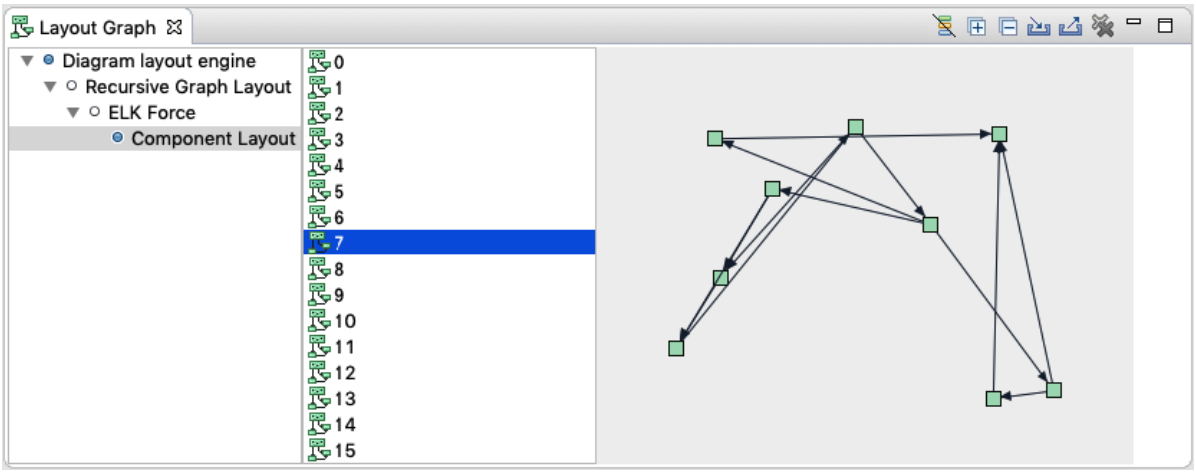


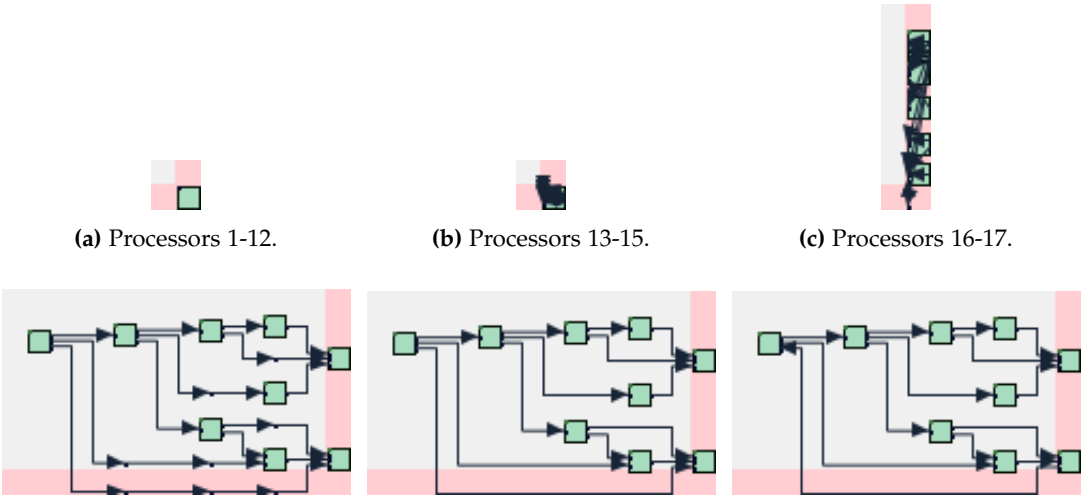
Figure 6.1. Using the layout graph view on ELK's force algorithm.

more than a single layout phase, all states are captured in one monitor. Thus, every graph is selectable in the list view and a specific iteration can be chosen to view its result. This way it is easy to observe how changes to layout options might impact the graph's layout not only at the end but also during the execution. For example the force algorithm introduced by Fruchterman and Reingold [FR91] uses a value called *temperature* which cools off with each iteration. It is used to scale down the force attracting and repulsing nodes during the execution. By increasing the initial temperature one could observe that the forces between the elements seem to be stronger than before.

I tried to do the same to ELK's layered algorithm using the already existing output which is made after every layout process. Unfortunately the JSON output contained properties unknown to the parser, so I modified the code such that no properties are attached to the graph anymore. This way the monitor could be passed to the algorithm's debug utility class which logs the JSON graph. By doing so, each processors's result is logged within the algorithm's root monitor. This is just a workaround since doing it properly would require changes to every single processor instead of just one to the debug utility. The observable changes to the graph are shown in Figure 6.2. The only thing one can see for the first twelve layout processes is all nodes being placed at the same coordinate. That is because the algorithm only does computations which aid the actual coordinate calculations later and are not applicable to the graph yet. In the 13th process edges are added, but they are crowded over the nodes. The interesting processes are the last five. The first observable change is the height calculation of the graph where one can see that the nodes finally having vertical coordinates assigned. Afterwards the edge routing takes place. One can see the nodes being placed in their final position and the edges being routed, while the graph still contains its dummy nodes. These are removed and subsequently the previously reversed edge is restored.

For now I do not see a particular use in visualizing intermediate steps for the layered algorithm on the laid out graph but for the force algorithm or others. So it depends on the algorithm whether the view's ability to display intermediate graphs is useful.

There is another use case developers might benefit from. The view obviously allows any graph to be passed and in case of the layered algorithm this could be used to visualize, for example, *dependency graphs*. These are used within the orthogonal edge routing process for determining how edges between two adjacent layers are routed. I can imagine that this may help when debugging the edge routing with a high density of edges between layers. The layered algorithm's debug utility class already supports



(a) Processors 1-12. **(b)** Processors 13-15. **(c)** Processors 16-17.

(d) Processor 18. **(e)** Processor 19. **(f)** Processor 20.

Figure 6.2. Observable changes in the layered algorithm.

JSON and *dot* output for dependency graphs. These could be logged with a monitor and examined in the layout graph view. So this view comes with the possibility to view auxiliary graphs used within an algorithm. This might be useful in highly complex ones such as ELK's layered algorithm. These graphs might not be laid out, therefore I experimentally implemented a check for an optional *algorithm* property, attached to the logged JSON graph, to lay those out if needed. So by adding this property to the root node one can inspect the graph even though it was never laid out before. It is not advised to use the algorithm one is developing to lay out an auxiliary graph because an algorithm in its development phase should not be used to help debug any other algorithm.

6.4 Incorporating Logging in Algorithms

Logging output and graphs requires a progress monitor to be in scope. This is currently not always the case in ELK's algorithms, which made it necessary to make the aforementioned changes the algorithms' code in the first place. So in future if one wants to utilize the full capabilities of the new debug features, one always has to pass a monitor to a class holding it in a variable or to a method directly in order to log anything. Consequentially, these feature cannot be used within ELK's algorithms right away. Developers should be urged to built their algorithm in a manner that allows for the usage of these debug features. This might make development slightly more tedious but eases the debugging process.

Chapter 7

Conclusion

This chapter will summarize the problems and corresponding solutions which were made subject of discussion in this thesis. In the end some future work is presented.

7.1 Summary

In this thesis I identified inconveniences in the process of developing layout algorithms with ELK.

This includes the development setup. I identified some issues in regards of testing and developing algorithms with certain setups. ELK's development tools are not usable without an external viewer, which is not included in the documentation's setups. While this is not a problem for developers with their own application having an external viewer, it might be one for people who just want to develop layout algorithms. I proposed to include an option in the setup to install the KlighD view along ELK's components. This way developers can choose to install the KlighD view if they have no external viewer at hand.

Setting up layout algorithm projects for ELK can be a tedious task. Numerous steps need to be made by hand and users have to handle things around Eclipse plug-in system. These are rather error-prone and take some time, thus I attempted to improve this by introducing a project wizard. It is supposed to make the whole process simpler and faster and developing layout algorithms for ELK more accessible. My implementation reduces the time needed immensely and immediately creates a runnable algorithm that serves as an example for new developers. Users, who will profit from the wizard's addition the most, are definitely student taking the lecture *Automatic Graph Drawing*.

While taking the lecture myself I experienced two more major inconveniences. One of those was the tedious switching between the development and the target platform while debugging an algorithm with console output. It also printed output from other code. For that I introduced the *layout log* view which allows the developer to inspect output logged through a progress monitor. The view structures the logs by their monitors and thus provides clarity which might be lost within the console. Logging only does happen when the preference newly added to ELK is enabled. Thus, it can be disabled, when one wants to measure an algorithm's performance. The preference is disabled by default, so no logging to the monitors is done when the algorithm is used in real-world applications.

This preference also accounts for logging graphs which is a functionality introduced in order to visualize intermediate and auxiliary graphs. The *layout graph* view was expanded to give developers a tool to view multiple graphs per execution. They now can log ELK graphs or strings declaring one in JSON or *dot* format. This way the behavior of an algorithm is evaluated easier and in a more graphical manner. However, this is not the only improvement to the view; the graph's drawing also was improved. Every element is now drawn regardless of its position. The area outside the graph's boundaries is highlighted in red to indicate an element's misplacement. The improved functionality of the layout graph view helps developers to spot mistakes in a graph's drawing at a glance and the possibility of viewing auxiliary graphs might help when debugging complex algorithms.

All in all the introduction of these new features to ELK might improve the development cycle of

layout algorithms. As these changes have been made with hindsight to my spent time in the lecture Automatic Graph Drawing, I think these will benefits students the most. However, I think they have potential to aid further development of ELK's algorithms as well.

7.2 Future Work

Some work still remains to further improve the tool support for layout algorithm development in ELK. To free the user from any interaction with Eclipse's plug-in system while creating layout algorithm projects, a way has to be found to make the wizard's generated plug-in a singleton automatically. The layout graph view still lacks the visualization of the *dot* format, thus this addition would round out the view's functionality. It could also allow users to load JSON and *dot* files via the load action. As for the layout time view, some functionalities could be added which are already available in the newly introduced views and would make the view more user-friendly. For example the expand and collapse actions could be introduced as well as the context menu allowing for single deletions of executions. Every view could also support a search function where one can enter a keyword and only monitors containing it show up. This might help, if the developers examines a specific part of an algorithm over multiple executions. Also, ELK has the build in layout option `debugMode` which more or less is obsolete now, because of new the check box preference for logging. ELK algorithms need some changes such as putting the monitor in scope when doing debug output and updating all console printing to log with the monitors. This is not mandatory for them to work, but for future modifications to the algorithms these changes may help with testing and debugging.

Bibliography

[BRS+07] Chris Bennett, Jody Ryall, Leo Spalteholz, and Amy Gooch. "The aesthetics of graph visualization". In: *Proceedings of the International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging (CAe'07)*. Banff, Alberta, Canada: Eurographics Association, 2007, pp. 57–64.

[DET+94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. "Algorithms for drawing graphs: an annotated bibliography". In: *Computational Geometry: Theory and Applications* 4 (June 1994), pp. 235–282.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. "Graph drawing by force-directed placement". In: *Software—Practice & Experience* 21.11 (1991), pp. 1129–1164. ISSN: 0038-0644. DOI: http://dx.doi.org/10.1002/spe.4380211102.

[Har98] David Harel. "On the aesthetics of diagrams". In: *LNCS, Mathematics of Program Construction* 1422/1998 (1998), pp. 1–5.

[HDM+13] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O'Brien. *SCCharts: Sequentially Constructive Statecharts for safety-critical applications*. Technical Report 1311. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2013.

[HFS11] Reinhard von Hanxleden, Hauke Fuhrmann, and Miro Spönemann. "KIELER—The KIEL Integrated Environment for Layout Eclipse Rich Client". In: *Proceedings of the Design, Automation and Test in Europe University Booth (DATE '11)*. Grenoble, France, Mar. 2011.

[Pur02] Helen C. Purchase. "Metrics for graph drawing aesthetics". In: *Journal of Visual Languages and Computing* 13.5 (2002), pp. 501–516.

[Rie10] Martin Rieß. "A graph editor for algorithm engineering". Bachelor Thesis. Kiel University, Department of Computer Science, Sept. 2010.

[Sch] Christoph Daniel Schulze. "Text in diagrams: challenges to and opportunities of automatic layout". Submitted. PhD dissertation. Faculty of Engineering, Kiel University.

[Sch15] Alan Schelten. "On the greedy reduction of edge crossings". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/alan-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Mar. 2015.

[STT81] K. Sugiyama, S. Tagawa, and M. Toda. "Methods for visual understanding of hierarchical system structures". In: *IEEE Transactions on Systems, Man, and Cybernetics* 11.2 (Feb. 1981), pp. 109–125. ISSN: 0018-9472. DOI: 10.1109/TSMC.1981.4308636.