Christian-Albrechts-Universität zu Kiel

Bachelor Project

# A Graph Editor for Algorithm Engineering

Martin Rieß

September 30, 2010

Department of Computer Science
Real-Time and Embedded Systems Group

Prof. Dr. Reinhard von Hanxleden

Advised by:
Dipl. Inf. Miro Spönemann

# Eidesstattliche Erklärung

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

Kiel,

_____

**Abstract**

Graph structures are used in many areas of computer science to represent different kinds of information. However the real importance of graphs lies in the algorithms that operate on them. One particular kind of graph-based algorithms are layout algorithms for the automatic layout of graph-based diagrams, and with the continuing rise in popularity of Model-Driven Software Development (MDSD) such pragmatics become even more important.

This thesis explains the concepts and implementation of a graph editor focused on the development of graph-based algorithms, in particular layout algorithms. In addition to the graph editor the development of a sophisticated graph analysis framework is explained.

All parts of this project contribute to the Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) project.

# Contents

# List of Figures

# Listings

# Abbreviations

**CAU** Christian-Albrechts-Universität zu Kiel

**EMF** Eclipse Modeling Framework

**GEF** Graphical Editing Framework

**GMF** Graphical Modeling Framework

**GML** Graph Modelling Language

**GPS** Global Positioning System

**GrAna** Graph Analysis

**GXL** Graph Exchange Language

**HTML** HyperText makeup Language

**IDE** Integrated Development Environment

**KIELER** Kiel Integrated Environment for Layout Eclipse RichClient

**KIML** KIELER Infrastructure for Meta Layout

**KSBasE** KIELER Structure Based Editing

**M2M** model-to-model

**MDSD** Model-Driven Software Development

**MVC** Model-View-Controller

**OCL** Object Constraint Language

**OMG** Object Management Group

**OSGi** Open Services Gateway initiative

**QVT** Query/View/Transformation

**QVTO** Operational QVT

**RCP** Rich Client Platform

**UI** User Interface

**UML** Unified Modeling Language

**SWT** Standard Widget Toolkit

**TGF** Trivial Graph Format

**XGMML** eXtensible Graph Markup and Modeling Language

**XML** Extensible Markup Language

# 1 Introduction

Graphs are a powerful mathematical concept used in many disciplines, especially in computer science. Computer networks, artificial neural networks, electronic circuits, bond graphs, mind maps, logistic networks, social networks and chemical formulas are all examples for specific problems that are often represented using graphs. Some of these problems were present and represented in diagrams that resemble those of graphs even before a formal definition of a graph was given. This indicates that graphs are an intuitive way to model problems that can be divided into entities and relations between these entities.
However, when using computers as tools to work with graphs, finding a graph representation of a problem is often only the necessary first step. Once such a graph is found, the user is commonly interested in a graphical representation of the graph structure or in additional information about the problem that can be derived from the graph structure. The former often involves an interactive process where the user modifies the graph while using the graphical representation to keep track of the whole structure. In this case a proper drawing of the graph, may it be a manual or automated drawing, is often crucial for a smooth work flow. An example for the latter are Global Positioning System (GPS) navigation devices that use graph structures to represent the street network. An algorithm such as *Dijkstra's algorithm* could supply the information that is required by the device to work properly. Often algorithms developed in computer science or mathematics can be reused for such use cases, but that is not always the case. Furthermore, when developing customized or entirely new algorithms, it is not always possible to verify the results automatically. That is especially true for graph drawing algorithms that try to satisfy specific aesthetic criteria.
The need for specialized development tools is inevitable.

## 1.1 Problem Statement

The main goal of this thesis is the development of a graph editor for the KIELER project, which in turn can be used to develop graph based algorithms, especially graph drawing algorithms. Besides the development of the graph editor itself this includes a sophisticated graph analysis mechanism.
To achieve this two overall goals a number of subtasks had to be completed.

1. Construct a graph editor that is compatible to the KIELER framework.
2. Provide structure-based editing commands for the graph editor.

3. Allow the import of graphs from common graph file formats.

4. Develop an easily extensible mechanism to analyze a given graph diagram and visualize the results of this analysis. This should also function as a constraint checker.

5. Implement a number of graph analysis algorithms as a basis using the mechanism mentioned above.

The tasks 1 to 3 belong to the graph editor, while the tasks 4 and 5 belong to the graph analysis. As this project is part of the KIELER framework it is developed as an Eclipse plug-in; more on KIELER and Eclipse in Chapter 3.

## 1.2 Outline

In Chapter 2 the different types of graphs and drawings that are commonly in use are introduced.
Chapter 3 gives a quick overview of the technologies used throughout the implementation. At first Eclipse itself is introduced, especially its plug-in and extension point mechanisms. On this basis some fundamental Eclipse technologies are presented, namely the Eclipse Modeling Framework (EMF), the Graphical Modeling Framework (GMF), and the transformation language *Xtend*. Afterwards the KIELER framework is introduced, in particular the KIELER Infrastructure for Meta Layout (KIML) and the KIELER Structure Based Editing (KSBasE). At last some file formats used to store graph structures are introduced.
The design and features of the graph editor, called GRAPHS, as well as it's implementation are presented in Chapter 4. Beginning with the editor itself the planned features are discussed, namely the random graph generation, the structure-based editing and the graph import.
The graph analysis mechanism, called KIML GrAna, is covered in Chapter 5. Firstly, the concepts and features of the mechanism are elucidated followed by a presentation of the planned interfaces, through which the user will communicate with the mechanism, programmatically and/or using the User Interface (UI). A number of example analyses are motivated and sketched next. Lastly a more complex extension to GrAna is described that allows the use of *Xtend* as a scripting language for analyses at runtime.
In Chapter 6 some use cases for the editor and its tools are presented.
The thesis concludes in Chapter 7, and also possible future work is discussed.

# 2 Graphs and Drawings

For a better understanding of graph related topics that arise throughout this thesis the commonly accepted types of graphs and graph drawings will be introduced shortly in this chapter. The mathematical definitions of graphs, graph drawings and other graph-related concepts are not relevant for this thesis and will be omitted.

## 2.1 Generalizations and Extensions of Graphs

A basic graph can consist of any number of *nodes*. Any two different nodes can be connected by a maximum of one *edge*. This is the simplest form of a graph, also just called *undirected graph*, and often it is not sufficient to represent specific problems. To model more complex problems, a number of graph generalizations and extensions have been developed:

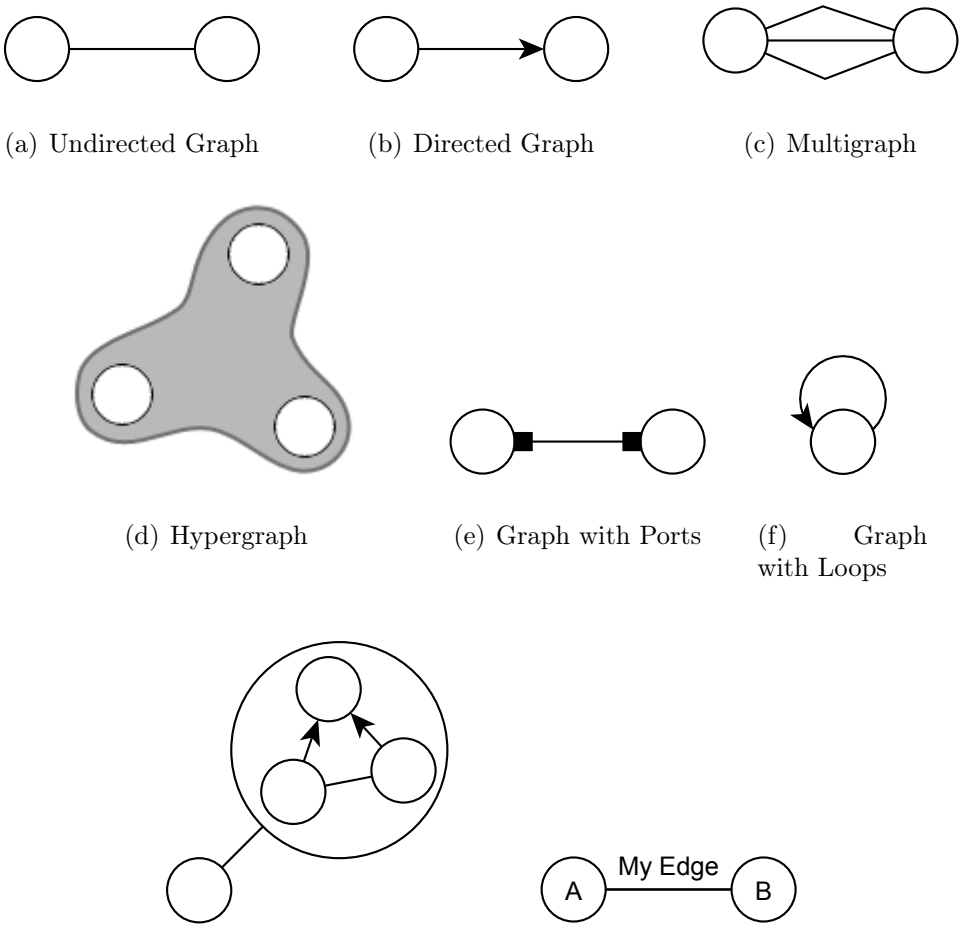
**Directed Graphs:** In a directed graph two nodes are not just connected by an edge. Instead any edge has a source and a target node, and is generally called a *directed edge* as opposed to *undirected edges* in undirected graphs. In particular it is possible that two nodes are connected by two opposite directed edges.

**Mixed Graphs:** A mixed graph can contain directed and undirected edges.

**Multigraphs:** A multigraph has no restriction on the number of edges, directed or undirected, that connect two different nodes. Several edges that connect the same two nodes are also often called *parallel edges* or *multiedges*.

**Hypergraphs:** In a hypergraph an edge can connect any number of nodes, even just one, and is typically called a *hyperedge*. Often the number of nodes in a hyperedge is fixed for a specific problem. A hypergraph where all hyperedges contain k nodes is usually called a k-*uniform-hypergraph*. Because the 2-uniform-hypergraph is a basic undirected graph the hypergraph can be seen as a generalization of the basic undirected graph.
When a hyperedge specifies a number of source and a number of target nodes, it is called a *directed hyperedge*, which again is a generalization of the basic directed edge.

**Graphs with Ports:** In a graph with ports every node has a number of ports attached to it. An edge in a graph with ports cannot only connect nodes, but also




(a) Undirected Graph (b) Directed Graph (c) Multigraph

(d) Hypergraph (e) Graph with Ports (f) Graph with Loops

(g) Graph with Hierarchy (h) Labeled Graph

Figure 2.1: Examples for different types of graphs

ports or nodes and ports. Depending on the problem edges may be restricted to only connect ports with ports and nodes with nodes.

**Graphs with Loops:** In a graph with loops an edge is allowed to connect a node with itself and is normally called a loop when it does so.

**Graphs with Hierarchies:** In a graph with hierarchies every node can contain a graph, usually referred to as a *nested graph*, on its own. Whether edges between nodes of different nested graphs are permitted depends on the problem.

**Labeled Graphs:** A *labeled graph* can have labels for nodes, edges and ports.

Furthermore many of these graph types can be combined, e.g. hypergraphs with ports.

## 2.2 Drawings

A drawing of a graph is a graphical representation of an embedding of the graph into the plane. In other words it is the arrangement of nodes, edges, ports and labels in a graphical diagram, probably some graph editor or a blackboard.
It is also the the subject of *layout algorithms*.

### 2.2.1 Aesthetics

Typically a graph drawing cannot be rated right or wrong. Different use cases require different properties of the drawing. Nevertheless some aesthetic criteria are commonly accepted as "good" characteristics for a drawing [4][9][1]:

**Area:** Minimization of the total space the drawing requires. This can be defined in different ways, for example as the minimization of the bounding box or the smallest convex polygon that completely surrounds the drawing.

**Aspect Ratio:** Minimization of the drawings aspect ratio, which is defined as the ratio of the length of the longest side of the bounding box to the length of smallest side of the bounding box.

**Direction:** Maximization of the number of edges that are pointing in a single direction.

**Edge Length:** Minimization of the edge lengths and minimization of the variance of the length of different edges.

**Number of Bends:** Minimization of the number of bends along the edges.

**Number of Crossings:** Minimization of the number of edge and node crossings.

**Symmetry:** This aesthetic is special as it is hard to define precisely. Symmetries in the graph have to be emphasized in some way.

This list is by no means complete and could even be inappropriate for some use cases. Furthermore different criteria are often contradicting each other.

### 2.2.2 Methods of Graph Drawing

There are a host of different approaches for the problem of graph drawing, a few of which are introduced in the following:

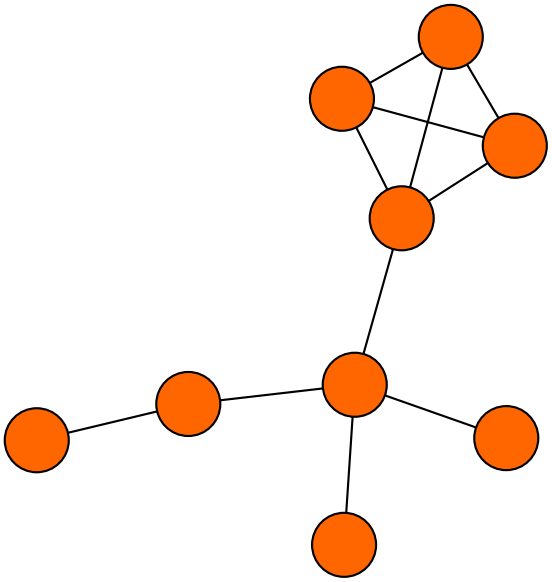
**Force-Based:** The *Force-Based* approaches identify the graph for which the drawing is to be found with a physical model and try to minimize the energy in that model. This approach is very likely to emphasize symmetries in the graph. However most variants draw all lines straight, which increases the chance of line crossings drastically. See Figure 2.2(a) for an example force-based drawing.

**Layered:** The *Layered* approaches focus on emphasizing the direction of the graph and thus require a directed graph. See Figure 2.2(b) for an example of a layered drawing.

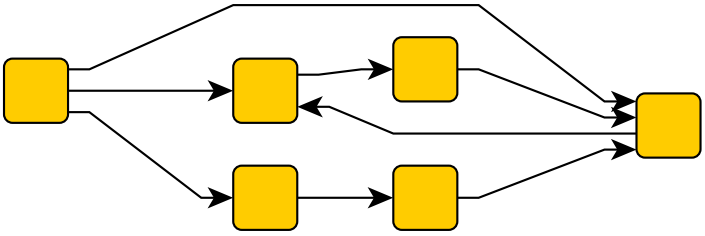
**Planarization-Based:** The *Planarization-Based* approaches try to minimize the amount of edge crossings by finding a planar embedding of the graph in the plane, i.e. a maximal subgraph for which a planar drawing exists. These kind of drawings are of interest in the design of *electronic circuits* and *class diagrams*. See Figure 2.2(c) for an example of a planarization-based drawing.

**Tree:** The *Tree* approaches usually require the graph to be a *forest*, i.e. contain no cycles, and show the common formation of a rooted tree (or forest). See Figure 2.2(d) for an example of a tree drawing.

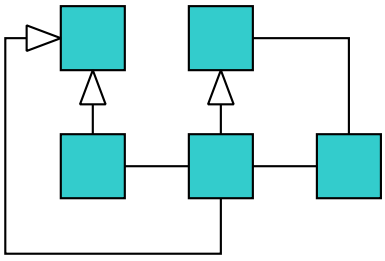
For further information on graph drawing the reader is advised to consult adequate literature[4][3].



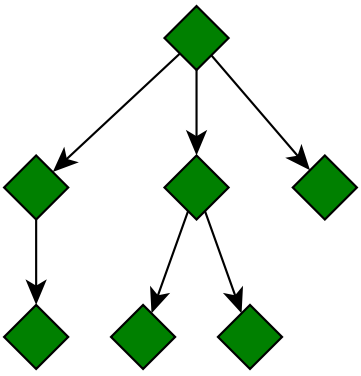
(a) Force-based drawing



(b) Layered drawing



(c) Planarization-based drawing with orthogonal edge routing



(d) Tree drawing

Figure 2.2: Examples for different types of drawings

# 3 Used Technologies

Before explaining any ideas, design decisions or implementations, the technologies used throughout this thesis have to be introduced. However, mainly the general concepts and functionalities will be elaborated.

## 3.1 Eclipse

On the surface *Eclipse* is an Integrated Development Environment (IDE) written in *Java* and initially also for Java development. However, in contrast to most other IDEs, it does not focus on a specific language anymore but can be customized to support a great number of different programming and modeling languages, e.g. Java, C++, Ruby, Unified Modeling Language (UML), Extensible Markup Language (XML) etc. This is summarized by the principle that Eclipse is "an IDE for anything, and nothing in particular"[8].

### 3.1.1 Plug-in System

Eclipse consists of a number of small components called *plug-ins* that work together to form the complete application. These plug-ins can practically contain any resources, however, Java binaries define the logic for the plug-in. They can be independent and self-contained or depend on other plug-ins to reuse their functionality. The plug-in management is done by the *platform runtime engine*, which is based on the Open Services Gateway initiative (OSGi) framework, a Java service platform that can install, start, stop, update and uninstall components in the form of so called bundles at runtime.
The information about a plug-in are contained in the `plugin.xml` and the `MANIFEST.MF` that can exist for every plug-in, the latter is mandatory.
The plug-in based design has several advantages like flexibility and re-usability. This allows the user for example to build a Rich Client Platform (RCP) application based on the Eclipse platform runtime engine while reusing many Eclipse features.
An example for two Eclipse plug-ins are the one that contains the *package explorer* and the one that provides the *Java editor*. For more information on the Eclipse plug-in system, one should visit the Eclipse website[1] or consult Eclipse literature[2].

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

### 3.1.2 Extension Points and Extensions

Another Eclipse concept related to plug-ins are *extension points* and *extensions*. A plug-in can define any number of extension points through the `plugin.xml` by referencing an extension point schema file, which is an XML file as well. By providing extension points, a plugin can be extended by other plug-ins in a controlled way.
A plug-in extending another one has to define an *extension* in the `plugin.xml` in a similar manner to the definition of the extension point.
An example for an extension point is the one that can be extended to add additional wizards.

### 3.1.3 The Workbench, Editors and Views

When opening an Eclipse instance, the user is confronted with the Eclipse workbench as shown in Figure 3.1. The workbench can be seen as the *main window* of the application and holds the various parts that form the application like the toolbar, the statusbar and two kinds of workbench parts whose difference is not immediately apparent:

**Editors:** An editor is usually associated with a file type, e.g. *C++* files. If the user opens a file of that type, the corresponding editor is opened or the default editor if more then one editor is registered on that file type. This is normally also the only way to open an editor, as editors are always associated with a concrete file instance.
Once a file is opened in an editor, this editor usually contributes to the workbench toolbar instead of having a custom one. Furthermore an editor type can be opened on several files or even the same file multiple times at the same time.
The *Java editor* is a common example for an editor and can be seen in the center of Figure 3.1.

**Views:** A view usually provides additional information about the content of the active editor or an element inside the editor and normally allows some kind of interaction, e.g. provides a table where the user can configure elements in the active editor or provides a search function for elements in the active editor. In contrast to editors there is normally at most one instance of a view open at a time, as several instances of a view would most likely display the same information.
The *Outline* is an example for a view and can be seen in Figure 3.1 on the right side. It currently shows the outline of the Java document that is opened in the editor.

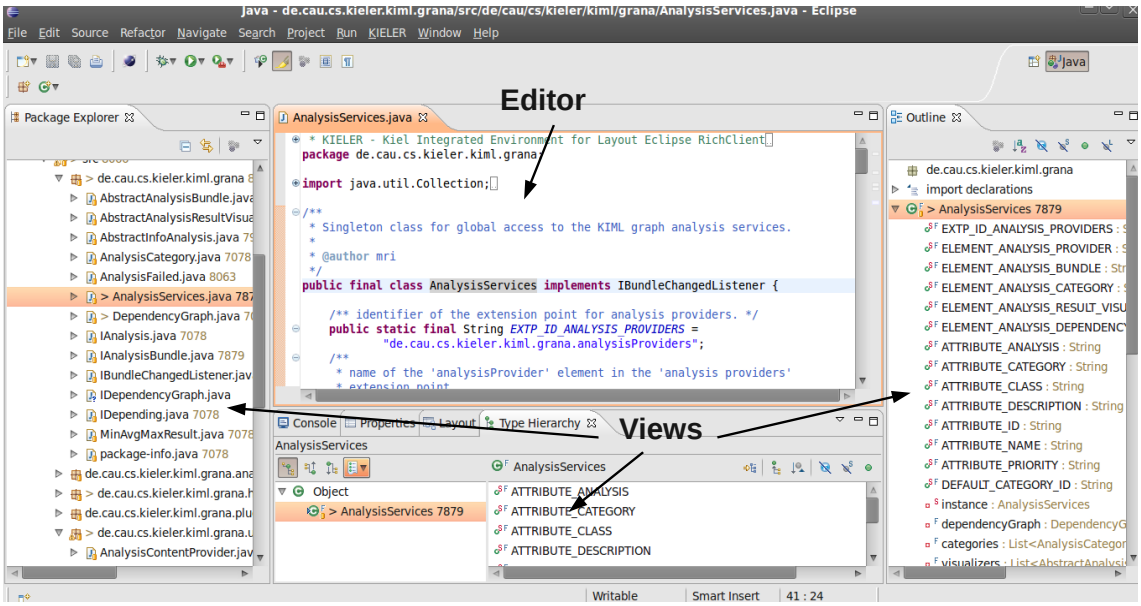



Figure 3.1: Eclipse workbench with editors and views

### 3.1.4 Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF) is the Eclipse implementation of the *Meta Object Facility* as defined by the Object Management Group (OMG)[2]. It is based on the concept of *metamodels* which define the abstract syntax of models.
As of now the EMF is *the* established quasi-standard modeling framework for Eclipse. This makes it reasonable to build plug-ins based on the EMF, to be compatible with as many Eclipse projects as possible.
The standard format for the EMF metamodels is *Ecore*, which itself is a model based on the Ecore metamodel. For the metamodel the user can choose from a number of tools. The most obvious are the manual creation using the tree or graphical Ecore editor. There are a several sources from which the model can be imported as well:

- Another Ecore model
- UML model
- XML schema
- Annotated Java

The EMF can transform all of these to an Ecore model.
With the use of an EMF *generator model* the user can configure and launch a *code generation* process that produces Java code to work programmatically with the

[2] http://www.omg.org/

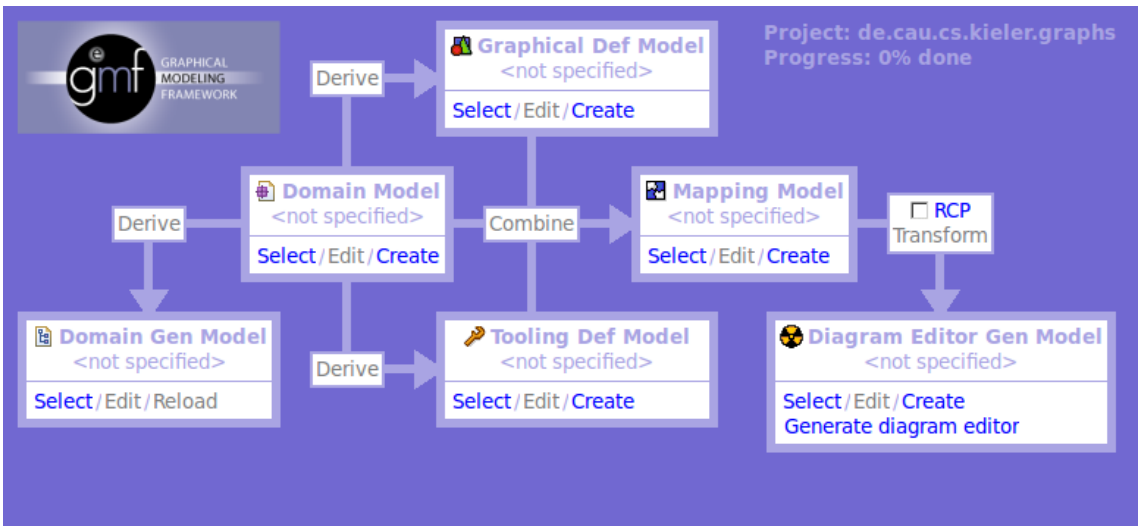



Figure 3.2: GMF Dashboard — a graphical representation of the GMF toolchain

model. This includes EMF features like for example serialization of the structure to XML and model validation.

For more detailed information about EMF, the reader should visit the website[3] or consult literature[10].

### 3.1.5 Graphical Modeling Framework

The Graphical Modeling Framework (GMF) utilizes both the EMF and the Graphical Editing Framework (GEF) to provide the functionality to generate highly customizable graphical editors for specified EMF metamodels.
The overall generation process consists of several generation steps which can be seen in the GMF *Dashboard* as shown in Figure 3.2. This dashboard is implemented as a view in Eclipse and serves as a user guide to generate the editor.

In the following the four major XML-based generation models for a GMF editor are described in more detail:

**Domain Model and Domain Generator Model:** The domain model for a GMF editor is defined by an Ecore file; see Section 3.1.4 on how to create such a file and a corresponding generator model.

**Graphical Definition Model:** The graphical definition model defines the look of the diagram. It can be roughly divided into two parts: the definition of the diagrams figures, i.e. shapes and connections, and the definition of the diagrams elements, i.e. nodes, edges, labels and compartments (node-container) and what figures are used for them. It does not depend on the information from the domain model.

[3] `http://www.eclipse.org/modeling/emf/`

**Tooling Definition Model:** The tooling definition model defines what categories and items are available on the palette, in context menus, in pop-up balloons etc. It does not depend on the information from the main model.

**Mapping Model:** The mapping model combines the domain-, graphical definition- and tooling definition model and defines how the elements defined in each model are connected. E.g. if the graphical definition model defines an element *Entity* using some rectangle figure and the tooling definition model defines a placing tool then the mapping model maps those elements on a class *Entity* defined in the domain model.

**Diagram Editor Generator Model:** The diagram editor generator model has to be generated from the mapping model and contains all the information from the other models in a form that is directly linked to generated code. E.g. class names are defined in there and the name of the diagram plugin.
The model can then be used to generate the actual diagram editor code; the generation process can be highly customized by providing modified *Xpand* template files.

For a deeper knowledge on Graph Modelling Language (GML) the reader should refer to the website[4] or adequate literature[6].

### 3.1.6 Xtend

*Xtend* is a functional language that is part of the Eclipse *Xpand* framework. It is statically typed and shares an expression language and type system with Xpand, which is a template language. It natively supports types such as `Integer`, `String` and collections such as `List[Integer]`, and can be extended by types defined in metamodels. The expression language supports basic arithmetics, member access (e.g. `element.doStuff()`), special collection operations (e.g. `list.select(i|i<42)`), operators (e.g. `!=`, `==`, `->`), numbers, strings and control flow statements (e.g. `if`, `let`, `create`).

One field of application for Xtend are M2M transformations[11]. An example M2M transformation can be seen in Listing 3.1. In lines 1 and 2 the metamodels involved in the transformation are included, which dynamically integrates the types that are defined in them into Xtends type system. In lines 4 to 6 the extension `doStuff` is defined, which returns `String` and takes an `Object` as argument. In line 5 Xtend escapes to Java using the `JAVA` keyword, i.e. calls a static Java method. In lines 8 to 12 a `create`-extension is defined, i.e. an extension that automatically creates a new object of the return type that can be accessed with `this`. When the extension is called a second time with the same arguments, the previously created return type instance is returned and the extension body is skipped. In line 9 a local variable is defined and set to an attribute of `type`.

[4] `http://www.eclipse.org/modeling/gmf/`

Listing 3.1: An example Xtend M2M transformation

```
1  import ecore1;
2  import ecore2;
3
4  Void doStuff(Object obj):
5     JAVA myUrl.myPackage.MyClass.myStaticFunction(java.lang.Object)
6  ;
7
8  create MyType1 transform(MyType2 type):
9     let name = type.name:
10    this.setId(name) ->
11    this.setName(name)
12 ;
```

## 3.2 Kiel Integrated Environment for Layout Eclipse RichClient

The Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) framework is an open-source research project about the enhancement of graphical model-based system design. It is developed by the Real-Time and Embedded Systems Group of the Department of Computer Science of the Christian-Albrechts-Universität zu Kiel (CAU).
The framework is organized as a set of Eclipse plug-ins, which integrate with the common Eclipse modeling projects, such as GMF and EMF.

### 3.2.1 KGraph

The *KGraph* is the central data structure in the KIELER project to represent graphs. It was created using the EMF framework and in consequence inherits the features of all EMF models, e.g. serializability to XML. Figure 3.3 shows the Ecore diagram that defines the KGraph data structure.

### 3.2.2 KIELER Infrastructure for Meta Layout

The KIELER Infrastructure for Meta Layout (KIML) is the subproject of KIELER that handles the automatic layout of graphical diagrams.
The approach which is followed in this project is that KIML manages the layout on an abstract level, while so-called layout providers supply algorithms that compute a concrete layout for a given diagram. The Eclipse extension point mechanism is used to allow plug-ins to connect such layout providers.

### 3.2.3 KIELER Structure Based Editing

The KIELER Structure Based Editing (KSBasE) is the subproject of KIELER that enables plug-ins to add structure-based editing commands to GMF-based editors. The term *structure-based* refers to the way how these editing is performed: by

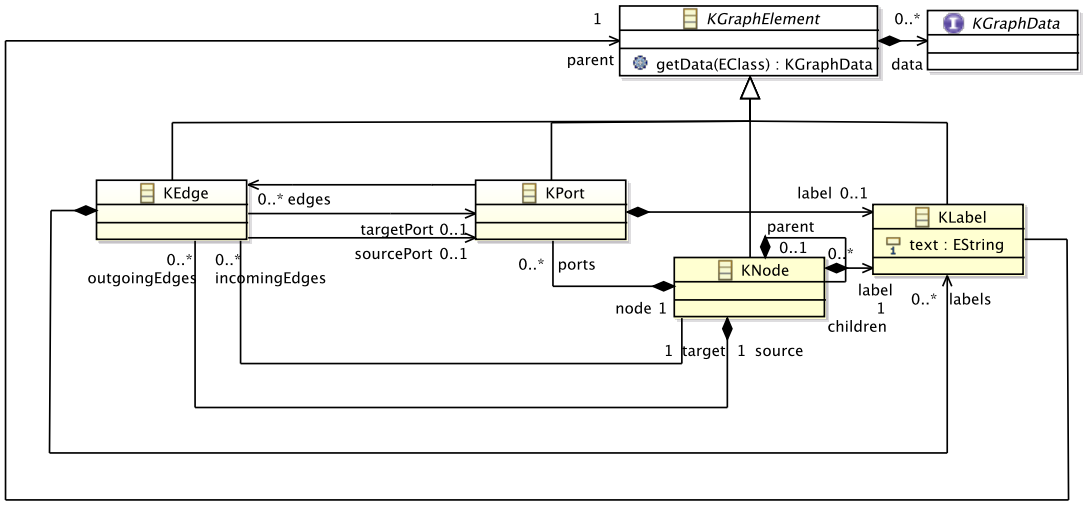



Figure 3.3: KGraph metamodel — the central data structure for graphs in KIELER

manipulating the underlying EMF metamodel instance. The editing commands are supplied using the Eclipse extension point mechanism.

## 3.3 File Formats

In this Section the GraphML file format is introduced. Some other graph file formats, which are of interest for potential future work on the topic of this thesis are mentioned as well.

### 3.3.1 GraphML

GraphML is a file format based on XML used to represent graphs. The format supports the following features:

- Directed and Undirected Edges
- Hyperedges
- Ports
- Hierarchies

It is also possible to attach custom data to the graph structure, but that is not relevant for this thesis.
An example GraphML file can be seen in Listing 3.2; a visualization of that graph is depicted in Figure 3.4.

Listing 3.2: GraphML example description

```
<?xml version="1.0" encoding="UTF-8"?>
<graphml xmlns="http://graphml.graphdrawing.org/xmlns"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://graphml.graphdrawing.org/xmlns
     http://graphml.graphdrawing.org/xmlns/1.0/graphml.xsd">
  <graph id="G" edgedefault="undirected">
    <node id="A">
        <port name="West"/>
    </node>
    <node id="B"/>
    <node id="C">
        <port name="East"/>
    </node>
    <node id="D"/>
    <node id="E"/>
    <edge id="E1" source="A" target="B"/>
    <edge id="E2" source="A" target="C" sourceport="East" targetport="West"/>
    <edge id="E3" directed="true" source="E" target="C"/>
    <hyperedge>
        <endpoint node="B"/>
        <endpoint node="E"/>
        <endpoint node="D"/>
    </hyperedge>
  </graph>
</graphml>
```

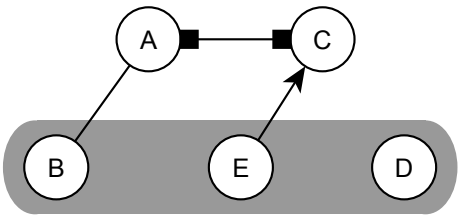



Figure 3.4: Possible visual representation of a graph

### 3.3.2 Other

In the following a few established graph file formats are listed.

- Graph Exchange Language (GXL), an XML-based file format, which was initially developed for graph exchange between software reengineering tools.
- eXtensible Graph Markup and Modeling Language (XGMML), an XML-based file format.
- Trivial Graph Format (TGF), a simple text-based file format.
- Graph Modelling Language (GML), a widely known text-based file format.
- dot, the file format used by Graphviz tools.

# 4 Graph Editor

In this chapter the design and implementation of the graph editor will be explained. The name of the graph editor is GRAPHS, but it will be referred to with the term *graph editor* or just *editor* as well.

Furthermore, as the editor is mainly generated from several generator models, the following will focus on decisions and customizations that had to be made rather than the concrete implementation of the editor.

The first three problems listed in Section 1.1 are handled in this chapter, which are:

1. Construct a graph editor that is compatible to the KIELER framework.
2. Provide structure-based editing commands for the graph editor.
3. Allow the import of graphs from common graph file formats.

## 4.1 Concepts and Features

The main guideline for the design of the graph editor is summarized by the title of this thesis: **A Graph Editor for Algorithm Engineering**.

In particular this excludes any features that are just noise for the process of developing graph-based algorithms. E.g., there are no semantic restrictions on what can be modeled. Everything the syntax allows is valid in terms of the graph editor. Yet it should be powerful enough to model all kinds of graphs that are presented in Section 2.1, i.e. the following elements:

- Nodes
- Directed Edges
- Undirected Edges
- Hyperedges
- Ports

Furthermore the features of the editor should be focused on building test-cases for the algorithms in question.

Often a random graph is sufficient for the purpose of testing the functionality of an algorithm and may be a good start for constructing a specific test case. Hence the graph editor should be able to generate random graphs.

Another important method to retrieve test-case graphs is the usage of existing graph libraries. The graphs in these libraries are given in a number of file formats; the graph editor should be able to import from such established graph file formats.

As the editor is part of KIELER, it is reasonable to accomplish as much compatibility to other KIELER concepts and projects as possible. One of these concepts is the *KGraph* (see Section 3.2.1). Since the editor will be built upon a metamodel for the representation of graphs, this should be the KGraph.

## 4.2 The GMF-Based Diagram Editor

The diagram editor has been constructed using the GMF (see Section 3.1.5). This involved the creation, customization and generation of several definition and generator models; the general workflow can be seen in Figure 3.2.

### 4.2.1 KGraph-based EMF Model

As depicted in Figure 3.2 the first step in the creation workflow is the selection of a *domain model*. In this case it should be the KGraph metamodel. In fact a metamodel is used that derives from the KGraph, the GRAPHS metamodel (see Figure 4.1), because a number of problems arose when mapping the graphical representation for the model to the KGraph:

**Labels:** Any labels for a KNode, KEdge or KPort are represented by the KLabel. As of now GMF is unable to dereference such labels and map them on their graphical counterparts. They have to be stored as `attributes` on the Ecore classes, i.e. on the nodes, edges and ports. In Figure 4.1 the EString typed attributes on the Node, Edge and Port realize this necessary modification by adding static labels.

**Hyperedges:** The usual representation of hyperedges[5][7] is unsuited for a framework that visualizes nodes and edges by shapes and connections between shapes.
To solve this problem a different representation for hyperedges has been chosen: *hypernodes*. A hypernode is a special node without labels, ports and hierarchy. Any node connected to a hypernode is part of one hyperedge and any number of connected hypernodes form a single hyperedge. This also allows for an easy modeling of directed hyperedges. As both hypernodes and normal nodes are represented by the same Ecore class, the KNode, the GMF is unable to differentiate between the two types of nodes. To solve this problem the boolean attribute isHypernode is added to the derived class Node to indicate whether a given node is a hypernode or not (see Figure 4.1).

**Node-Port Edges:** The graph editor permits edges between nodes and nodes, nodes and ports, and ports and ports. In the KGraph the difference between all these

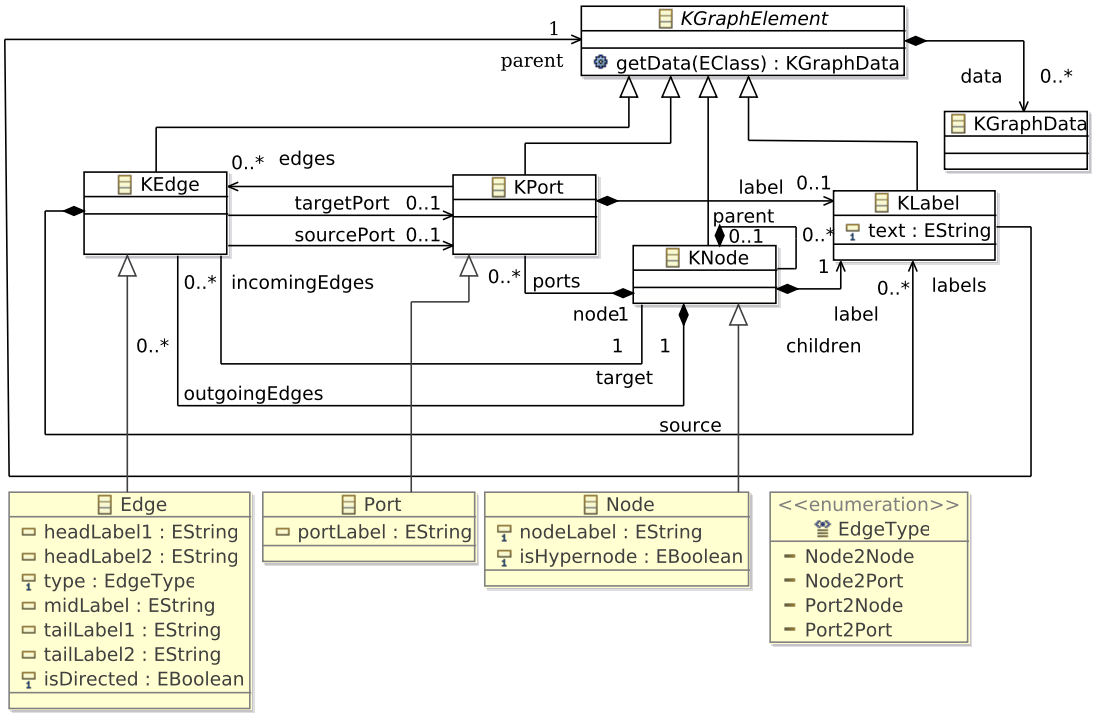



Figure 4.1: GRAPHS metamodel — the domain model for the graph editor

different types of how two nodes can be connected is whether the attributes sourcePort and/or targetPort of an edge are set. For example an edge with sourcePort and targetPort set connects two nodes by using the corresponding ports, while an edge with only targetPort set connects two nodes by using a port on the target node and directly connecting the source node.
The GMF is unable to differentiate between these kinds of edges. To solve this problem an enumeration, which represents the four kinds of possible edge types, is added to the the GRAPHS metamodel, and an attribute, type, using that enumeration is added to the Edge class (see Figure 4.1). So an edge that has only targetPort set would have the attribute type set to Node2Port.

All of the changes made in the derived GRAPHS metamodel contain only information that can already be expressed in the KGraph metamodel, which allows for a trivial transformation between both models.

### 4.2.2 GMF Models

As described in Section 3.1.5 to create a diagram editor with GMF a number of models are required that partly build on one another:

### Graphical Definition Model

This model defines the look of the graph editors elements on the canvas. In Figure 4.2 the model can be seen in the corresponding GMF tree editor, with the figures and diagram elements labeled accordingly. The resulting graphics that are defined by the figures and diagram elements can be seen in Figure 4.3. These diagram elements resemble the depictions of graph types as introduced in Section 2.1 and shown in Figure 2.1.

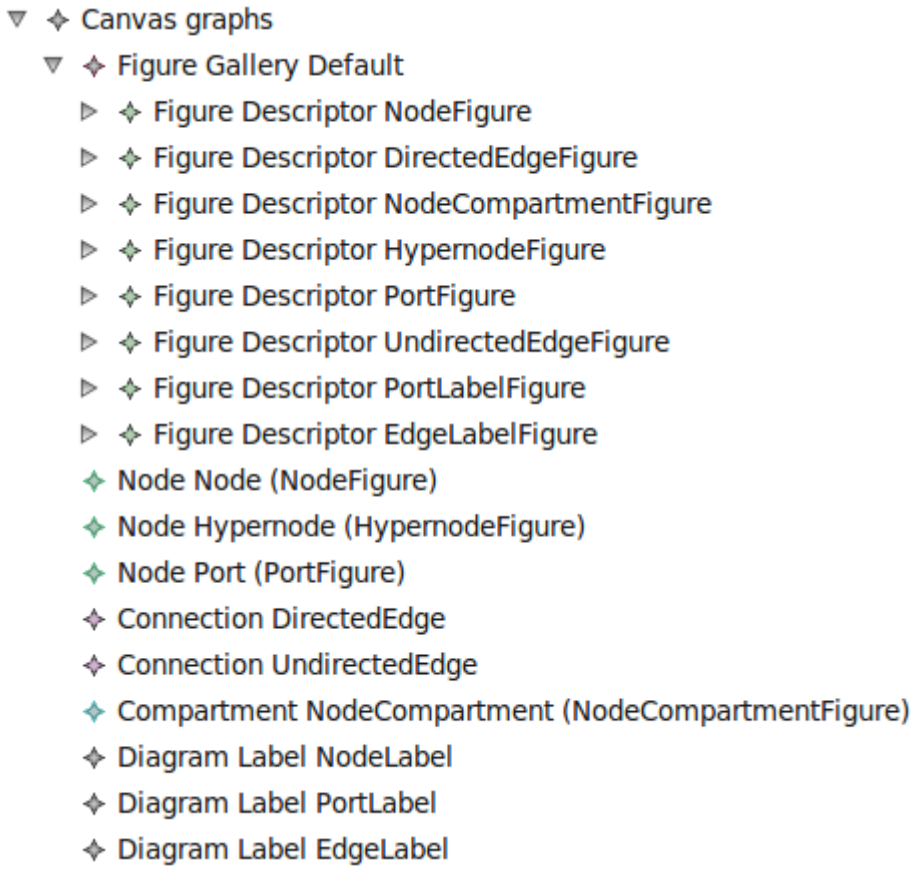
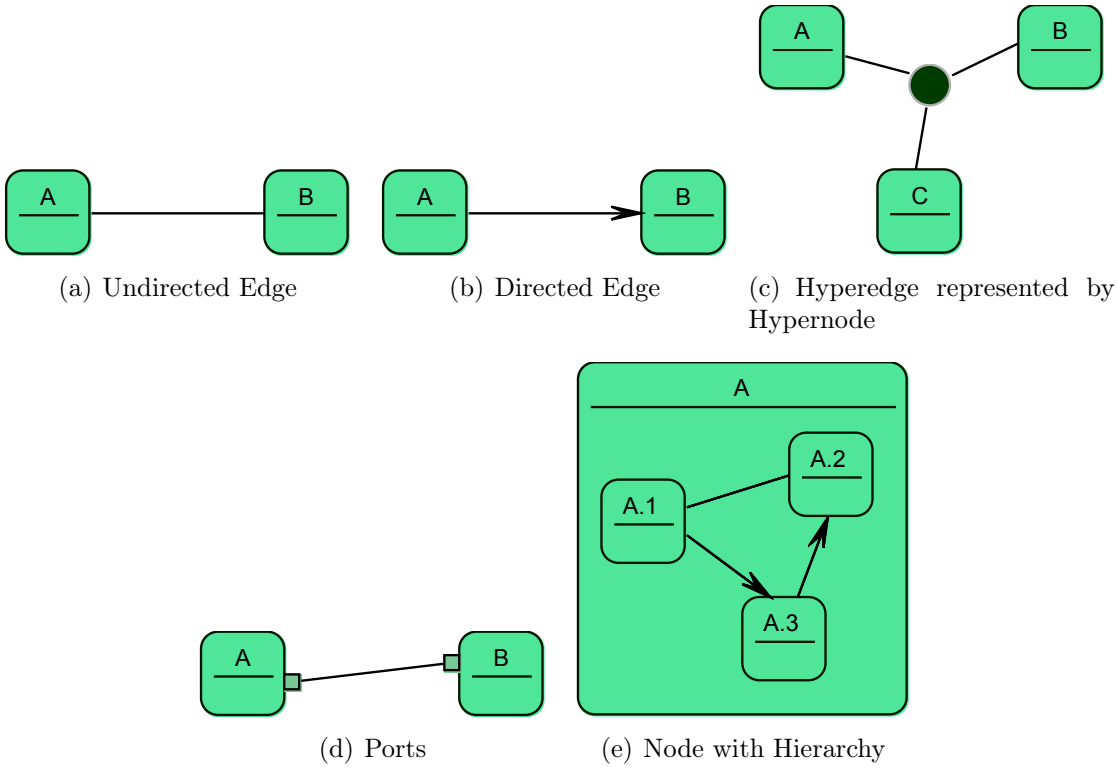



Figure 4.2: GRAPHS Graphical Definition Model in the tree editor

### Tooling Definition Model

For the graph editor this model defines the contents of the palette, as the other targets for tool placement are not used. In Figure 4.4(a) the tooling definition model can be seen in the tree editor. Figure 4.4(b) shows the resulting palette as it appears in the graph editor. Note that most information that makes up the palette, such as the icons and what diagram element is linked to an entry are implicitly defined by the EMF generator model or by the mapping model, see next section.



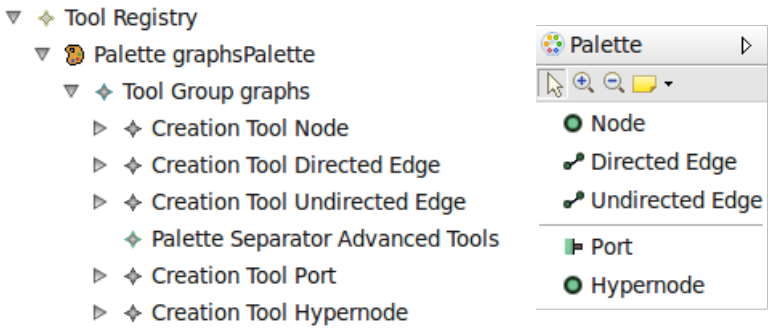

(a) Undirected Edge (b) Directed Edge (c) Hyperedge represented by Hypernode

(d) Ports (e) Node with Hierarchy

Figure 4.3: GRAPHS figures for graph elements

### Mapping Model

The mapping model connects the domain model with the tooling definition model and the graphical definition model. In Figure 4.5 the model is shown in the corresponding tree viewer. The structure of the model resembles the structure of a graph as defined in the domain model. The top level element of a GRAPHS model is a Node and is mapped to the canvas. This can be done in the properties of the CanvasMapping element as shown in Figure 4.6 at the Element attribute. Every node that is placed on the canvas has to match a Top Node Reference and a nested Node Mapping. In the graph editor there are two elements that are directly placed on the canvas: nodes and hypernodes. The mapping for top level nodes starts at (a) in Figure 4.5. As the difference between a node and a hypernode is an attribute in the domain model the mapping contains an Object Constraint Language (OCL) constraint as seen at (b) which separates the node types. The label for the node is mapped at (c). As nodes in the graph editor can contain nodes as well, a Child Reference recursively maps the children of the node to the previously defined Node Mapping at (d) and to the yet to be described Node Mapping for hypernodes at (f). At (e) ports are mapped to their diagram counterpart.
The mapping for hypernodes is done at (g) and is similar to the one of nodes with the exception that hypernodes have no label, no ports and no children.




(a) GRAPHS Tooling Definition Model in the tree editor (b) GRAPHS palette

Figure 4.4: GRAPHS Tooling Definition Model and palette

The last type of mappings are the eight Link Mappings, which result from the combination of the properties directed/undirected with node-to-node, port-to-port, node-to-port and port-to-node edges. This distinction of cases is handled with OCL constraints as depicted at (h). As a edge in the graph editor has five possible labels also five label mappings have to be defined, as done at (j). At (i) the domain elements attribute to differentiate between the edge types (see above) is initialized.

#### Diagram Editor Generator Model

This model is generated from the mapping model and combines all informations from the previous models. To preserve the capability to make changes in the other models and then generating this model again without loosing any customizations, no changes are directly done in the model.
Instead the generation process from the mapping model is customized by a Operational QVT (QVTO) file shown in Listing 4.1. The most important customizations are the enabling of *Template Customizations* for the next Section in lines 8 and 9, the setting of the file endings for the domain file to *graph* in line 12 and the notation file to *graphdiag* in line 13 and the correction of the label positions in line 25.

### 4.2.3 Template Customizations

By customizing the Xpand generation templates the generation of Java code can be directly influenced. This is used for a few things on the graph editor:

**Splines:** By default GEF does not support spline figures for edges. A temporary solution for this problem exists in the KIELER project, which requires the alteration of the base class for the edges.

**Marker Interfaces:** The diagram editor generator model defines a number of classes for edges depending on the number of Link and Node Mappings. To have an easy way to identify a given `EditPart` (diagram element) as a node, edge or

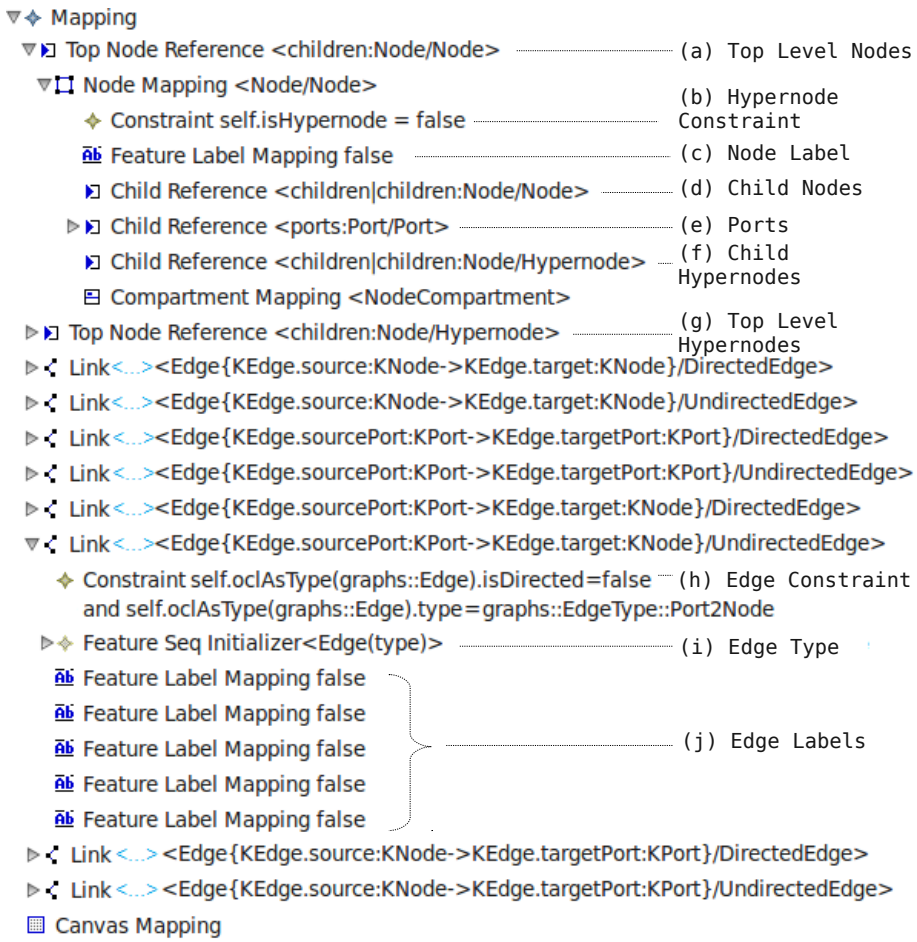



Figure 4.5: GRAPHS Mapping model in the tree editor

label a number of marker interfaces have been integrated into the generation process.

**Copy and Paste:** A custom implementation is given for *Copy and Paste* commands (see Section 4.4). However before these become active the standard versions have to be disabled; this is done by altering the generation of the `plugin.xml`.

Since the customizations are trivial but very technical, any more details will be omitted.

## 4.3 Random Graph Generation

The *Random Graph Generator* creates random graphs based on a number of options.

**Number of nodes:** The graph will contain the given number of nodes.

**Minimal number of outgoing edges per node:** Every node will have at least this number of outgoing edges.

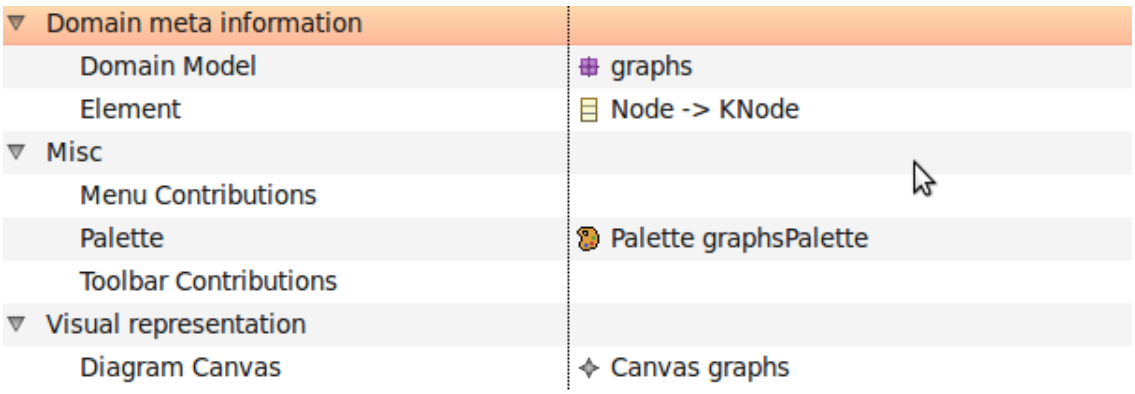



Figure 4.6: Property View showing the properties of the CanvasMapping

**Maximal number of outgoing edges per node:** Every node will have no more than this number of outgoing edges. The chances for numbers of nodes between the minimal and maximal value are evenly distributed.

**Probability of introducing new hierarchy levels:** The probability that a node will contain a subgraph.

**Probability of creating hypernodes:** The probability that a node will be a hypernode. This and the previous option exclude each other and can both together not exceed 100 percent.

**Undirected or directed edges:** All edges are undirected or directed. The generator will never generate mixed graphs.

**Ports:** All edges are connected using ports or not.

The Random Generator can be invoked using the *Random Graph Wizard* as depicted in Figure 4.7. After finishing the wizard a graph file will be created in the specified destination containing the generated graph.

## 4.4 Structure-Based Editing

So far the only way to add nodes and edges to the canvas is by dragging them from the palette or by using the pop-up balloons. Since this is a time consuming task, the KSBasE framework is used to add more specialized editing commands to the editor.

**Add Successor:** Adds a new node to the canvas and connects the selected node with a directed edge to the new node.

**Add Predecessor:** Adds a new node to the canvas and connects it with a directed edge to the selected node.

Listing 4.1: Diagram editor generator model customization file

```
modeltype GMFGEN uses gmfgen('http://www.eclipse.org/gmf/2009/GenModel');

transformation GraphsCustomization(inout gmfgen:GMFGEN);

main() {
  var model := gmfgen.rootObjects()![GenEditorGenerator];

  model.dynamicTemplates :=true;
  model.templateDirectory := "de.cau.cs.kieler.graphs/gmf-templates";
  model.sameFileForDiagramAndModel := false;

  model.domainFileExtension := "graph";
  model.diagramFileExtension := "graphdiag";

  // some general plugin settings
  model.plugin.requiredPlugins += "de.cau.cs.kieler.core.ui";
  model.plugin.version := "0.1.0.qualifier";
  model.plugin.provider := "Christian-Albrechts-Universitaet zu Kiel";
  model.plugin.name := "Graphs Editor";

  // use our own category for the "new" wizard
  model.diagram.creationWizardCategoryID := "de.cau.cs.kieler";

  // correct the edge label placement
  model.diagram.allSubobjectsOfType(GenLink)[GenLink] ->map edgeSettings();
}

mapping inout GenLink::edgeSettings(){
  self.allSubobjectsOfType(GenLinkLabel)[GenLinkLabel] ->map edgeLabelSettings();
}

mapping inout GenLinkLabel::edgeLabelSettings(){
  self.alignment := switch {
    // SOURCE and TARGET seem to be confused but they are not -> gmf problem
    case (self.editPartClassName.startsWith('EdgeHead')) LinkLabelAlignment::SOURCE;
    case (self.editPartClassName.startsWith('EdgeMid')) LinkLabelAlignment::MIDDLE;
    case (self.editPartClassName.startsWith('EdgeTail')) LinkLabelAlignment::TARGET;
  };
}
```


**Add Neighbor:** Adds a new node to the canvas and connects it with an undirected edge to the selected node.


**Connect Directed:** Connects two selected nodes with a directed edge. The order in which the nodes were selected determines which node is the source and which the target node.


**Connect Undirected:** Connects two selected nodes with an undirected edge.


**Connect Hyperedge:** Connects any number of selected nodes with a hyperedge by adding a new hypernode to the canvas and connecting it with undirected edges to all selected nodes.


**Flip Edge:** Swaps the source and target node of a selected edge.

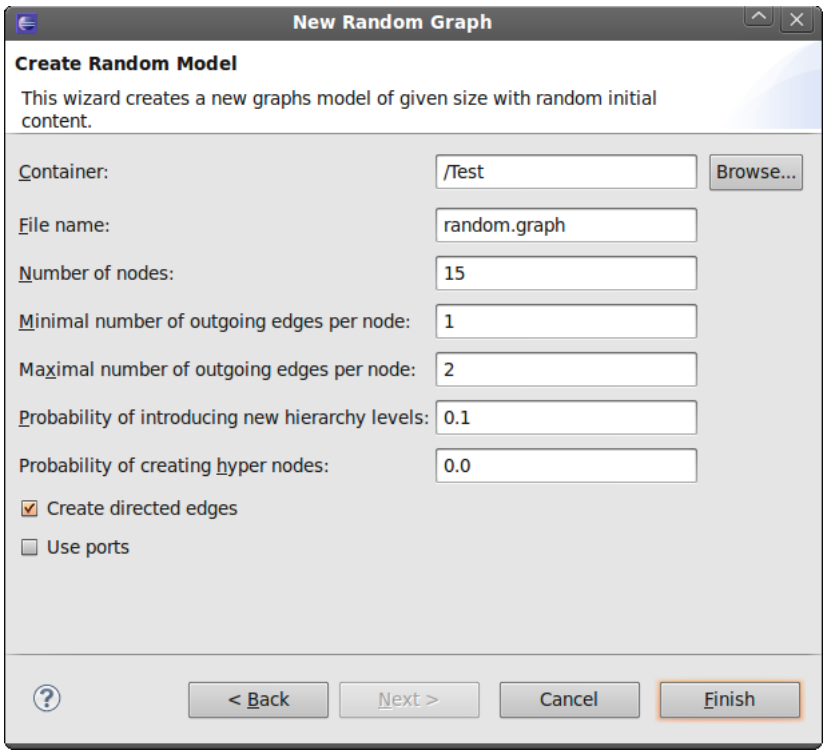



Figure 4.7: Wizard for creating random graphs

**Toggle Directed/Undirected Edge:** Changes a selected directed edge to an undirected edge and vice versa.

**Toggle Node/Hypernode:** Changes a selected node into a hypernode and vice versa.

**Encapsulate:** Adds a new node to the canvas and moves all selected nodes as a nested graph into that node.

**Add Clique:** Prompts the user for a positive whole number and adds a subgraph to the canvas that is the complete graph on the specified number of nodes.

**Add Tree:** Prompts the user for two positive whole numbers and adds a subgraph to the canvas that is a tree with a number of layers equal to the first specified number and with every node having a number of children equal to the second specified number.

**Add Circle:** Prompts the user for a positive whole number and adds a subgraph to the canvas that is a circle on the specified number of nodes.

These commands can be invoked from a number of sources: the toolbar, the KIELER menu and the context menu. The implementation for the Connect Hyperedge command is exemplarily listed in Listing 4.2. In lines 3 to 14 Java functionality is used

Listing 4.2: Snippets from the Xtend command definition file

```
1  import graphs;
2
3  Integer getIntFromUser(String userMessage, Integer defaultValue):
4    JAVA de.cau.cs.kieler.ksbase.util.UserDialogUtil.getUserInt(
5        java.lang.String, java.lang.Integer)
6  ;
7
8  Integer getPositiveIntFromUser(String userMessage, Integer defaultValue, Integer min):
9    let n = getIntFromUser(userMessage, defaultValue):
10   if n < min then min else n
11 ;
12
13 Integer getPositiveIntFromUser(String userMessage, Integer defaultValue):
14   getPositiveIntFromUser(userMessage, defaultValue, 1)
15 ;
16
17 //Connects any number of nodes with a hyperedge
18 Void connectHyper(List[Node] nodes):
19   let hypernode = new Node:
20   hypernode.setIsHypernode(true) ->
21   nodes.connectUndirectedReverse(hypernode) ->
22   nodes.get(0).parent.children.add(hypernode) ->
23   setSelection(hypernode)
24 ;
25
26 //Connects two nodes with an undirected edge (parameter ordering reversed)
27 Void connectUndirectedReverse(Node target, Node source):
28   let edge = new Edge:
29   edge.setIsDirected(false) ->
30   edge.setSource(source) ->
31   edge.setTarget(target) ->
32   setSelection(edge)
33 ;
34
35 //Sets the object that should be selected after the transformation is executed
36 Void setSelection(Object object):
37  JAVA de.cau.cs.kieler.ksbase.ui.utils.TransformationUtil
38    .setPostTransformationSelection(java.lang.Object)
39 ;
```

to define the user prompts.

In line 17 the definition of the `connectHyper` extension begins, the attribute `List[Node] nodes` implicitly defines that the command can be invoked on any number of selected nodes. Once executed a new hypernode is created in lines 18 and 19, followed by the connection of all selected nodes to the hypernode in line 20 using the extension `connectUndirectedReverse`. The hypernode is then added to the parent of the first selected node in line 21.

Another feature that was realized using the KSBasE framework are Copy and Paste commands. The GMF default handler just copies elements in the notation model, i.e. the actual graph is not altered. To prepare a custom handler for Copy and Paste the default one was disabled as mentioned in Section 4.2.3. A custom handler was installed using the appropriate extension point and using KSBasE to realize the functionality.

## 4.5 Graph Import

Various file formats exist to store graph structures, two of which have been introduced in Section 3.3. Importing a graph from such a file can be a fast and easy way to obtain a graph with specific properties.

In Section 3.1.6 the possibility to use Xtend to transform between different EMF models was elucidated. This leads to the following approach for realizing an import function for a given graph file format:

1. Create an EMF metamodel for the data structure defined by the file format.
2. Implement the functionality to load a file of the file format into an instance of that metamodel.
3. Build an Xtend transformation that transforms from that metamodel to the GRAPHS metamodel.

The actual import process for a graph defined in a file of a supported file format can then be performed in three steps:

1. Load the file into the appropriate metamodel instance.
2. Execute the Xtend transformation file to receive a GRAPHS model that represents the graph.
3. Serialize the GRAPHS model to XML using the functionality provided by EMF.

In Section 3.1.4 it was mentioned that XML schema files are a valid source for creating an EMF metamodel. This makes importing graph file formats that are based on an XML schema especially comfortable, as the only real work that has to be done is writing the Xtend transformation file.

### 4.5.1 Model-to-Model Transformation with Xtend

Using Xtend to transform between two metamodels has a few advantages over using plain Java:

- It is specialized on such transformations, i.e. it features some mechanics that support model transformation, e.g. create-extensions (see Section 3.1.6).
- Xtend is a scripting language and inherits the common advantages of such languages, like improved maintainability and a faster development time.

In the following the Xtend transformation for GraphML (see Section 3.3.1) will be explained in detail to serve as a sufficient example, because all transformations have a similar structure.

The transformation starts in the extension `transform(graphml::Document-Root doc)` in line 4 with the parameter being the root element of a GraphML

file, containing exactly one `GraphmlType` attribute which is passed on to `graphml::GraphmlType graphml` in line 8 using Xtends polymorphism capability. This extension distinguishes between the number of graphs which are defined in the document: if zero graphs are defined the transformed graph is simply a `Node` without children, one graph is transformed to a`Node` that represents that graph and any greater number of graphs are interpreted as nested graphs in the transformed graph. The transformations for the latter two cases do not differ much: first all nodes in the graph(s) are transformed and then the edges and hyperedges as shown in lines 15 to 17 and lines 21 to 23. The only differences are the use of Xtends mechanism to call extensions on lists in the latter case and the two extensions `transformGraph` and `transformGraphNew`, which is simply nesting the graph inside another `Node` but is proceeding in the same manner as `transformGraph` then. Both extensions call `transformNode` on every node in the current graph in lines 29 and 36 and in addition pass the current parent `Node`. The `transformNode` extension in line 39 adds the passed node to the passed graph by calling the `getNode` in line 50 `create`-extension for the first time with the node identifier, calls `transformPort` on all attached ports and if present recursively calls `transformGraph` on any nested graph. The `transformPort` extension in line 46 creates the port by calling the `create`-extension `getPort` in line 59 for the first time with the ports name and the node it is attached to.
After all nodes are transformed the extensions `transformGraphEdges` and `transformGraphHyperedges` are called for every graph in lines 16, 17, 22 and 23. The extension `transformGraphEdges` in line 64 calls `transformEdge` on every edge in the current graph in line 65 and 66, passing the default edge direction as an argument, and `transformNestedGraphEdges` on every node in line 67, which recursively transforms all edges in all nested graphs. The `transformEdge` extension in line 60 creates a new `Edge` and fetches target nodes and ports using the `getNode` and `getPort` extensions; this time the `create` modifier makes sure the nodes and ports are not created for a second time. In this step a problem occurred, because the GraphML schema permits edges to specify their type as directed or undirected but this is not mandatory. If an edge in the GraphML definition does not specify the type, the `boolean` attribute directed in the EMF model representing this option is set to `false`, which is also true for undirected edges. This problem cannot be solved with the EMF model that was generated from the XML schema; the current implementation reinterprets the `directed` attribute of a GraphML graph so that all edges will be directed when that attribute is set.
The extension `transformGraphHyperedges` in line 95 calls `transformHyperedge` on every hyperedge in that graph in line 96 and `transformNestedGraphHyperedges` on every node in line 97, which recursively transforms all hyperedges in all nested graphs. The extension `transformHyperedge` in line 105 creates a new `Node` as hypernode and calls the extension `connectEndpointToHypernode` on every endpoint of the GraphML hyperedge in line 109. The extension `connectEndpointToHypernode` in line 112 creates a new `Edge` with the hypernode as source and fetches the target node and port using the `getNode` and `getPort` extensions.

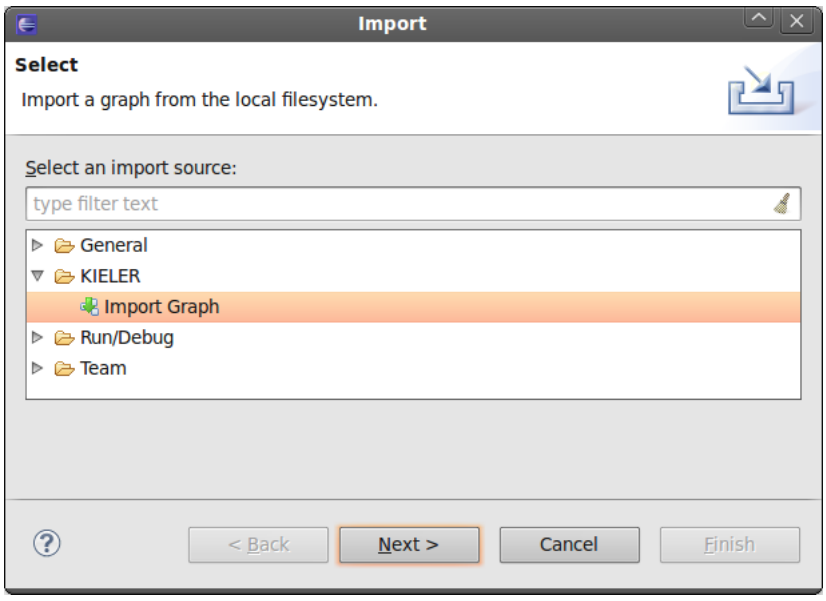



Figure 4.8: Eclipse import wizard showing the Import Graph entry

### 4.5.2 Import Wizard

To invoke the transformations through the UI, an import wizard has been added to the Eclipse import mechanism by providing an extension. In Figure 4.8 the *Import Graph* entry can be seen in the Eclipse import wizard.

Listing 4.3: Xtend transformation file for GraphML-to-GRAPHS- Part 1

```
1  import graphs;
2  import graphml;
3
4  Node transform(graphml::DocumentRoot doc):
5    transformGraphML(doc.graphml)
6  ;
7
8  Node transformGraphML(graphml::GraphmlType graphml):
9    switch (graphml.graph.size) {
10   case 0:
11     (let parent = new Node:
12     parent)
13   case 1:
14     (let parent = new Node:
15     transformGraph(graphml.graph.get(0), parent) ->
16     transformGraphEdges(graphml.graph.get(0)) ->
17     transformGraphHyperedges(graphml.graph.get(0), parent) ->
18     parent)
19   default:
20     (let parent = new Node:
21     graphml.graph.transformGraphNew(parent) ->
22     graphml.graph.transformGraphEdges() ->
23     graphml.graph.transformGraphHyperedges(parent) ->
24     parent)
25   }
26 ;
27
28 Void transformGraph(graphml::GraphType graph, Node parent):
29   graph.node.transformNode(parent)
30 ;
31
32 Void transformGraphNew(graphml::GraphType graph, Node parent):
33   let newNode = new Node:
34   newNode.setNodeLabel(graph.id) ->
35   parent.children.add(newNode) ->
36   graph.node.transformNode(newNode)
37 ;
38
39 Void transformNode(graphml::NodeType node, Node parent):
40   let newNode = getNode(node.id):
41   parent.children.add(newNode) ->
42   node.port.transformPort(newNode) ->
43   (if node.graph != null then transformGraph(node.graph, newNode))
44 ;
45
46 Void transformPort(graphml::PortType port, Node node):
47   getPort(port.name, node)
48 ;
49
50 Node getNode(String id):
51   let node = id == null ? new Node : getNodeHelper(id):
52   node
53 ;
54
55 create Node getNodeHelper(String id):
56   this.setNodeLabel(id)
57 ;
58
59 create Port getPort(String name, Node node):
60   this.setNode(node) ->
61   this.setPortLabel(name)
62 ;
```

Listing 4.4: Xtend transformation file for GraphML-to-GRAPHS- Part 2

```
64  Void transformGraphEdges(graphml::GraphType graph):
65    graph.edge.transformEdge(
66        graph.edgedefault == graphml::GraphEdgedefaultType::directed) ->
67    graph.node.transformNestedGraphEdges()
68  ;
69
70  Void transformNestedGraphEdges(graphml::NodeType node):
71    if node.graph != null then transformGraphEdges(node.graph)
72  ;
73
74  Void transformEdge(graphml::EdgeType edge, boolean directed):
75    let source = getNode(edge.source):
76    let target = getNode(edge.target):
77    let newEdge = new Edge:
78    newEdge.setMidLabel(edge.id) ->
79    newEdge.setIsDirected(directed || edge.directed) ->
80    newEdge.setSource(source) ->
81    newEdge.setTarget(target) ->
82    (if edge.sourceport != null then newEdge.setSourcePort(
83        getPort(edge.sourceport, source))) ->
84    (if edge.targetport != null then newEdge.setTargetPort(getPort(
85        edge.targetport, target))) ->
86    (if edge.sourceport != null
87          then (if edge.targetport != null
88               then newEdge.setType(graphs::EdgeType::Port2Port)
89               else newEdge.setType(graphs::EdgeType::Port2Node))
90          else (if edge.targetport != null
91               then newEdge.setType(graphs::EdgeType::Node2Port)
92               else newEdge.setType(graphs::EdgeType::Node2Node)))
93  ;
94
95  Void transformGraphHyperedges(graphml::GraphType graph, Node parent):
96    graph.hyperedge.transformHyperedge(parent) ->
97    graph.node.transformNestedGraphHyperedges()
98  ;
99
100 Void transformNestedGraphHyperedges(graphml::NodeType node):
101   if node.graph != null then transformGraphHyperedges(
102       node.graph, getNode(node.id))
103 ;
104
105 Void transformHyperedge(graphml::HyperedgeType hyperedge, Node parent):
106   let hypernode = new Node:
107   hypernode.setIsHypernode(true) ->
108   parent.children.add(hypernode) ->
109   hyperedge.endpoint.connectEndpointToHypernode(hypernode)
110 ;
111
112 Void connectEndpointToHypernode(graphml::EndpointType endpoint, Node hypernode):
113   let target = getNode(endpoint.node):
114   let newEdge = new Edge:
115   newEdge.setMidLabel(endpoint.id) ->
116   newEdge.setIsDirected(false) ->
117   newEdge.setSource(hypernode) ->
118   newEdge.setTarget(target) ->
119   (if endpoint.port != null then newEdge.setTargetPort(
120       getPort(endpoint.port, target))) ->
121   (if endpoint.port != null
122     then newEdge.setType(graphs::EdgeType::Node2Port)
123     else newEdge.setType(graphs::EdgeType::Node2Node))
124 ;
```

# 5 Graph Analysis

In this chapter the design and implementation of the *Graph Analysis* framework will be explained.

The latter two problems listed in Section 1.1 are handled in this chapter, which are:

4. Develop an easily extensible mechanism to analyze a given graph diagram and visualize the results of this analysis. This should also function as a constraint checker.

5. Implement a number of graph analysis algorithms as a basis using the mechanism mentioned above.

## 5.1 Concepts and Features

Before any concepts can be discussed a definition is necessary:

**Definition.** *An algorithm that takes a graph structure with attached layout-information as input and returns any kind of data is called a* **graph analysis**. *To perform a graph analysis with an input graph is also called:* **to analyze the graph**.

The purpose of GrAna is to supply a framework for the KIELER project that supplies a graphical and programmatical interface for executing and visualizing graph analyses.

The following requirements are desirable for the framework to be usable:

**Extensibility:** It should be possible to supply graph analyses to the framework without altering the implementation.

**Performance:** The number of duplicate computations should be kept at a minimum.

**Usability:** Launching a graph analysis, through the UI or programmatically, has to be an uncomplicated task.

**Flexibility:** The usage of the framework must be as unrestricted as possible, i.e. the invocation of graph analyses has to be possible on a high-level, but the low-level way should be possible as well. The visualization should be able to be used in a whole, only partially or not at all.

### 5.1.1 KGraph-based Graph Analysis Mechanism

To make GrAna compatible to other KIELER projects, the graph structure that serves as the input format for the graph analyses is the KGraph (see Section 3.2.1).
This immediately provides one useful feature: with the KIML it is possible to build a KGraph from every GMF diagram. By reusing this functionality it is also possible to analyze every GMF diagram.

### 5.1.2 Dependencies

For performance reasons and to avoid duplicate code it is necessary that analyses can reuse the functionality of other analyses. In GrAna this is accomplished by letting analyses depend on other analyses.

The first approach to solve this problem would be to simply let an analysis depend on other analyses, in a way that the algorithm receives the results of the dependencies as additional input.
While this approach solves the problem in most cases, it still can produce unnecessary overhead. Consider the calculation of the aspect ratio of a drawing (see Section 2.2.1) as an example analysis. The information needed to calculate this are the width and the height of the drawing. Both can be calculated by traversing all nodes in the graph, which requires $\mathcal{O}(n)$ operations. If the aspect ratio analysis now simply has dependencies on a width and a height analysis, all nodes would be traversed twice instead of only once. To solve this problem, the second approach provides two kinds of dependencies: strong dependencies and weak dependencies:

**Strong dependency:** A strong dependency matches the description of a dependency in the first approach. I.e. when performing an analysis that has a strong dependency, the dependency is performed before that analysis. Cyclic dependencies are not allowed.

**Weak dependency:** If an analysis has a weak dependency, the dependency is only performed when the user does so explicitly. I.e., if a set of analyses is about to be called, GrAna tries to find a schedule where strong dependencies have to precede their dependents and as many weak dependencies as possible precede their dependents. This only works if every analysis includes code to calculate the results of weak dependencies. This kind of dependency can form cycles without consequences.

### 5.1.3 Visualizers

According to the definition graph analyses can return any kind of data, e.g. `Integer`, `Boolean`, `List<String>` or any other classes. In order to give the user a possibility to evaluate these data, they have to be visualized depending on the type. In GrAna this is done with visualizers. Every visualizer is able to decide whether it can visualize given data or not and at what priority it is chosen. I.e. visualizers with a

higher priority are preferred when visualizing the result.
To keep the visualization as abstract as possible it uses HTML to display the results.

### 5.1.4 Contributions

There are a number of ways to realize the contribution of graph analyses and visualizers. The Eclipse-recommended way is the usage of the extension point mechanism (see Section 3.1.2). This will also be the main approach used in GrAna. It will allow the developer to supply graph analyses, categories to sort the analyses into and visualizers by supplying small amounts of Java code. Additional information like identifier, names and descriptions can or have to be supplied as well.

Another way to allow contributions is to integrate some kind of scripting functionality, which allows the user to supply analyses at runtime. An example for this approach is given in Section 5.4.

## 5.2 Implementation of GrAna

GrAna is part of the KIELER project and in particular the KIML, and as such part of the KIELER view component. The functionality is contained in an Eclipse plugin (see Section 3.1.1).

### 5.2.1 IAnalysis

The central interface for the GrAna framework is the `IAnalysis` interface as shown in Listing 5.1. Every Java representation for a graph analysis in GrAna inherits from this interface. It declares only one method, `doAnalysis`, which takes three arguments:

**`KNode parentNode`:** A `KNode` is the representation for a node and a whole graph in the KGraph, in this case it represents the graph which is about to be analyzed.

**`Map<String, Object> results`:** A Java `Map` stores a number of key-value pairs, in this case the key is a `String` and should be the identifier of a graph analysis and the value is the result of that analysis. In particular all results of strong dependencies should be contained in this argument, as well as the results of any number of weak dependencies.

**`IKielerProgressMonitor progressMonitor`:** The `IKielerProgressMonitor` is the interface for all KIELER progress monitors, i.e. a class that keeps track of the process of a task, in this case the analysis.

The return value of the `doAnalysis` method is of type `Object` which matches the definition of the graph analysis in Section 5.1, i.e. a graph analysis can return any kind of data.

Listing 5.1: IAnalysis interface

```
1  public interface IAnalysis {
2      /**
3       * Performs the actual analysis process and returns the results.
4       *
5       * @param parentNode
6       *            the parent node which the analysis is performed on
7       * @param results
8       *            the result of analyses that were performed before this one (it
9       *            should include the results of all dependency analyses)
10      * @param progressMonitor
11      *            progress monitor used to keep track of progress
12      * @throws KielerException
13      *             if the method fails to perform the analysis
14      * @return the analysis results
15      */
16     Object doAnalysis(KNode parentNode, Map<String, Object> results,
17             IKielerProgressMonitor progressMonitor) throws KielerException;
18 }
```

Besides this interface an abstract class for analyses exists, named `AbstractInfoAnalysis`, which implements `IAnalysis` and `IDepending<String>`, an interface for classes with an identifier, strong and weak dependencies, and additional getters for a name and a description. Most analyses inside of GrAna inherit from this class.

### 5.2.2 AbstractAnalysisResultVisualizer

As stated in Section 5.1.3, the visualization of analysis results is done by producing HTML depending on the type of the result. The `AbstractAnalysisResultVisualizer` class can be seen in Listing 5.2. It is the base for classes which provide additional visualization and it declares three methods, two of which are abstract:

**`abstract boolean canVisualize(Object result)`:** This method checks whether or not the visualizer can visualize the given result.

**`boolean usesResultDialog()`:** By default every visualizer produces HyperText makeup Language (HTML) code, but it is not mandatory. A visualizer that returns `false` in this method is expected to visualize the result by other means, e.g. by updating a view.

**`abstract String visualize(final Object result)`:** This method performs the actual visualization. If `canVisualize` and `usesResultDialog` both return true, it is supposed to return a `String` containing a HTML visualization of the result.

The easiest example for a visualizer is the `toStringVisualizer`, which is supplied by GrAna. It simply supports every type of result and visualizes by using the `toString` method, which is supported by every Java class. The priority of the `toStringVisualizer` is 1, it is always chosen last. How to set the priority is explained in the next Section.

Listing 5.2: AbstractAnalysisResultVisualizer class

```
1  public abstract class AbstractAnalysisResultVisualizer {
2      /**
3       * Returns whether this class can visualize the given analysis result.
4       *
5       * @param result
6       *             the result of an analysis
7       * @return true if this class can visualize the result
8       */
9      public abstract boolean canVisualize(Object result);
10
11     /**
12      * Returns whether this class uses the main result dialog.
13      *
14      * @return true if this class uses the result dialog
15      */
16     public boolean usesResultDialog() {
17         return true;
18     }
19
20     /**
21      * Visualizes the given result object by returning html if {@code
22      * canVisualize} returns true for the given result. Returns null if
23      * {@code usesResultDialog} returns true.
24      *
25      * @param result
26      *             the result to visualize
27      * @return the html to display in the result dialog or null if this
28      *         visualizer displays the results in another way
29      */
30     public abstract String visualize(final Object result);
31 }
```

### 5.2.3 IAnalysisBundle

The `IAnalysisBundle` is an interface for classes that contribute analyses at runtime. The Java definition can be seen in Listing 5.3.

**`Collection<AbstractInfoAnalysis> getAnalyses()`:** This method returns all analyses that are provided through this bundle.

**`void addBundleChangedListener(...)`:** Adds a listener to the bundle; a listener must implement the interface `IBundleChangedListener`, which declares two functions: one that is called when an analysis is added and one when an analysis is removed.

**`void removeBundleChangedListener(...)`:** Removes a listener from the bundle.

### 5.2.4 AnalysisProvider Extension Point

The AnalysisProvider extension point is the main mechanic for contributing to the GrAna plugin as stated in Section 5.1.4. It allows four kinds of contributions, which will be explained in the following:

Listing 5.3: IAnalysisBundle interface

```
1  public interface IAnalysisBundle {
2      /**
3       * Returns a collection of all analyses provided by this bundle.
4       *
5       * @return the analyses
6       */
7      Collection<AbstractInfoAnalysis> getAnalyses();
8
9      /**
10      * Adds a listener to the bundle.
11      *
12      * @param listener
13      *             the listener
14      */
15     void addBundleChangedListener(final IBundleChangedListener listener);
16
17     /**
18      * Removes a listener from the bundle.
19      *
20      * @param listener
21      *             the listener
22      */
23     void removeBundleChangedListener(final IBundleChangedListener listener);
24 }
```

**category:** Given a unique identifier, a name and a description, categories can be added to GrAna to sort graph analyses into.

**provider:** A provider adds new analyses to GrAna. The information required for this contribution are a unique identifier, a name, a description and a Java class that implements `IAnalysis`. Optionally a category can be defined; if no category is given, the analysis is sorted into the default category (Other). Additionally any number of analyses dependencies can be defined as either strong or weak dependencies.

**visualizer:** To add a visualizer, a class that extends `AbstractResultVisualizer` and a priority have to be specified.

**bundle:** The bundle is an alternative way to supply analyses. It allows the developer to add a manager to GrAna which can add analyses at a later point during runtime. A name, a description and a class implementing `IAnalysisBundle` are required for this extension element.

An example XML definition for an extension to this extension point can be seen in Listing 5.4.

### 5.2.5 DependencyGraph

The `DependencyGraph` is a helper class designed to solve the problem that was outlined in Section 5.1.2. The interface for this class, `IDependencyGraph`, can be seen in Listing 5.5. As the problem of computing a scheduling for entities with

Listing 5.4: An example extension to the AnalysisProviders extension point

```
<extension
      point="de.cau.cs.kieler.kiml.grana.analysisProviders">
   <category
         description="An example category."
         id="de.cau.cs.kieler.kiml.grana.example.exampleCategory"
         name="Examples">
   </category>
   <provider
         category="de.cau.cs.kieler.kiml.grana.example.exampleCategory"
         class="de.cau.cs.kieler.kiml.grana.example.ExampleAnalysis"
         description="A description of the example analysis."
         id="de.cau.cs.kieler.kiml.grana.example.exampleAnalysis"
         name="Example Analysis">
      <dependency
         analysis="de.cau.cs.kieler.kiml.grana.nodeCount">
      </dependency>
   </provider>
   <visualizer
         class="de.cau.cs.kieler.kiml.grana.example.ExampleVisualizer"
         priority="15">
   </visualizer>
   <bundle
         class="de.cau.cs.kieler.kiml.grana.example"
         description="This could be a script-analyzes provider."
```

dependencies is a common problem the implementation was kept as abstract as possible by using interfaces and Java generics, i.e. it manages objects that implement `IDepending<S>`. The interface declares five methods, where `T` is a class that implements `IDepending<S>` and `S` is a comparable class:

**`boolean add(final T object)`:** Adds an entity to the graph and tries to resolve the dependencies; if a dependency is missing the entity is not added and the method returns `false`, else `true` is returned.

**`List<T> remove(final T entity)`:** Removes an entity and all dependent entities; returns a list containing all removed entities.

**`List<T> addAll(final Collection<T> entities)`:** Adds a collection of entities to the graph and tries to resolve the dependencies. Furthermore removes entities which are part of a cyclic dependency. Returns all entities which had missing dependencies or were part of a cycle. This method always tries to add as many entities as possible.

**`List<T> computeScheduling(final List<T> entities)`:** Given a list of entities, this method adds all strong dependencies and computes a scheduling that respects strong dependencies and as many weak dependencies as possible.

### 5.2.6 AnalysisServices

This class is a singleton, it serves as the core programmatical interface for the usage of GrAna and loads the configuration from the extension point. It provides the

Listing 5.5: IDependencyGraph interface

```
1  public interface IDependencyGraph<S extends Comparable<S>, T extends IDepending<S>> {
2      /**
3       * Adds an entity to the graph if all dependencies can be resolved.
4       *
5       * @param entity the entity
6       * @return true if the entity was added
7       */
8      boolean add(final T entity);
9
10     /**
11      * Removes an entity from the graph and all entitys depending on it.
12      *
13      * @param entity the entity to remove
14      * @return the removed entitys
15      */
16     List<T> remove(final T entity);
17
18     /**
19      * Adds a collection of entities to the graph and tries to resolve
20      * dependencies.<br>
21      * Returns a list of entities that could not be added cause they had missing
22      * dependencies or were part of a cycle.
23      *
24      * @param entities the entities to add
25      * @return the list of entities that could not be added
26      */
27     List<T> addAll(final Collection<T> entities);
28
29     /**
30      * Returns a sorted list of the entities so that an entity that depends on
31      * another entity precedes it in the list. Removes entities that are not
32      * represented in this graph.
33      *
34      * @param entities the entities
35      * @return a sorted list respecting dependencies between the entities
36      */
37     List<T> computeScheduling(final List<T> entities);
38 }
```

following functionality:

**Getters:** The singleton provides several methods to obtain the analyses, categories and visualizers as lists and by identifier. Note that the returned analyses inherit from `AbstractInfoAnalysis` and have the extension point data attached. The same is true for the categories and visualizers.

**Scheduling:** To manage the analyses dependencies a `DependencyGraph<String, AbstractInfoAnalysis>` is used with the analyses as entities. The `AnalysisServices` contain a method that has a signature that is identical to the `computeScheduling` method in the `DependencyGraph` and simply forwards the method call.

The actual invocation of graph analyses can either be done manually, which is the low-level approach, or using a helper class, namely `DiagramAnalyzer`, the high-level approach. The former allows other plug-ins to reuse the functionality of GrAna,

while being able to e.g. cache analysis results and optimize the analysis process for the specific application. The latter allows the developer to e.g. launch an analysis on the active editor by calling only a few methods. In Listing 5.6 an example can be

Listing 5.6: An example for programmatical GrAna usage

```
1  public Object execute(final ExecutionEvent event) throws ExecutionException {
2      // get the active editor
3      IEditorPart editorPart = HandlerUtil.getActiveEditor(event);
4      // specify the analysis to perform
5      AbstractInfoAnalysis analysis =
6          AnalysisServices.getAnalysisById("de.cau.cs.kieler.grana.nodeCount");
7      // perform the analysis on the active diagram
8      Object result = DiagramAnalyzer.analyze(editorPart, null, analysis, true);
9      // print the result
10     System.out.println(result);
11 }
```

seen that shows the `execute` method of a common Eclipse command handler, i.e. the method that is invoked when e.g. pressing a toolbar button. In line 3 an Eclipse utility class is used to get the active editor, in lines 5 and 6 `AnalysisServices` is utilized to get an analysis, in line 8 the `DiagramAnalyzer` utility class is called to perform the analysis while showing a progress bar, which shows the progress of the analysis, and in line 10 the result is printed to `stdout` (without the usage of a visualizer).

### 5.2.7 Analyses Selection Dialog

The dialog for the selection of analyses is one part of the GrAna UI contributions. It is implemented using the Standard Widget Toolkit (SWT) `CheckedTreeSelectionDialog` as a base class, which provides, together with a custom content provider, the functionality that can be seen in Figure 5.1.

After the user selects a number of analyses and presses the *OK* button the selection is saved to the Eclipse preference store, which is an Eclipse mechanic to store persistent data.

### 5.2.8 Result Dialog and View

The result dialog and the result view are the two UI contributions that can be used to display the results of a set of analyses using the visualizers. They both contain an SWT browser, an element that can display HTML. A default way of performing a set of analyses is also implemented:

**Analyze:** This button has been added to the toolbar. Pressing it performs the analyses that were selected in the analyses selection dialog by loading the required data from the preference store. See Listing 5.7 for the implementation of the handler that is responsible for performing the analyses once the analyze button is pressed. The methods `getLastAnalysesSelection`,

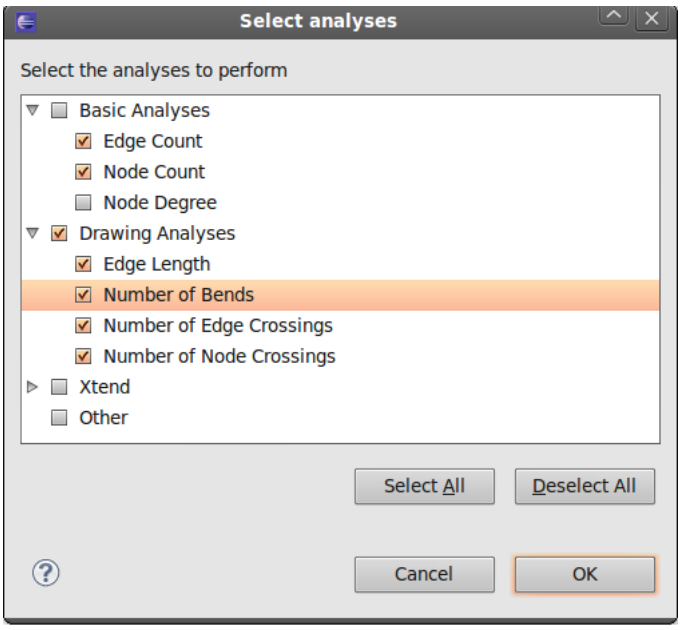



Figure 5.1: Analyses selection dialog

`isResultViewEnabled` and `isResultDialogEnabled` access the preference store to fetch the required options; their implementation is omitted as it is not important for the understanding of the handler.

The result of a set of analyses, visualized in the dialog and view, together with the analyzed graph can be seen in Figure 5.2. At the moment the combination of the visualizations that is shown there is the only implemented one; in the future this should be customizable through the preference pages (see next Section).

### 5.2.9 Preference Page

The preference page for GrAna contains three configurable options:

**Enable Result Dialog:** This option defines whether or not the result dialog is shown upon finishing the performing of a set of analyses through the UI.

**Enable Result View:** Same as *Enable Result Dialog* but for the view.

**Perform analyses after layout:** If selected, after every autolayout process performed by KIML, the configured analyses are performed and the result is visualized in the result view.

The options can be seen in Figure 5.3.

Listing 5.7: The handler that starts a set of analyses

```
1  public Object execute(final ExecutionEvent event) throws ExecutionException {
2      Shell shell = HandlerUtil.getActiveWorkbenchWindow(event).getShell();
3      // get the active editor
4      IEditorPart editorPart = HandlerUtil.getActiveEditor(event);
5      // get the last selected analyses
6      List<AbstractInfoAnalysis> analyses = getLastAnalysesSelection();
7      // perform the analyses on the active diagram
8      Map<String, Object> results =
9          DiagramAnalyzer.analyze(editorPart, null, analyses, true);
10     // is the view enabled?
11     if (isResultViewEnabled()) {
12         // refresh the result view
13         AnalysisResultViewPart view = AnalysisResultViewPart.findView();
14         if (view != null) {
15             view.setAnalysisResults(analyses, results);
16         }
17     }
18     // is the dialog enabled?
19     if (isResultDialogEnabled()) {
20         // prepare the result dialog
21         AnalysisResultDialog resultDialog =
22             new AnalysisResultDialog(shell, analyses, results);
23         // only show the result dialog if there is something to show
24         if (!resultDialog.isEmpty()) {
25             resultDialog.open();
26         }
27     }
```

## 5.3 Analyses

A number of analyses are supplied by GrAna. They can be categorized into two different categories.

### 5.3.1 Basic Analyses

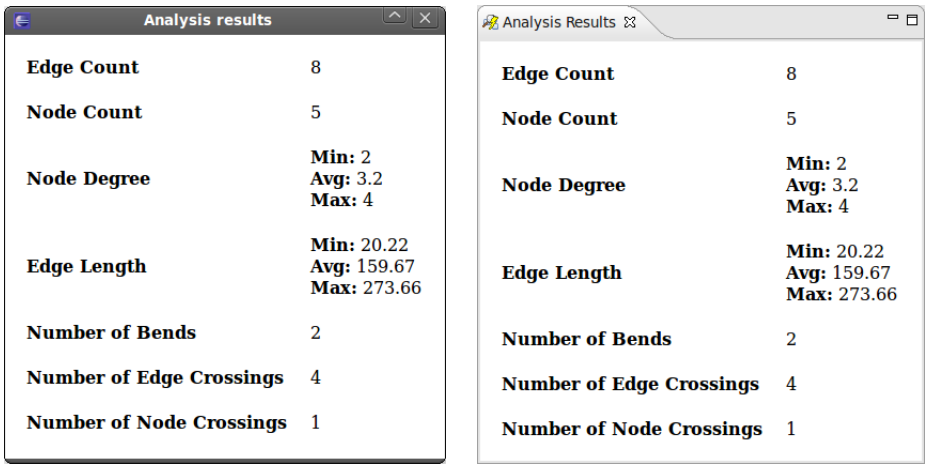
The basic analyses compute simple properties of the graph, which are defined by the structure of the graph and not the drawing:

**Node Count:** The number of nodes in the graph.

**Edge Count:** The number of edges in the graph.

**Node Degree:** The degree of a node is the number of incoming edges plus the number of outgoing edges of that node. This analysis returns an instance of a class named `MinAvgMaxResult`, which can store a minimal, an average and a maximum value. GrAna supplies a special visualizer for this type of result, as shown in Figure 5.2 for the edge length analysis.

**Number of Connected Components:** Returns the number of connected components. Two nodes lie in a different connected component, when they are not connected by a path, i.e. when one can not be reached by the other one following incoming and outgoing edges.




(a) Result Dialog (b) Result View

(c) Analyzed graph

Figure 5.2: GrAna UI contributions to visualize results

### 5.3.2 Drawing Analyses

The drawing analyses compute properties of the graph that are based on the specific drawing; these often resemble properties which are of interest for aesthetic criteria (see Section 2.2.1):

**Width:** The width of the drawings bounding box.

**Height:** The height of the drawings bounding box.

**Aspect Ratio:** The ratio of the longest to smallest side of the drawings bounding box.

**Edge Length:** The minimal, average and maximal edge length.

**Number of Bends:** The number of bend points on the edges.

**Number of Edge Crossings:** The number of edge-edge crossings.

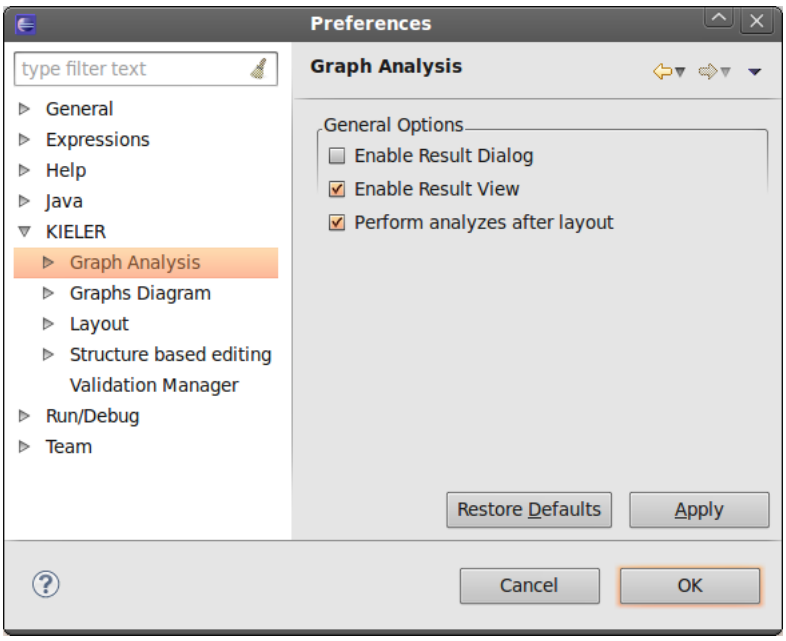



Figure 5.3: GrAna preference page

**Number of Node Crossings:** The number of edge-node crossings.

## 5.4 Exemplary Xtend Analysis Extension

A more complex extension to GrAna is presented in this Section.

So far the only way to contribute analyses is by defining an extension and a Java class, which contains the code for the analysis. For analyses like node crossings and similar complex algorithms this is a sufficient solution, but for e.g. simple combinations of several analyses, this produces a lot of organizational overhead.

An approach to solve this problem is provided in this Section: the integration of Xtend as a scripting language to define analyses. In addition the UI has been extended to allow the user to easily add and manage Xtend analyses.

### 5.4.1 Providing analyses at Runtime

Instead of providing the analyses directly through the extension point, an analysis bundle is used (see Section 5.2.3), which is instead contributed through the extension point.

A specific Xtend analysis represents an extension, which has been defined in an Xtend file. The first parameter to such an extension is the KGraph instance, that represents the graph, the parameters that follow are the dependencies; this means in particular that for an Xtend analysis the ordering of the dependencies is important. The information needed to define an Xtend analysis are the following:

- An identifier (is generated from name, extension and file path)
- A name
- A description
- A file path to the Xtend file containing the extension
- The extension name
- An ordered list of dependencies

This information are stored in a class that inherits from `AbstractInfoAnalysis` (see Section 5.2.1). The `doAnalysis` method contains the most important part for the implementation of the Xtend analysis mechanism and can be seen in Listing 5.8.

From line 7 to 17 the results of dependency analyses are ordered to fit the ordering required by the extension; if a dependency is missing the analysis process is interrupted. From lines 19 to 27 the Xtend transformation framework is initialized by resolving the file path and setting the active extension. If the process fails, the analysis is aborted. From line 29 to 41 the extension is executed; if it does not return anything that is interpreted as a failure.

### 5.4.2 Xtend Analysis Wizard

To add a new Xtend analysis a wizard has to be invoked, which can be done by right-clicking an Xtend file (*.ext) in the Eclipse package explorer and selecting Make Xtend Analysis ... from the KIELER context menu, as depicted in Figure 5.4. The wizard contains three pages, which collect all the necessary data. The general page collects the file path, the name and the description. The extension page contains a table of extensions that could be parsed from the given file. One of this extensions has to be selected. Lastly the dependencies page consists of two tables. The lower one contains all the dependencies of the the analysis, which is about to be created. It can be sorted to fit the extension parameters ordering. The upper one contains all analyses that are known to GrAna and are not currently in the dependencies table. The buttons between the tables can be used to move analyses from one to the other. When pressing finish on the last page, the analysis is added to the list of available analyses.

The management of the Xtend analyses in the UI is done using a preference page, as shown in Figure 5.6. All Xtend analyses are listed there and can be removed or edited. In addition the wizard can be launched to add analyses.

Listing 5.8: `doAnalysis` method for Xtend analyses

```
public Object doAnalysis(final KNode parentNode,
        final Map<String, Object> results,
        final IKielerProgressMonitor progressMonitor)
        throws KielerException {
    progressMonitor.begin("Xtend analysis: " + name, 1);
    // build the parameters
    Object[] parameters = new Object[dependencies.size() + 1];
    parameters[0] = parentNode;
    int i = 1;
    for (String dependencyId : dependencies) {
        Object result = results.get(dependencyId);
        if (result == null) {
            progressMonitor.done();
            return new AnalysisFailed(AnalysisFailed.Type.Dependency);
        }
        parameters[i++] = result;
    }
    // configure the framework
    IPath path = new Path(filename);
    IFile file = ResourcesPlugin.getWorkspace().getRoot().getFile(path);
    framework.setParameters(parameters);
    if (!framework.initializeTransformation(file.getLocation().toString(),
            extension, "de.cau.cs.kieler.core.kgraph.KGraphPackage")) {
        framework.reset();
        progressMonitor.done();
        return new AnalysisFailed(AnalysisFailed.Type.Failed);
    }
    // execute the transformation
    Object result;
    try {
        result = framework.executeTransformation();
        if (result == null) {
            framework.reset();
            progressMonitor.done();
            return new AnalysisFailed(AnalysisFailed.Type.Failed);
        }
    } catch (TransformationException e) {
        framework.reset();
        progressMonitor.done();
        return new AnalysisFailed(AnalysisFailed.Type.Failed);
    }
    progressMonitor.done();
    return result;
}
```

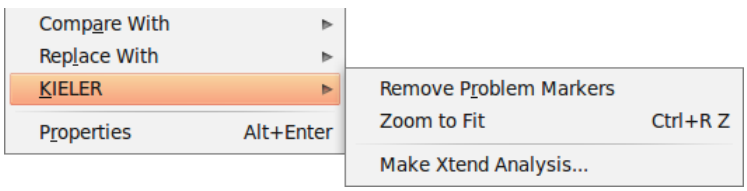
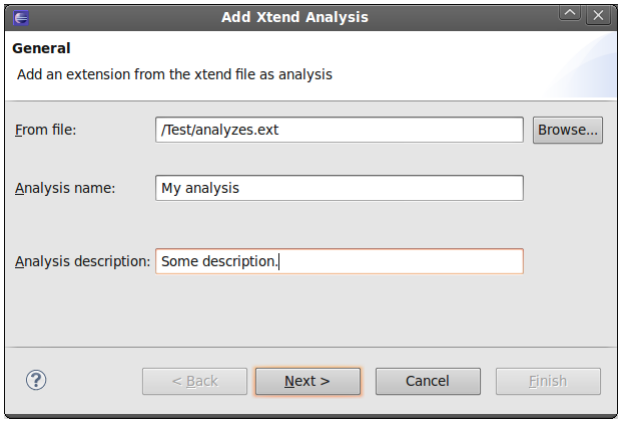
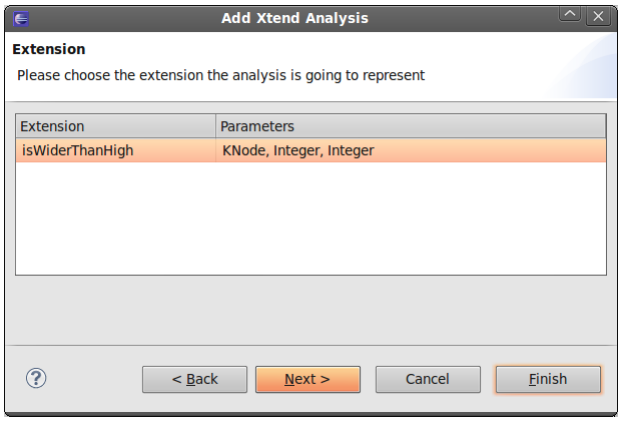


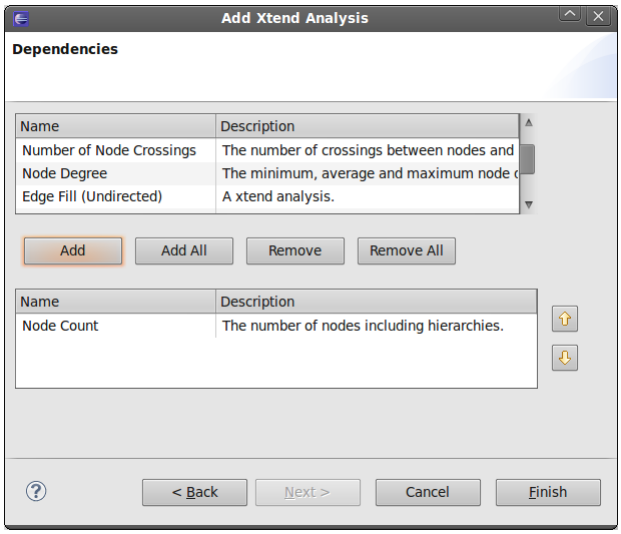

Figure 5.4: Make Xtend analysis context menu entry




(a) General page




(b) Extension page




(c) Dependencies page

Figure 5.5: Add Xtend Analysis wizard

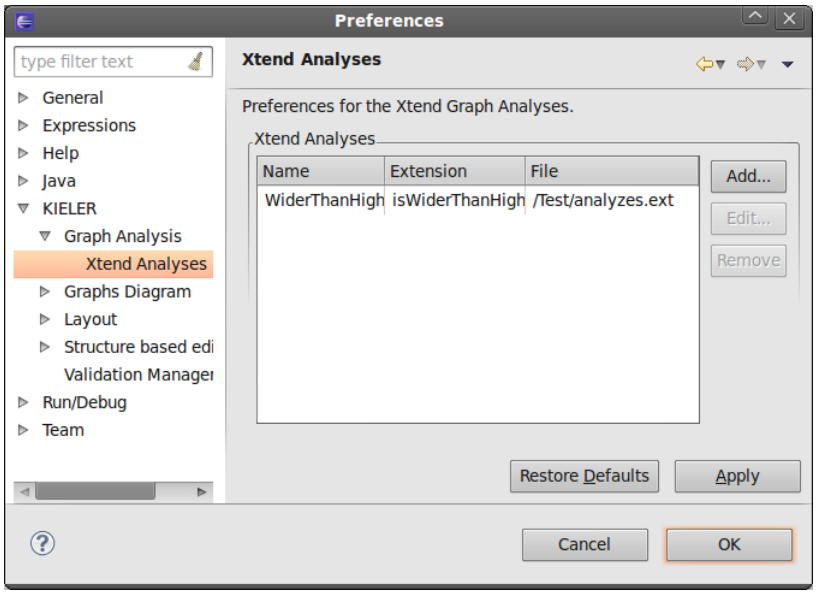



Figure 5.6: Xtend analyses preference page

# 6 Using graphs and GrAna

In this chapter a few common use cases for both GRAPHS and GrAna are presented and explained.

## 6.1 Constructing Test Cases

The graph editor basically supplies three mechanisms to construct test cases:

**Palette and structure-based editing:** The basic way to build a graph is by drag and drop elements from the palette. This is extended by the structure-based editing commands, especially the template commands to create trees, cliques and circles can be used to create specific formations very fast.

**Random graph generation:** If no specific graph is required or as the start point for building a test case, the functionality to generate random graphs can be utilized.

**Graph import:** The most comfortable way to get a graph with specific properties is by simply importing it.

Whatever way is chosen to construct the graph, GrAna can be used to verify that the graph has the required properties.

## 6.2 Layouter Configuration

When developing a layout algorithm eventually a number of parameters appear, which value can not be determined exactly. Given a workbench setup as shown in Figure 6.1, the following workflow is possible:

1. Adjust the values for the parameters in the layout view.
2. Launch a layout process.
3. Manually perform a number of analyses or enable the automatic analysis after a layout process and check the properties.
4. Repeat this cycle as many times as necessary.

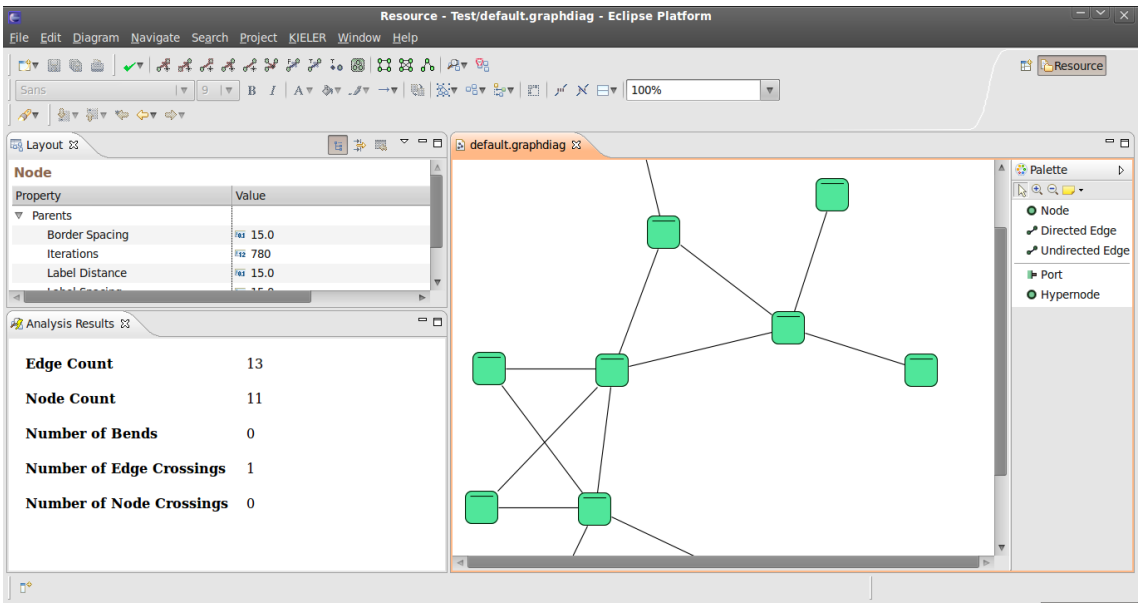



Figure 6.1: Layouter configuration workbench setup

## 6.3 User-defined Constraints

In this use case a number of constraints for various properties of a graph exist, which have to be fulfilled, e.g. when configuring a layouter (see above) or modeling manually. The easiest way to do this is by using Xtend analyses. An example can be seen in Listing 6.1.

Listing 6.1: Xtend constraint analyses

```
1 import kgraph;
2
3 Boolean constraint1(KNode graph, Integer edgeCrossings):
4    edgeCrossings < 5
5 ;
6
7 Boolean constraint2(KNode graph, Integer edges, Integer numberOfBends):
8    numberOfBends / edges < 3
9 ;
```

# 7 Conclusion

The last chapter of this thesis will summarize the results and give an outlook on possible future work concerning GRAPHS and GrAna.

## 7.1 Summary

As stated in Section 1.1, the problem consists in the development of a graph editor and a graph analysis mechanism. To achieve this a number of subtasks have been mentioned which together complete the overall goal; recalling the progress throughout the thesis shows that all these tasks have been completed:

1. Construct a graph editor that is compatible to the KIELER framework.

In Section 4.1 the concepts and features have been introduced and in Section 4.2 the construction and generation of the editor from various definition and generator models is explained in detail.

2. Provide structure based editing commands for the graph editor.

The integration of structure-based editing using the KSBasE framework has been explained in Section 4.4.

3. Allow the import of graphs from common graph file formats.

In Section 4.5 an abstract approach for the import of graphs from file formats has been discussed and the implementation for a specific file format has been explained in detail.

4. Develop an easily extensible mechanism to analyze a given graph diagram and visualize the results of this analysis. This should also function as a constraint checker.

In Section 5.1 the concepts for GrAna have been discussed, in particular different approaches to improve performance and usability, and in Section 5.2 the implementation has been explained. The possibility to use GrAna as a constraint checker is presented in Section 6.3.

5. Implement a number of graph analysis algorithms as a basis using the mechanism mentioned above.

In Section 5.3 a number of analyses for GrAna have been presented, which serve as a basis for other analyses and as examples for the usage of the framework. In addition a flexible way to add analyses at runtime has been elucidated and explained in Section 5.4.

## 7.2 Future Work

Even though all tasks for this thesis have been completed, there are still a lot of possible improvements:

**Real KGraph-based editor:** At the moment GRAPHS is build on a metamodel that inherits from the KGraph (see Section 4.2.1), because of some missing GMF features. As soon as it is possible to express the current graph editor using the KGraph, the project should be updated.

**Complex structure-based editing:** Most of the structure-based editing commands, which are implemented so far, perform only trivial transformations. For the construction of test-cases more complex commands could be helpful, e.g. transformations that preserve specific graph properties like planarity.

**More import file formats:** Currently only GraphML can be imported. Graph file formats such as the ones mentioned in Section 3.3.2 can be rich sources for graphs with specific properties.

**Complex analyses:** Some non-trivial analyses are already present and have been described in Section 5.3. To further ease the development of graph algorithms more analyses should be developed and added to GrAna.

**More scripting for GrAna:** As described in Section 5.4 Xtend has been successfully integrated as a scripting language for GrAna analyses. However for complex algorithms Xtend can be too limited; connecting scripting languages like Lua, Python or Ruby could solve this problem.

# Bibliography


[1] Franz J. Brandenburg, Michael Jünger, and Petra Mutzel. Algorithmen zum automatischen Zeichnen von Graphen. *Research Report Number: MPI-I-97-1-007*, Informatik-Spektrum 20(4):199–207, 1997.

[2] Eric Clayberg and Dan Rubel. *Eclipse Plug-ins*. Addison Wesley, 2009.

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

[4] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998.

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

[6] Richard C. Gronback. *Eclipse Modeling Project: A Domain-Specific Language (DSL) Toolkit*. Addison Wesley, 2009.

[7] Erkki Mäkinen. How to draw a hypergraph. *International Journal of Computer Mathematics*, 34:177–185, 1990.

[8] Object Technology International, Inc. Eclipse Platform Technical Overview, 2003.

[9] Helen C. Purchase, Robert F. Cohen, and Murray James. Validating graph drawing aesthetics. In F. Brandenburg, editor, *Proceedings of Graph Drawing Symposium*, volume 1027 of *LNCS*, pages 435–446. Springer Verlag, 1996.

[10] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF - Eclipse Modeling Framework*. Addison Wesley, 2009.

[11] Sven Efftinge. Model2Model transformation with Xtend, 2006. `http://blog.efftinge.de/2006/04/model2model-transformation-with-xtend_15.html`.