

Graph Layout Support for Model-Driven Engineering

Dipl.-Inf. Miro Spönemann

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2014

Kiel Computer Science Series (KCSS) 2015/2 v1.0 dated 2015-3-13

ISSN 2193-6781 (print version)

ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via <https://www.informatik.uni-kiel.de/kcss>

Published by the Department of Computer Science, Kiel University

Real-Time and Embedded Systems Group

Please cite as:

- ▷ Miro Spönemann. *Graph layout support for model-driven engineering*. Number 2015/2 in Kiel Computer Science Series. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel, 2015.

```
@book{Spoonemann15,  
  author   = {Miro Sp{"o"}nemann},  
  title    = {Graph layout support for model-driven engineering},  
  publisher = {Department of Computer Science},  
  year     = {2015},  
  isbn     = {9783734772689},  
  series   = {Kiel Computer Science Series},  
  number   = {2015/2},  
  note     = {Dissertation, Faculty of Engineering,  
             Christian-Albrechts-Universit{"at"} zu Kiel}  
}
```

© 2015 by Miro Spönemann

Herstellung und Verlag:

BoD – Books on Demand, Norderstedt

ISBN 978-3-7347-7268-9

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit <http://www.informatik.uni-kiel.de/kcss> for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter: Prof. Dr. Reinhard von Hanxleden
Institut für Informatik
Christian-Albrechts-Universität zu Kiel
2. Gutachter: Prof. Dr. Petra Mutzel
Fakultät für Informatik
Technische Universität Dortmund
3. Gutachter: Prof. Dr. Wilhelm Hasselbring
Institut für Informatik
Christian-Albrechts-Universität zu Kiel

Datum der mündlichen Prüfung: 9. Februar 2015

Zusammenfassung

Wie bereits von Fuhrmann gezeigt [Fuh11], kann die Produktivität modellgetriebener Softwareentwicklung durch zahlreiche Konzepte zur Verbesserung der praktischen Handhabung von Modellen erhöht werden. Dabei ist das automatische Layout graphenbasierter Modelle ein zentraler Schlüssel. Allerdings gibt es einen bemerkenswerten Kontrast zwischen der Fülle an Forschungsergebnissen im Bereich des Graphen-Layout und dem aktuellen Stand graphischer Modellierungswerkzeuge, bei denen nur ein kleiner Teil dieser Ergebnisse übernommen wird. Das Ziel dieser Arbeit ist diese Lücke auf drei separaten Ebenen zu überbrücken: spezialisierte Layout-Algorithmen, Verwaltung von Konfigurationen und Software-Infrastruktur.

Im Bezug auf Layout-Algorithmen liegt der Schwerpunkt auf dem *Layer*-basierten Ansatz. Wir untersuchen dessen Erweiterung zur Unterstützung von *Ports* und *Hyperkanten*, was wesentliche Bestandteile bestimmter Arten von Graphen sind, z. B. Datenflussmodelle. Der Hauptbeitrag ist die Einbeziehung von Bedingungen für die Positionierung von Ports, vor allem während der Kreuzungsminimierung und der Kantenführungsphase. Hyperkanten werden durch normale Kanten repräsentiert, was deren Verarbeitung vereinfacht aber Ungenauigkeiten beim Zählen von Kreuzungen verursacht. Als letzte Erweiterung betrachten wir einen *Sketch*-basierten Ansatz für die einfache Integration von Nutzerinteraktivität.

Ein *abstraktes* Layout ist die Auswahl eines Layout-Algorithmus zusammen mit einer Abbildung seiner Parameter auf konkrete Werte, während ein *konkretes* Layout Positionsdaten beschreibt, die von einem Algorithmus berechnet wurden. Wir diskutieren ein neues Metamodell, mit dem sowohl die Struktur als auch das abstrakte sowie das konkrete Layout eines Graphen spezifiziert werden kann. Dies bildet eine Grundlage für die effiziente Verwaltung von Layout-Konfigurationen. Zudem untersuchen wir einen evolutionären Algorithmus für die Suche im Lösungsraum abstrakter Layouts, wobei zur Bewertung von Lösungen Ästhetikkriterien ausgewertet werden.

Die in dieser Arbeit entwickelte Software-Infrastruktur hat als Ziel, beliebige Graphen-basierte Diagramme (*front-ends*) mit beliebigen Layout-Algorithmen (*back-ends*) zu verbinden. Die größte Herausforderung dabei ist das Finden geeigneter Abstraktionen, die eine solche Allgemeingültigkeit erlauben und gleichzeitig die Komplexität so niedrig wie möglich halten. Wir betrachten eine mögliche Realisierung, die auf Eclipse basiert, eine von vielen Modellierungswerkzeugen verwendete Plattform. Eine Web-basierte Umfrage wurde unter Nutzern der Layout-Infrastruktur durchgeführt, um zu untersuchen inwieweit die gesteckten Ziele erfüllt worden sind. Die allgemeine Resonanz zu dieser Umfrage ist sehr positiv.

Abstract

As shown previously by Fuhrmann [Fuh11], there are several concepts for increasing the productivity of model-driven engineering by improving the practical handling of models. The automatic layout of graph-based models is a key enabler in this context. However, there is a striking contrast between the abundance of research results in the field of graph layout methods and the current state of graphical modeling tools, where only a tiny fraction of these results are ever adopted. This thesis aims to bridge this gap on three separate levels: specialized layout algorithms, configuration management, and software infrastructure.

Regarding layout algorithms, here we focus on the *layer-based* approach. We examine its extension to include *ports* and *hyperedges*, which are essential features of certain kinds of graphs, e.g. data flow models. The main contribution is the handling of constraints on the positioning of ports, which is done mainly in the crossing minimization and edge routing phases. Hyperedges are represented with normal edges, simplifying their handling but introducing inaccuracies for counting crossings. A final extension discussed here is a *sketch-driven* approach for simple integration of user interactivity.

An *abstract* layout is the selection of a layout algorithm with a mapping of its parameters to specific values. We discuss a new meta model allowing to specify the structure of a graph as well as its abstract layout and its *concrete* layout, i. e. positioning data computed by the layout algorithm. This forms a basis for efficient management of layout configurations. Furthermore, we investigate an evolutionary algorithm for searching the solution space of abstract layouts, taking readability criteria into account for evaluating solutions.

The software infrastructure developed here targets the connection of arbitrary diagram viewers (front-ends) with arbitrary graph layout algorithms (back-ends). The main challenge is to find suitable abstractions that

allow such generality and at the same time keep the complexity as low as possible. We discuss a possible realization based on the Eclipse platform, which is used by several modeling tools, e. g. the Graphical Modeling Framework (GMF). A web-based survey has been conducted among users of the layout infrastructure in order to evaluate to what extent the stated goals have been met. The overall feedback collected from this survey is very positive.

Contents

1	Introduction	1
1.1	Contributions of This Thesis	4
1.1.1	Publications	6
1.2	Modeling Languages	11
1.2.1	Data Flow Languages	12
1.2.2	Control Flow Languages	12
1.2.3	Static Structure Languages	15
1.3	Definitions	16
1.3.1	Graphs	16
1.3.2	Graph Layout	18
1.3.3	Statistics	19
1.4	Graph Layout Methods	22
1.4.1	Planarization-Based Layout	22
1.4.2	Force-Based Layout	27
1.4.3	Tree and Circular Layout	28
1.5	The KIELER Project	29
1.6	Related Work	30
I	Layout Algorithms	37
2	The Layer-Based Approach	39
2.1	Base Algorithms	40
2.1.1	Elimination of Cycles	40
2.1.2	Layer Assignment	41
2.1.3	Crossing Minimization	44
2.1.4	Node Placement	46
2.1.5	Edge Routing	47
2.2	Port Constraints	48
2.2.1	Handling Constraint Levels	49
2.2.2	Crossing Minimization	52

Contents

2.2.3	Edge Routing	59
2.2.4	Evaluation	66
2.3	Hyperedges	75
2.3.1	General Representation	75
2.3.2	Junction Points	79
2.3.3	Counting Crossings	81
2.3.4	Evaluation	87
2.4	Interactive Layout	92
2.4.1	Layer-Based Sketch-Driven Layout	93
2.4.2	Evaluation	97
3	Meta Layout	103
3.1	A Generic Layout Interface	104
3.1.1	The KGraph Meta Model	106
3.1.2	Layout Configuration	110
3.1.3	Meta Data of Layout Algorithms	114
3.2	Optimal Layout Configuration	118
3.2.1	Genotypes and Phenotypes	119
3.2.2	Evolutionary Process	127
3.2.3	Evaluation	132
3.2.4	An Alternative Method: Successive Optimization . . .	138
II	From Theory to Practice	141
4	Integration in Eclipse	143
4.1	Programming Interface	145
4.1.1	Meta Data	146
4.1.2	Layout Configuration	150
4.1.3	Performance	154
4.2	User Interface	156
4.2.1	Layout View	156
4.2.2	Preference Page	158
4.3	Connection to Diagram Viewers	160
4.3.1	GMF Editors	161

4.3.2	Graphiti Editors	166
4.3.3	Transient Views	169
4.4	Connection to Algorithms	172
4.4.1	The KIELER Layouters (K Lay)	174
4.4.2	Graphviz	178
4.4.3	OGDF	180
4.5	Tools for Algorithm Developers	181
4.5.1	Graph Editor	182
4.5.2	Textual Format	183
4.5.3	Graph Analysis	186
4.5.4	Graph Formats	188
4.5.5	Additional Tools	190
5	Integration in Other Applications	193
5.1	Class Library	193
5.2	Web Service	197
6	Survey	201
6.1	Detailed Results	202
6.1.1	Application Requirements	203
6.1.2	Quality Evaluation	206
6.2	Discussion	227
7	Conclusion	233
7.1	Summary of Results	233
7.2	Lessons Learned	237
7.3	Future Work	240
	Appendix	245
A	Sketch-Driven Layout Experiment	246
B	Evolutionary Meta Layout Experiment	249
C	Survey Questionnaire	255
	Bibliography	265

Acronyms

ASCET	Advanced Simulation and Control Engineering Tool (ETAS)
API	application programming interface
AST	abstract syntax tree
AUTOSAR	Automotive Open System Architecture http://www.autosar.org/
AWT	Abstract Window Toolkit http://docs.oracle.com/javase/
BPMN	Business Process Model and Notation (OMG) http://www.omg.org/spec/BPMN/
CAU	Christian-Albrechts-Universität zu Kiel http://www.uni-kiel.de/
CPU	central processing unit
CSV	comma-separated values
DAG	directed acyclic graph
DSL	domain-specific language
ECU	electronic control unit
EMF	Eclipse Modeling Framework http://projects.eclipse.org/projects/modeling.emf
EMP	Eclipse Modeling Project http://projects.eclipse.org/projects/modeling
ETAS	Engineering Tools, Application and Services http://www.etas.com/

GEF	Graphical Editing Framework
GLMM	graph layout meta model
GMF	Graphical Modeling Framework http://projects.eclipse.org/projects/modeling.gmf.gmf-runtime
GML	Graph Modelling Language [Him97]
GPL	GNU General Public License https://www.gnu.org/copyleft/gpl.html
GraphML	Graph Markup Language [BELP13]
GSoC	Google Summer of Code http://developers.google.com/open-source/soc/
GWT	GWT Web Toolkit http://www.gwtproject.org/
HTTP	Hypertext Transfer Protocol
IBM	International Business Machines http://www.ibm.com/
IDE	integrated development environment
ILP	integer linear program
JAR	Java Archive
JAX-WS	Java API for XML Web Services (Oracle) https://jax-ws.java.net
JDT	Java Development Tools http://projects.eclipse.org/projects/eclipse.jdt
JETI	Electronic Tool Integration Platform [MNS05] http://jeti.cs.uni-dortmund.de
JSON	JavaScript Object Notation http://www.json.com/

Acronyms

JVM	Java Virtual Machine
KAOM	KIELER Actor Oriented Modeling
KCSS	Kiel Computer Science Series http://www.informatik.uni-kiel.de/kcss
KEG	KIELER Editor for Graphs
KIEL	Kiel Integrated Environment for Layout
KIELER	Kiel Integrated Environment for Layout Eclipse Rich Client http://www.informatik.uni-kiel.de/rtsys/kieler/
KIML	KIELER Infrastructure for Meta Layout
KLay	KIELER Layouters
KLighD	KIELER Lightweight Diagrams
KLoDD	KIELER Layout of Dataflow Diagrams
KWebS	KIELER Web Services
MDE	model-driven engineering
MDSD	model-driven software development
MDV	model-driven visualization [BSLF06]
MoML	Modeling Markup Language (Ptolemy)
MVC	model-view-controller
NP	nondeterministic polynomial (an execution time complexity class)
OGDF	Open Graph Drawing Framework http://www.ogdf.net/
OGML	Open Graph Markup Language (TU Dortmund)

OMG	Object Management Group http://www.omg.org/
OS	Operating System
OSGi	Open Services Gateway Initiative (OSGi Alliance)
RCA	rich client application
RCP	rich client platform
ROOM	Real-Time Object-Oriented Modeling [SGW94]
SCADE	Safety Critical Application Development Environment (Esterel Technologies SA) http://www.esterel-technologies.com/products/
SCT	Statechart Tools (Yakindu)
SOAP	Simple Object Access Protocol
SUD	system under development
SVG	Scalable Vector Graphics (W3C)
SWT	Standard Widget Toolkit http://www.eclipse.org/swt/
UI	user interface
UML	Unified Modeling Language (OMG)
URL	uniform resource locator
W3C	World Wide Web Consortium http://www.w3.org/
WSDL	Web Services Description Language
XMI	XML Metadata Interchange (OMG)
XML	Extensible Markup Language (OMG)

Introduction

There is a striking contrast between the abundance of research results in the field of graph layout methods and the current state of graphical modeling tools, where only a tiny fraction of these results are adopted. This thesis aims to bridge this gap on three separate levels: specialized layout algorithms, configuration management, and software infrastructure. The goal is to increase the productivity of software engineering processes by improving the practical handling of models.

A *model* is an artifact that represents other artifacts with a purpose [Tha13]. In the field of computer science, usually this purpose is to facilitate the understanding of systems through abstraction. Many models are made with no formal basis, relying on the intuition and the knowledge background of their users. The *model-driven engineering* (MDE) approach¹ is based on formal models, leading to the notion of *modeling languages* [Sch06, SV06, FR07]. The concepts behind modeling languages are very similar to those of programming languages: they have an *abstract syntax* that determines the structure, a *concrete syntax* defining the actual representation, and a formal or informal *semantics* defining the meaning. There are *general-purpose* languages, e. g. following open standards such as the Unified Modeling Language (UML), or *domain-specific* languages (DSLs) designed for particular applications. A concrete syntax can be either one-dimensional, i. e. text, or two-dimensional (sometimes three-dimensional), in which case it is called a *graphical* representation. The main strength of MDE is the ability to automatically *transform* models from one representation to another, allowing to build models of a high abstraction level and then to generate models of a lower abstraction level. This is analogous to compilers, which

¹an interchangeable term for MDE is *model-driven software development* (MDSD)

1. Introduction

generate low-level machine code from high-level programming languages.

A major advantage of graphical representations is that they allow to directly visualize relationships, whereas in text indirections through name references are often necessary. For this reason it is not surprising that the majority of graphical languages can be mathematically abstracted with *graphs*, i. e. collections of objects (*nodes*) and their relationships (*edges*). The concrete syntax of a textual language often prescribes the order of elements though its grammar. The elements of a graph, in contrast, can be arranged more or less arbitrarily on the two-dimensional plane. Some application domains limit this freedom in order to ensure a consistent appearance of model instances, e. g. requiring edges to point from left to right, but even then the problem of finding a suitable arrangement is much more complex compared to text.

A good graph layout must be *readable*, meaning that it must support the persons working with it in quickly understanding the underlying model. Readability depends on a variety of factors, named *aesthetic criteria* [Pur97, WPCM02, BRSG07]. From the beginning of the 1980s, a large number of research groups have sought for methods to optimize these criteria, a research field known as *graph drawing* [DETT99, KW01, Tam13]. Many different algorithmic approaches have been developed through these efforts, with different priorities on aesthetic criteria and application constraints. Among these, for instance, are methods for minimizing the number of edge crossings [Wei01], maintaining layout stability for dynamically changing graphs [Bra01], and considering special requirements of widespread notations such as UML class diagrams [See97, EGK⁺04a].

Automatic graph layout has the potential to boost the productivity of model-driven engineering as a key enabler of efficient model creation and exploration concepts [FvH10a, FvH10b]. A layout algorithm can relieve users from the time-consuming task of manually arranging graphical models, allowing them to focus on the semantic aspects of their system under development. Furthermore, graphical representations can be generated fully automatically [SSvH12a, SSvH13], e. g. from the results of model transformations or from textual notations. This includes the dynamic creation of optimized views for the efficient browsing of large collections of existing models [FvHK⁺14].

Looking at the modeling tools that have been in use during the past decade, however, one realizes that the state of the practice is quite far from fully exploiting the potential of automatic layout. Some tools offer only assistance for the horizontal or vertical alignment of selected nodes, e. g. Simulink² or SCADE³, and others include layout algorithms of rather low quality, e. g. the Eclipse Graphical Modeling Framework (GMF)⁴ or Ptolemy⁵ (a better algorithm has been built into the Ptolemy editor as a result of this thesis, see Section 5.1). I identified several possible reasons for this discrepancy between the scientific work and its realization in MDE environments.

- P1 Users are skeptical about the quality of automatically generated layouts. This is similar to how automatic code generation was seen with concerns in the early days of MDE [Sel03].
- P2 Domain-specific layout constraints are not adequately addressed by the established methods. In particular, constraints on the positioning of *ports* (connection points of edges) are hardly considered.
- P3 Graph layout algorithms are complex and thus their implementation requires considerable effort. For the same reason, commercial graph layout libraries are quite expensive.
- P4 The integration of graph layout libraries written in C or C++, e. g. Graphviz [GN00] or OGDF [CGJ⁺13], into other platforms such as Eclipse is an intricate task.
- P5 Users are quickly overwhelmed by the multitude of graph layout methods and their configuration parameters. The usage of many parameters requires a detailed understanding of the underlying methods.

These problems are even more evident in the context of domain-specific languages, where the focus is often on light-weight modeling approaches

²<http://www.mathworks.com/>

³<http://www.esterel-technologies.com/>

⁴<http://www.eclipse.org/modeling/gmf/>

⁵<http://ptolemy.eecs.berkeley.edu>

1. Introduction

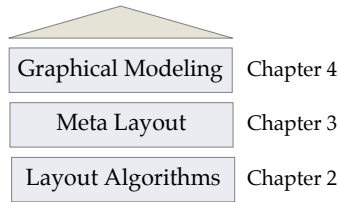


Figure 1.1. The approach of this thesis is structured in three layers: modeling applications with graphical viewers or editors (*front-ends*), a collection of specialized layout algorithms (*back-ends*), and a *meta layout* framework for controlling the algorithms.

with frameworks such as Xtext.⁶ Domain-specific notations require domain-specific layouts, i. e. a suitable layout algorithm must be found and configured. This task, however, is not well supported by today's modeling platforms.

1.1 Contributions of This Thesis

I propose a threefold approach to support the usage of automatic graph layout in modeling applications (see Figure 1.1).

1. Specialized layout algorithms address frequently occurring requirements in order to meet the needs of typical modeling applications. These algorithms are the *back-ends* of the layout infrastructure.
2. Layout algorithms are selected and configured in a process named *meta layout*. Configurations can be specified by tool developers, adapted by tool users, or determined automatically.
3. Viewers or editors for graphical model representations, the *front-ends* in the terminology used here, are connected to the layout infrastructure. These connections must be as generic as possible in order to allow simple and light-weight integrations.

⁶<http://www.eclipse.org/Xtext/>

1.1. Contributions of This Thesis

The goal is to provide a large number of high-quality back-ends for use in a large number of front-ends. I believe that Problem P1, i. e. the widespread skepticism on the usefulness of automatic layout, can only be properly addressed if the overall quality of the layout infrastructure is convincing. This premise has to be considered in all three layers of the proposed approach. An outline of the concrete contributions of this thesis is given in the following.

Layer-based layout (Chapter 2). The *layer-based* graph layout approach is extended in order to support some domain-specific constraints (Problem P2). The main extension concerns *port constraints*, which are necessary for many modeling languages (see Section 1.2). A general scheme for handling different levels of port positioning constraints is discussed, and adaptations for crossing minimization and edge routing are presented. The methods are evaluated with respect to different possible solutions and the execution time of the whole algorithm. Another extension addresses the handling of *hyperedges*, for which some practical approaches are given. The problem of predicting the number of hyperedge crossings during crossing minimization is discussed, for which two heuristics are introduced and compared. Finally, a *sketch-driven* layout approach is presented, allowing users to influence the result of the layer-based layout algorithm by giving hints on the order of elements.

Meta layout (Chapter 3). The concept of connecting multiple front-ends to multiple back-ends requires an interface that is capable of passing all necessary information. This is realized with a meta model named *KGraph*, which includes the graph structure, the *abstract layout* (configuration of parameters of layout algorithms), and the *concrete layout* (positioning of graph elements). The problem of finding suitable configurations without requiring users to understand all parameters (Problem P5) is handled with a general scheme of combined *layout configurators*, allowing to specify parameter values for particular applications. Furthermore, a heuristic for finding configurations that match the user's expectation is presented. The heuristic is based on an evolutionary algorithm and is evaluated with a user experiment.

1. Introduction

Integration in Eclipse (Chapter 4). Eclipse is the main application platform into which the results of this thesis have been integrated. The choice of this platform is quite natural, since it offers a multitude of frameworks for working with models from different perspectives, and many industrial as well as academic applications use Eclipse for graphical modeling. The integration includes programming interfaces, user interfaces, connections to different editing and viewing frameworks, connections to libraries of layout algorithms, and additional tools for the development of layout algorithms. By integrating a variety of very different layout algorithms, tool developers do not need to implement their own algorithms (Problem P3), but can reuse existing ones. In particular, the algorithms provided by Graphviz [GN00] and OGDF [CGJ⁺13] are made available through standard I/O interfaces, thus decoupling these C/C++ libraries from the Java-based Eclipse process (Problem P4).

Other applications (Chapter 5). The interfaces discussed in Chapter 4 are designed such that they can also be employed in Java applications that are not built on Eclipse. Furthermore, they are offered through a web service, making them available for an even wider range of applications.

Survey (Chapter 6). A survey was conducted in order to assess the perceived quality of the concepts and implementations presented here, and to gather some interesting information on the requirements of actual applications. The results of the survey are very positive, revealing a high satisfaction among users of the layout infrastructure.

1.1.1 Publications

The following is a list of publications I have been involved in, each with a brief summary and an explanation of how they relate to this thesis. The essential publications are given first, where at least a half of the paper is based on my contributions.

Essential publications.

[SFvHM10] M. Spönemann, H. Fuhrmann, R. von Hanxleden, and P. Mutzel. Port constraints in hierarchical layout of data flow diagrams. *GD 2009*, LNCS vol. 5849, Springer, 2010.

This paper reports the results of my diploma thesis [Spö09]. It introduces the problem of drawing graphs with port constraints, with data flow diagrams as the main application, and reports extensions of the layer-based drawing approach. This includes parts of the crossing minimization adaptations discussed here in Section 2.2.2, but here they are put in a more general context and compared with alternatives.

[SSM⁺13] M. Spönemann, C. D. Schulze, C. Motika, C. Schneider, and R. von Hanxleden. KIELER: Building on automatic layout for pragmatics-aware modeling. *VL/HCC 2013*, San Jose, USA, 2013.

The concepts for flexible configuration of layout algorithms (Section 3.1) are summarized in this two-page showpiece, which was accompanied by a demonstration and a poster. A special focus is given to the configuration of KLightD views (see Section 4.3.3), where the effect of layout parameters can be visualized immediately.

[SSvH14] C. D. Schulze, M. Spönemann, and R. von Hanxleden. Drawing layered graphs with port constraints. *Journal of Visual Languages and Computing* 25(2), 2014.

The methods for layer-based layout with port constraints, first proposed in previous conference papers [SFvHM10, KSSvH12], are discussed with more detail in this journal publication. The content is largely equivalent with Section 2.2 of this thesis. Furthermore, the basic structure of *KLay Layered* is described, that is the implementation of the layer-based algorithm used here (Section 4.4.1).

[SSRvH14a] M. Spönemann, C. D. Schulze, U. Rüegg, and R. von Hanxleden. Counting crossings for layered hypergraphs. *DIAGRAMS 2014*, LNAI vol. 8578, Springer, 2014.

Typical heuristics for crossing minimization in the layer-based approach require to count the number of edge crossings obtained after each

1. Introduction

intermediate step. With hyperedges, however, the actual number of crossings cannot be reliably determined at this rather early stage of the algorithm. Methods for predicting the number of crossings have been developed, clearly outperforming the standard counting method that does not consider hyperedges. The discussion of these counting methods and their evaluation is found in Section 2.3.

[SDvH14a] M. Spönemann, B. Duderstadt, and R. von Hanxleden. Evolutionary meta layout of graphs. *DIAGRAMS 2014*, LNAI vol. 8578, Springer, 2014.

An approach for the automatic selection and configuration of layout algorithms is described, here discussed in Section 3.2. An evolutionary algorithm is used to repeatedly modify a set of initial random configurations. Each configuration is evaluated automatically by executing the respective layout algorithm and analyzing the resulting concrete layout with a set of metrics that determine to which degree certain aesthetic criteria are satisfied.

Further publications.

[FSMvH10] H. Fuhrmann, M. Spönemann, M. Matzen, and R. von Hanxleden. Automatic layout and structure-based editing of UML diagrams. *Workshop on Model Based Engineering for Embedded Systems Design*, Dresden, Germany, 2010.

An early state of the Eclipse integration of the layout infrastructure is reported, which includes some of the concepts presented in Chapter 4. The integration is applied to the Papyrus UML tool in order to arrange activity diagrams, state machines, and other diagram types. Based on the layout infrastructure, a *structure-based* editing approach is presented, which uses transformations for modifying models.

[CGM⁺11] M. Chimani, C. Gutwenger, P. Mutzel, M. Spönemann, and H.-M. Wong. Crossing minimization and layouts of directed hypergraphs with port constraints. *GD 2010*, LNCS vol. 6502, Springer, 2011.

This is an extension of the *upward planarization* approach [CGMW10, CGMW11] to consider port constraints and hyperedges. The motivation was to develop alternative methods for the layout of data flow diagrams,

1.1. Contributions of This Thesis

but the algorithms proposed in this paper have not been implemented yet. Planarization-based drawing is not covered in detail here; a brief introduction is given in Section 1.4.1.

[BBE⁺11] C. Bachmaier, F.J. Brandenburg, P. Effinger, C. Gutwenger, J. Katajainen, K. Klein, M. Spönemann, M. Stegmaier, and M. Wybrow. The Open Graph Archive: A community-driven effort. *GD 2011*, LNCS vol. 7034, Springer, 2012.

During the Dagstuhl Seminar 11191, “Graph Drawing with Algorithm Engineering Methods” [DKKM11], a working group on archives of example graphs was formed. The goal was to initiate a community effort to maintain a web-based archive where graphs used for experiments can be shared. The proposal was communicated with a poster at the Graph Drawing Symposium. The archive server uses the web service presented in Section 5.2 for translating between different graph formats.

[KSSvH12] L. K. Klauske, C. D. Schulze, M. Spönemann, and R. von Hanxleden. Improved layout for data flow diagrams with port constraints. *DIAGRAMS 2012*, LNAI vol. 7352, Springer, 2012.

The diploma thesis of Schulze [Sch11], conducted under my supervision, built on the previously published work on port constraints [SFvHM10] and improved on several topics, including the routing of edges where the ports are constrained to certain sides of the nodes. These results have been published in collaboration with Klauske, who has worked on a layout algorithm for Simulink [KD10, KD11], and are described here in Section 2.2.3.

[SSvH12a] C. Schneider, M. Spönemann, and R. von Hanxleden. Transient view generation in Eclipse. *Workshop on Academics Modeling with Eclipse*, Kgs. Lyngby, Denmark, 2012.

This paper introduces the KLightD framework for the light-weight synthesis of transient graphical views (see Section 4.3.3). The framework uses the KGraph meta model, which is also the basic interface between layout algorithms and modeling applications, and relies on the layout configuration concepts discussed in Section 3.1.

1. Introduction

[FvHK⁺14] P. Frey, R. von Hanxleden, C. Krüger, U. Rüegg, C. Schneider, and M. Spönemann. Efficient exploration of complex data flow models. *Modellierung 2014*, Vienna, Austria, 2014.

The layout infrastructure and the layer-based layout algorithm, which are both implemented in Eclipse (Chapter 4), have been employed in an industrial project that targets the visualization of large hierarchical data flow models. This paper describes the requirements of that project and compares two visualization approaches, one using GMF and one using KLightD (see Section 4.3.3).

[GvHM⁺14] C. Gutwenger, R. von Hanxleden, P. Mutzel, U. Rüegg, and M. Spönemann. Examining the compactness of automatic layout algorithms for practical diagrams. *Workshop on Graph Visualization in Practice*, Melbourne, Australia, 2014.

The layout algorithm extensions discussed here address mainly the number of edge crossings and bends, but in the context of industrial applications it could be observed that the amount of whitespace and the aspect ratio can be much stronger factors limiting the readability of a diagram. Some experimental results supporting this observation as well as ideas for improving the situation are presented in this workshop paper.

Advised theses.

- ▷ Matthias Schmeling. *Effective visualization of IEC 61499 Function Blocks*, April 2010 (Diploma Thesis).
- ▷ Martin Rieß. *A graph editor for algorithm engineering*, September 2010 (Bachelor Thesis).
- ▷ Christian Kutschmar. *Planarisierung von Hypergraphen*, September 2010 (Bachelor Thesis).
- ▷ Philipp Döhning. *Algorithmen zur Layerzuweisung*, September 2010 (Bachelor Thesis).
- ▷ Ole Claußen. *Implementing an algorithm for orthogonal graph layout*, September 2010 (Bachelor Thesis).

1.2. Modeling Languages

- ▷ Christian Schneider. *On integrating graphical and textual modeling*, February 2011 (Diploma Thesis).
- ▷ Hauke Wree. *Ein Gleisplaneditor basierend auf Graphiti*, March 2011 (Bachelor Thesis).
- ▷ Björn Duderstadt. *Evolutionary meta layout for KIELER*, May 2011 (Student Research Project).
- ▷ Christoph Daniel Schulze. *Optimizing automatic layout for data flow diagrams*, July 2011 (Diploma Thesis).
- ▷ Stephan Wersig. *Ein Web Service für das automatische Layout von Graphen*, October 2011 (Diploma Thesis).
- ▷ Insa Fuhrmann. *Layout of compound graphs*, February 2012 (Diploma Thesis).
- ▷ Sven Gundlach. *Synthese von Datenflussdiagrammen aus annotierten C-Programmen*, March 2012 (Bachelor Thesis).
- ▷ Paul Klose. *A generic framework for the topology-shape-metrics based layout*, October 2012 (Master Thesis).
- ▷ Heiko Wißmann. *Graphische Visualisierung von Java-Variablen zur Laufzeit*, March 2013 (Bachelor Thesis).

1.2 Modeling Languages

This section provides an overview of typical modeling languages used in the context of MDE. Although the graphical notation of most languages can be abstracted to graphs, they are quite different regarding what is represented by the basic graph elements, i. e. nodes and edges. Three important groups of languages can be found based on this criterion: *data flow* languages, *control flow* languages, and *static structure* languages. Here the focus is on the special requirements of these languages with regard to graph layout.

1. Introduction

1.2.1 Data Flow Languages

Data flow means that the main focus is on the processing and communication of data, hence nodes represent processing units (often named *actors* or *operators*) and edges represent streams of data. Processing units have predefined interfaces that are specified through *ports*. Each edge incident to a given node must be assigned to a port of that node. Ports where only incoming or only outgoing edges may be connected are often called *input* or *output* ports, respectively. For instance, a subtraction operator can be realized with two input ports and one output port. The correct assignment of incoming edges to the input ports is crucial, since the subtraction operation is not commutative. Furthermore, many data flow languages allow *hyperedges*, i. e. connections between more than two ports. In most cases a hyperedge may have one source and one or more targets, meaning that the data streamed through one output port are transported to multiple input ports.

The edges in data flow diagrams are drawn orthogonally with a left-to-right orientation, except for *feedback* edges forming directed cycles. The layout of ports and hyperedges is covered in Chapter 2.

There are numerous tools for creating data flow models, e. g. Simulink⁷ (Figure 1.2(a)), SCADE⁸ (Figure 1.2(b)), ASCET⁹ (Figure 4.18 on p. 172), or Ptolemy [EJL⁺03] (Figure 2.16 on p. 68). Often such tools are used to generate C code from a set of data flow models, which is then compiled and deployed on an embedded microcontroller.

1.2.2 Control Flow Languages

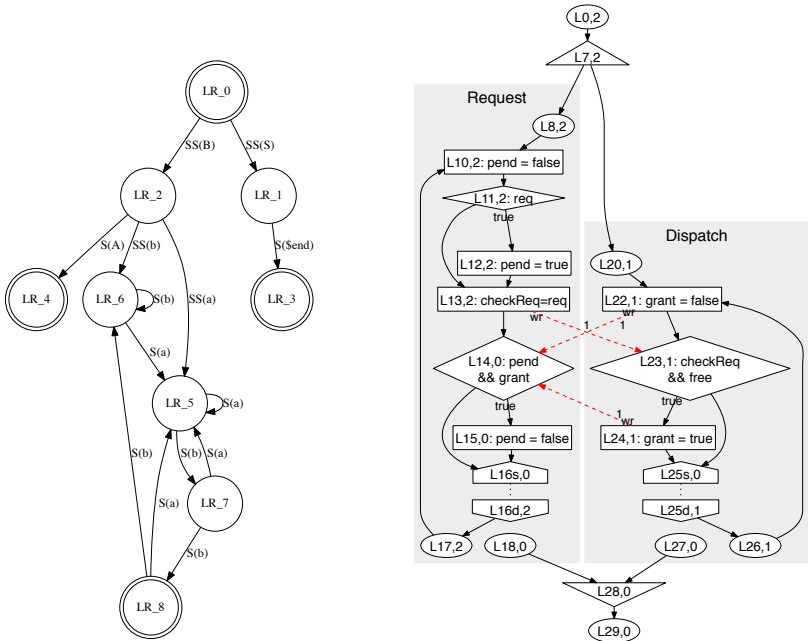
In control flow languages, nodes represent program states and edges represent transitions between states. The simplest form of control flow is a state machine (Figure 1.3(a)), but in many cases *statecharts* are used instead, i. e. state machines extended with hierarchy and concurrency [Har87]. The modeling of statecharts is supported by many tools and is included in the

⁷<http://www.mathworks.com/>

⁸<http://www.esterel-technologies.com/>

⁹<http://www.etas.com/>

1. Introduction



(a) A state machine drawn with Graphviz (<http://www.graphviz.org/>)

(b) A control flow graph [vHMA⁺13]

Figure 1.3. Different notations for specifying control flow: (a) state machines and (b) control flow graphs.

UML standard. Examples are shown in Figure 4.10 (p. 163), Figure 4.13 (p. 166), and Figure 4.19(a) (p. 173). Some data flow modeling tools, e.g. Simulink and SCADE (see above), offer statecharts as an additional notation complementing the data flow paradigm.

There are several notations for specifying control flow, in addition to state-machine-based notations. *Control flow graphs* describe the possible transitions between executable statements of programs and are often generated during the compilation process of programming languages or models. Examples are shown in Figure 1.3(b) and Figure 4.19(b) (p. 173). *Activity*

diagrams are conceptually similar, but do not refer to program execution, but rather to workflows, e. g. as applied in businesses. They are also part of the UML standard. A special form of activity diagrams are *business process models*, which have specialized standards adapted to the requirements of enterprise applications, e. g. *Business Process Model and Notation* (BPMN).¹⁰

Many graphical representations of control flow languages are drawn with curved edges and with a main edge orientation from left to right or from top to bottom, but these are not hard requirements. In most cases, the standard layer-based graph drawing approach fits very well to these representations.

1.2.3 Static Structure Languages

Both data flow and control flow are paradigms for modeling the *behavior* of systems. There are other languages that address the *structure* of systems independently of their exact behavior, hence they are referred to as *static* modeling languages. For instance, in the context of object-oriented programming, UML class diagrams visualize the relationships between a set of classes of a software implementation. There are three types of relationships: *associations* for direct references between classes, *dependencies* for weaker references (i. e. one class uses another in some form), and *generalizations* expressing class inheritance. Chapter 4 contains some diagrams of Java classes implementing parts of the Eclipse integration of the concepts described in this thesis (Figure 4.2 on p. 148, Figure 4.3 on p. 149, and Figure 4.4 on p. 151). Meta models such as those created with the *Ecore* format of the Eclipse Modeling Framework (EMF) are also visualized with class diagrams, e. g. Figure 3.2 (p. 107).

While class diagrams focus on specific details of software implementations, *component* or *architecture* diagrams describe more abstract views on the structure of systems. An important example from the automotive industry is the AUTOSAR standard,¹¹ which formally describes the interaction of software components. However, architecture descriptions are often used informally, e. g. in Figure 4.1 (p. 144).

¹⁰<http://www.omg.org/spec/BPMN/>

¹¹<http://www.autosar.org/>

1. Introduction

For class diagrams it is important to handle the relationship types differently in the graphical layout: generalizations should be drawn from bottom to top, while other edges can be oriented arbitrarily. Therefore a class diagram layout algorithm must support mixed graphs with both directed and undirected edges. For component diagrams, the layout requirements vary depending on the kind of representation and the application domain. In some cases users may expect the layout to reflect the symmetries of the architecture. Furthermore, many component descriptions use ports to specify connections, hence similar constraints as for data flow diagrams apply. For instance, AUTOSAR defines a number of different port types to represent the interfaces of components.

1.3 Definitions

This section introduces the basic terminology used throughout this thesis. Given any set M , we denote its power set by 2^M . M^k is the set of k -tuples of elements of M , with $k \in \mathbb{N}$. M^* is the set of finite sequences of elements of M , i. e. $M^* = \bigcup_{k \in \mathbb{N}} M^k$.

1.3.1 Graphs

Definition 1.1 (Graph). A *directed graph* is a pair (V, E) where $E \subseteq V^2$, i. e. edges are ordered pairs. An *undirected graph* is a pair (V, E) where $E \subseteq \{e \subseteq V : |e| = 2\}$, i. e. edges are unordered pairs.

For the following definitions, let $G = (V, E)$ be a directed graph. Many of these definitions can be applied similarly to undirected graphs.

Definition 1.2 (Source node, target node, adjacent nodes, predecessor, successor, incoming and outgoing edge, degree, source, sink, self-loop). Given an edge $e = (v, w) \in E$, $v_s(e) = v$ is called the *source node* of e and $v_t(e) = w$ is called the *target node* of e . The nodes v and w are said to be *adjacent*. We call v a *predecessor* of w and w a *successor* of v . $E_i(v) = \{e \in E \mid v = v_t(e)\}$ is the set of *incoming edges* of a node v and $E_o(v) = \{e \in E \mid v = v_s(e)\}$ is the set of *outgoing edges* of v . The *degree* of v is $|E_i(v)| + |E_o(v)|$. A *source* is a

1.3. Definitions

node with no incoming edges and a *sink* is a node with no outgoing edges. An edge $(v, v) \in E$ is called a *self-loop*.

The above definitions of graphs do not permit multiple edges with the same source and the same target because mathematically they are equivalent. However, practical applications often require to distinguish such multiple occurrences. One way to express this is through the matrix representation of a graph.

Definition 1.3 (Adjacency matrix, multiedge). Let $V = \{1, \dots, n\}$ be a numbered node set. The *adjacency matrix* of (V, E) is an integer matrix $(a_{i,j})$ with n rows and n columns where $a_{i,j} > 0$ if $(i, j) \in E$ and $a_{i,j} = 0$ otherwise. An edge $(i, j) \in E$ with $a_{i,j} > 1$ is called a *multiedge*. Multiedges can also be interpreted as *weights* on edges.

Definition 1.4 (Path, cycle, connected graph, tree). Let $p = v_1, \dots, v_k$ be a sequence of nodes. We call p a *directed path* from v_1 to v_k if $(v_i, v_{i+1}) \in E$ for $1 \leq i < k$. We call p an *undirected path* if $(v_i, v_{i+1}) \in E$ or $(v_{i+1}, v_i) \in E$ for $1 \leq i < k$. If p is a path and $v_1 = v_k$, we call p a *cycle*, which can also be directed or undirected. A graph without cycles is called *acyclic*. G is called *connected* if for all $v, w \in V$ there is an undirected path between v and w , and we call the graph *strongly connected* if for all $v, w \in V$ there is a directed path from v to w . An *undirected tree* is an undirected graph that is acyclic and connected. A *directed tree* is a directed acyclic graph (V, E) that contains a *root* $r \in V$ such that $|E_i(r)| = 0$ and for all $v \in V \setminus \{r\} : |E_i(v)| = 1$. Sinks of a directed tree are called *leafs*.

Definition 1.5 (Subgraph, k -connected graph). A *subgraph* of G is a graph $G' = (V', E')$ with $V' \subseteq V$ and $E' \subseteq E$. Given a node set $A \subseteq V$, the *induced subgraph* of A is (A, E_A) , where $E_A = \{(v, w) \in E \mid v, w \in A\}$. G is called *k -connected* if for all $C \subseteq V$ with $|C| = k - 1$ the subgraph induced by $V \setminus C$ is connected. A 2-connected graph is also called *biconnected*.

Definition 1.6 (Topological numbering). Let $\alpha : V \rightarrow \mathbb{N}$ be a numbering of the nodes. We call α a *topological numbering* if for all $(v, w) \in E : \alpha(v) < \alpha(w)$. A topological numbering is called a *topological sorting* if $\{\alpha(v) \mid v \in V\} = \{1, \dots, |V|\}$.

1. Introduction

In some application domains, especially data flow languages (Section 1.2.1), the normal graph representations do not include sufficient information to compute correct layouts, but they must be extended by ports and hyperedges.

Definition 1.7 (Port-based graph). G is a *port-based* graph if each edge $e \in E$ can be assigned a *source port* $p_s(e)$ and a *target port* $p_t(e)$. The set of ports of a node $v \in V$ is denoted as $P(v)$; for all $(v, w) \in E$ we require $p_s((v, w)) \in P(v)$ and for all $(w, v) \in E$ we require $p_t((w, v)) \in P(v)$. For each port p in a port-based graph there is exactly one v such that $p \in P(v)$. The set of incident edges of a port $p \in P(v)$ is denoted as $E(p) = \{e \in E \mid p_s(e) = p \vee p_t(e) = p\}$. Furthermore, a *side* $s \in \{\text{north, east, south, west}\}$ can be assigned to each port, indicating on which side of the associated node the port shall be positioned.

Definition 1.8 (Hypergraph). A *directed hypergraph* is a pair (V, H) where $H \subseteq 2^V \times 2^V$. The elements $h = (S, T) \in H$ are called *hyperedges*, S is the set of *source nodes* of h , and T is the set of *target nodes* of h . In a *port-based* hypergraph we can additionally assign source ports $P_s(h)$ and target ports $P_t(h)$. Usually each port is constrained to at most one incident hyperedge.

Many modeling languages are inherently hierarchical. We need another extension of the graph abstraction in order to express this.

Definition 1.9 (Compound graph, parent, child). A *compound graph* is a triple (V, E, I) such that (V, E) is a graph and (V, I) is a directed tree, called the *inclusion tree*. Given a node v , all successors of v in the inclusion tree are called *children* of v and are denoted with $V_c(v)$, while the predecessor of v is called its *parent* and is denoted with $v_p(v)$ unless v is the root node, which has no parent. If for all $(v, w) \in E$ both v and w are leafs of the inclusion tree, the compound graph is called a *clustering*. Edges $(v, w) \in E$ where $v_p(v) \neq v_p(w)$ are called *cross-hierarchy* edges.

1.3.2 Graph Layout

There are many alternatives for formally describing the layout (a.k.a. drawing) of a graph. Here we give a very simple definition that is focused on

the task of layout algorithms: to assign positions to all graph elements. Let $G = (V, E)$ be a directed graph.

Definition 1.10 (Graph layout). A *layout* γ of G , also called *drawing* of G , consists of a node positioning $\gamma_V : V \rightarrow \mathbb{R}^2$ and an edge routing $\gamma_E : E \rightarrow (\mathbb{R}^2)^*$ such that the end points of each edge touch its connected nodes, i. e. for all $e = (v, w) \in E$ with a routing $\gamma_E(e) = a_1, \dots, a_k$ we require $a_1 = \gamma_V(v)$ and $a_k = \gamma_V(w)$. The points a_2, \dots, a_{k-1} are called the *bend points* of e . Let $a_i = (x_i, y_i)$ for $1 \leq i \leq k$. The edge routing of e is called *orthogonal* if for all $1 \leq i < k$: $x_i = x_{i+1}$ or $y_i = y_{i+1}$.

In most cases nodes are drawn with certain shapes instead of points, e. g. rectangles. In this case, the end points specified by the edge routing of a graph layout must touch the rectangular shapes of the corresponding nodes. For port-based graphs, on the other hand, the end points must follow the port positions.

Definition 1.11 (Port-based graph layout). Let G be a port-based graph and $P = \bigcup_{v \in V} P(v)$ be the set of all ports. A layout γ of G consists of a node positioning and an edge routing as defined above and additionally a port positioning $\gamma_P : P \rightarrow \mathbb{R}^2$, where port positions are regarded as relative to the corresponding node positions. For each $e = (v, w) \in E$ with a routing $\gamma_E(e) = a_1, \dots, a_k$ we require $a_1 = \gamma_V(v) + \gamma_P(p_s(e))$ and $a_k = \gamma_V(w) + \gamma_P(p_t(e))$.

1.3.3 Statistics

Statistical methods are important for the experimental evaluation of graph layout methods, where certain criteria are measured for a number of input graphs, e. g. the number of edge crossings in the resulting layouts. Let $x_1, \dots, x_n \in \mathbb{R}$ be a sequence of samples.

Definition 1.12 (Mean, standard deviation). The *mean* of the samples x_1, \dots, x_n is

$$\bar{x} = \frac{1}{n} \sum_{i=1}^n x_i .$$

1. Introduction

The mean is an approximation of the *expected value* μ of the observed variable. The *standard deviation* is the average difference of the samples to this expected value, computed as

$$\sigma = \sqrt{\frac{1}{n} \sum_{i=1}^n (x_i - \mu)^2} .$$

The problem of the above definition of the standard deviation is that the expected value is usually unknown. The approximated mean value could be used instead, but then the approximated standard deviation would be too low because the average difference of a sample to the observed mean value \bar{x} is lower than the average difference to the expected value μ if $\bar{x} \neq \mu$. This can be counterbalanced with a slightly modified formula.

Definition 1.13 (Corrected standard deviation). The *corrected standard deviation* of the samples x_1, \dots, x_n is

$$s = \sqrt{\frac{1}{n-1} \sum_{i=1}^n (x_i - \bar{x})^2} .$$

The mean value is influenced by all observed samples, hence it has a tendency towards extreme values, also called *outliers*. A more stable measure is determined by the *median*.

Definition 1.14 (Median). The *median* of the samples x_1, \dots, x_n is $m = (x_{n/2} + x_{n/2+1})/2$ if n is even and $m = x_{\lceil n/2 \rceil}$ if n is odd, i. e. the number of samples below m equals the number of samples above m .

Often the actual result of the mean or median value is not very meaningful, but it should be compared with the result of another set of samples, e. g. the number of edge crossings obtained with an alternative layout method. Let $x_1, \dots, x_n \in \mathbb{R}$ be samples obtained with a method X and $y_1, \dots, y_n \in \mathbb{R}$ be samples obtained with a method Y . If $\bar{x} > \bar{y}$, and assuming that lower values are better, we could be tempted to deduce that method Y is better than method X . This deduction, however, is only valid if we consider the error of our observations. We do this by computing the probability p that

1.3. Definitions

the observed results have been obtained under the *null hypothesis*, which states that both methods are equally good. If p is too high, we cannot reject the null hypothesis and thus no statement on the relative quality of the two compared methods can be made. If p is sufficiently low, we rate the result as *significant*, which is usually done for $p \leq 5\%$.

The probability p is computed differently depending on whether the two sequences of samples are *paired* or *independent*. Paired samples means that for each $i \in \{1, \dots, n\}$ the samples x_i and y_i have been generated from the same source, e. g. by feeding the same graph into two different algorithms. In both cases we first compute a *t-value* as an intermediate step. This approach is called *t-test*.

Definition 1.15 (*t-value with paired samples*). Let $\bar{\delta}$ be the mean and s_δ be the corrected standard deviation of the sequence $\delta_1, \dots, \delta_n$ formed by $\delta_i = x_i - y_i$ for $1 \leq i \leq n$. The *t-value with paired samples* is

$$t = \frac{\bar{\delta} \cdot \sqrt{n}}{s_\delta} .$$

Definition 1.16 (*t-value with independent samples*). Let $s = \sqrt{(s_x^2 + s_y^2)/2}$ be the combined standard deviation obtained from the corrected standard deviations s_x and s_y of the two sample sets. The *t-value with independent samples* is

$$t = \frac{\bar{x} - \bar{y}}{s \cdot \sqrt{\frac{2}{n}}} .$$

Given a *t-value*, we compute the probability p using *Student's t-distribution* [Fis25].

Definition 1.17 (*p-value of a t-test*). The *p-value* for a given *t-value* is

$$p = \frac{\Gamma(\frac{\nu+1}{2})}{\sqrt{\nu\pi} \cdot \Gamma(\frac{\nu}{2})} \left(1 + \frac{t^2}{\nu}\right)^{-\frac{\nu+1}{2}} ,$$

where $\nu = n - 1$ for paired samples and $\nu = 2n - 2$ for independent samples,

1. Introduction

and

$$\forall a \in \mathbb{R}, a > 0 : \quad \Gamma(a) = \int_0^{\infty} x^{a-1} e^{-x} dx .$$

The function Γ is an extension of the factorial function to real numbers. For positive integers a this yields $\Gamma(a) = (a - 1)!$.

1.4 Graph Layout Methods

A layout algorithm is expected to produce a well readable arrangement. Readability is influenced by many factors [Pur97, WPCM02, BRSG07], e. g. avoiding nodes overlapping each other, avoiding edges crossing each other, or keeping the length of edges short and the overall drawing area small. It is not possible to optimize all criteria; for instance, avoiding edge crossings often leads to longer edges. Furthermore, some criteria may depend on the requirements of the specific application, e. g. whether edges are to be drawn orthogonally or with straight lines. For these reasons there cannot be a single graph layout method that fits to all applications, but different approaches must be studied instead.

There are several books about algorithms for graph layout [DETT99, KW01, Sug02, JM04, Tam13]. This section gives a brief overview of the most important approaches, excluding the layer-based approach, which is discussed with much greater detail in Chapter 2.

1.4.1 Planarization-Based Layout

A graph is called *planar* if it can be drawn without edge crossings. Several algorithms are specialized to create layouts for planar graphs. In the following, let $G = (V, E)$ be an undirected graph.

Planarization. If a graph is not planar, it requires a preprocessing before a planarity-based algorithm can be applied to it. The basic idea is simple: whenever an unavoidable crossing is encountered, it is replaced with a dummy node. After this is done for all edge crossings, the resulting graph is planar. The process of deriving a planar graph from a non-planar graph is

1.4. Graph Layout Methods

called *planarization*. For instance, the edges $\{1, 5\}$ and $\{2, 6\}$ cross each other in the drawing seen in Figure 1.4(c), but in the corresponding planarized graph (Figure 1.4(b)) that crossing is represented by a node v with four connections $\{v, 1\}$, $\{v, 2\}$, $\{v, 5\}$, and $\{v, 6\}$.

Minimizing the number of dummy nodes in the planarized graph is equivalent to minimizing the number of edge crossings, which in turn is an NP-hard problem [GJ83]. However, testing whether a graph is planar can be done in linear time [BM04]. This leads to a simple heuristic for planarization: start with a subgraph (V, E') that is known to be planar, e.g. $E' = \emptyset$, and add each edge e to E' if and only if $(V, E' \cup \{e\})$ is still planar [Wei01]. The remaining edges $E \setminus E'$ are then reinserted one by one, replacing all occurring crossings with dummy nodes as described above. The simplest reinsertion approach is to create an *embedding* of the planar subgraph, which means to fix the order of incident edges of each node such that there exists a planar drawing that respects these edge orders. For a given embedding the set of *faces* can be constructed, i. e. the areas bounded by edges in a planar drawing of the embedding. The *dual graph* (V^*, E^*) consists of the faces V^* connected by edges $\{a, b\} \in E^*$ if and only if there is an edge in E' that borders both a and b (see Figure 1.4(a)). For any edge $e = \{v, w\} \in E \setminus E'$, an embedding of $(V, E' \cup \{e\})$ can be found by searching a shortest path from a face touching v to a face touching w in the dual graph. For instance, the missing edge $\{1, 5\}$ in Figure 1.4(a) can be routed through the path (c, a) in the dual graph, introducing one edge crossing. Alternatively, it is also possible to insert missing edges considering all possible embeddings [GMW05]. Gutwenger and Mutzel compared different planarization approaches with respect to the resulting number of crossings [GM04].

Topology-shape-metrics. Tamassia et al. proposed a method for computing orthogonal drawings in three steps [Tam87, TDB88]: determine the *topology* through planarization, determine the *shape* through *orthogonalization*, and determine the *metrics* through *compaction*.

Orthogonalization means to add 90° bends to the edges of the planarized graph without specifying concrete coordinates of the nodes and bend points. This can be computed efficiently for graphs where every node has at most

1. Introduction

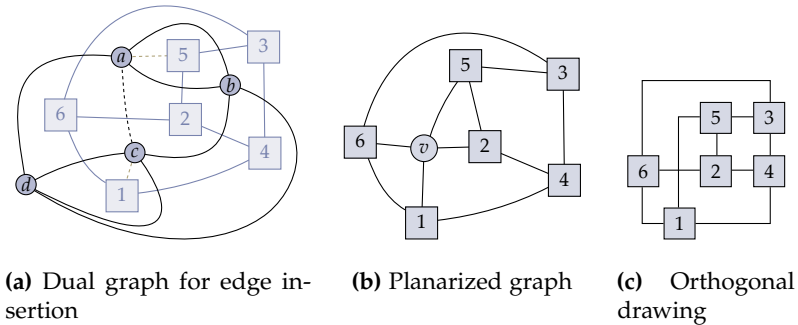


Figure 1.4. Example for the planarization process: (a) the dual graph formed by the faces of a given embedding with a shortest path for the insertion of the missing edge $\{1, 5\}$ (dashed lines); (b) planarized graph with a dummy node v ; (c) orthogonal drawing created with the topology-shape-metrics method.

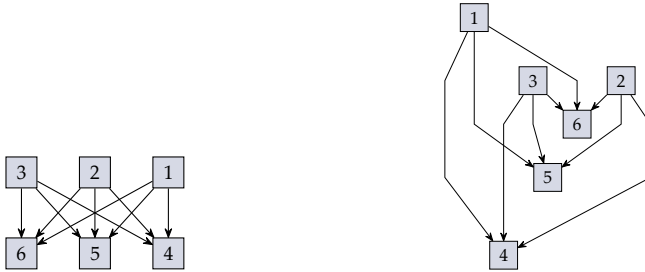
four incident edges [Tam87]. There are different approaches for applying orthogonalization to graphs with arbitrary node degree [TDB88, FK96, KM98]. The general goal is to minimize the number of bends.

Compaction means to assign coordinates to nodes and bend points such that the drawing respects the previously computed planar embedding and orthogonal routing. Possible optimization goals are to minimize the area, the sum of edge lengths, or the length of the longest edge. Again, different approaches have been proposed [KM99, BDD⁺00, EK02].

The drawing in Figure 1.4(c) is made with the topology-shape-metrics method. There are also extensions for UML class diagrams [GJK⁺03, EGK⁺04a], where generalization edges require special treatment because they should preferably be drawn upward.

Upward planarization. The layer-based graph layout approach aligns edges in the same direction by arranging nodes in *layers*. In Figure 1.5(a), for instance, nodes 1, 2, and 3 are in the first layer and nodes 4, 5, and 6 are in the second layer. The resulting drawing is quite compact, but there are a lot of edge crossings. The problem is that first the layers are formed, then the node order is optimized with respect to the number of crossings. However,

1.4. Graph Layout Methods



(a) Layer-based method (9 crossings) (b) Upward planarization (1 crossing)

Figure 1.5. Upward planarization of directed graphs can result in fewer edge crossings, but also longer edges and a larger drawing area.

in this example the number of crossings is the same no matter which node order is chosen for each of the two layers. Chimani et al. have proposed a different approach, called *upward planarization* [CGMW10, CGMW11], that results in considerably fewer edge crossings at the cost of longer edges (see Figure 1.5(b)). The principle of this approach is similar to planarization for undirected graphs: determine a planar subgraph (V, E') and then insert the missing edges $E \setminus E'$, creating a dummy node for each edge crossing. The difference is that the resulting graph must be *upward planar*, i.e. a drawing with no edge crossings exists where all edges are monotonically increasing in the vertical direction. In order to ensure this, the subgraph must be *feasible*, which means that it must be possible to insert the edges $E \setminus E'$ obtaining an upward planar graph [CGMW10].

Once an upward planar embedding is found, a layer assignment and node order can be derived from it, allowing to apply the same methods for node placement and edge routing as in the layer-based approach.

Ports and hyperedges. There are some proposals for extending planarization-based methods to consider ports and hyperedges. Gutwenger et al. included *embedding constraints* in the planarization process [GKM08], which can be used to realize port positioning constraints. Eiglsperger et al. [EFK00] and Siebenhaller [Sie09] discussed approaches to consider such constraints

1. Introduction

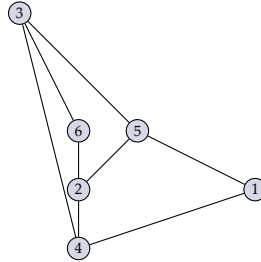


Figure 1.6. A straight-line drawing made with the method of Schnyder [Sch90].

in the orthogonalization step. The handling of hypergraphs has been discussed by Chimani and Gutwenger [CG07]. In a work of Chimani et al., the upward planarization approach has been extended to both embedding constraints and hyperedges [CGM⁺11]. However, none of these extensions have been applied to data flow diagrams yet, hence their practical applicability to this important application domain has still to be evaluated.

Straight-line and mixed-model drawings. Algorithms for straight-line drawings of planar graphs have a high theoretical value, since they are used to prove the existence of such drawings for graphs with certain properties. There are approaches for drawing faces with convex shapes [CON85, Kan96] and approaches based on triangular faces [Sch90, FPP90], i. e. where all faces are bounded by three edges. A graph where all faces are triangular is called *maximal planar* because no further edge can be added to it while preserving planarity. Given an arbitrary planar graph G , a maximal planar graph can be constructed by adding edges to G until all faces are triangular, a process called *augmentation*. Figure 1.6 illustrates such a straight-line drawing based on augmentation (see also Figure 4.23(a) on p. 181).

Planar straight-line drawings are hardly usable in practice: they often require a large drawing area, and the nodes are spread rather unevenly in that area. Kant built on the ideas of Schnyder [Sch90] and de Fraysseix et al. [FPP90], who used a specific node ordering named *canonical ordering* in their algorithms, to develop an orthogonal drawing algorithm [Kan96]. Furthermore, he proposed a *mixed-model* approach that produces

quasi-orthogonal drawings, where line segments of edges are aligned horizontally or vertically except the segments directly connected to nodes. His approach requires the graph to be planar and 3-connected, hence general planar graphs need to be augmented in order to increase their connectivity, possibly compromising the quality of the drawings. Gutwenger and Mutzel proposed another method for mixed-model drawings that can be applied to arbitrary connected planar graphs [GM98] (see Figure 4.23(b) on p. 181).

1.4.2 Force-Based Layout

Eades proposed to draw undirected graphs by computing attracting and repelling forces between nodes and repeatedly moving each node towards the sum of its forces [Ead84]. Starting with an arbitrary node positioning, the forces tend to reach an equilibrium after a certain number of iterations, often yielding quite balanced and well readable drawings. In this model, two adjacent nodes v_1, v_2 with positions p_1, p_2 attract each other with the force

$$F_a^E(v_1, v_2) = C_1 \cdot \log \frac{\|p_1 - p_2\|}{C_2} ,$$

and if they are nonadjacent a repelling force

$$F_r^E(v_1, v_2) = \frac{C_3}{\sqrt{\|p_1 - p_2\|}}$$

is applied. $C_1, C_2,$ and C_3 are constants to be determined experimentally. Another model has been proposed by Fruchterman and Reingold, who applied an attracting force

$$F_a^{FR}(v_1, v_2) = \frac{\|p_1 - p_2\|^2}{k}$$

between adjacent nodes and a repelling force

$$F_r^{FR}(v_1, v_2) = \frac{k^2}{\|p_1 - p_2\|}$$

1. Introduction

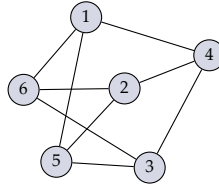


Figure 1.7. A force-based drawing made with the force model of Fruchterman and Reingold [FR91].

between all nodes [FR91]. Note that if $\|p_1 - p_2\| = k$, the two forces are equal, thus k can be used to control the length of edges. An example is shown in Figure 1.7.

Kamada and Kawai based their method on the *total energy*

$$E = \sum_{i=1}^{n-1} \sum_{j=i+1}^n \frac{K}{2d_{ij}^2} (\|p_i - p_j\| - L \cdot d_{ij})^2$$

caused by the node positions p_1, \dots, p_n [KK89]. K and L are constants, and d_{ij} is the *graph-theoretical distance*, i. e. the number of edges on a shortest path between nodes v_i and v_j . The goal is to find a local minimum of the energy function E , which is done using the partial derivatives of that function. The resulting drawings tend to be more balanced than those created with simpler force-based methods [Ead84, FR91], but the computation time of the method of Kamada and Kawai is much higher. Gansner et al. later improved the computation time with an approach called *stress majorization* [GKN05]. For graphs with thousands of nodes this may still be too slow. In these cases *multilevel* approaches can be applied [BGKM11] in combination with an approximation of the repelling forces [HJ05], eliminating the quadratic computation time required for comparing all pairs of nodes.

1.4.3 Tree and Circular Layout

In general, trees are easier to arrange compared to arbitrary graphs, since they can always be drawn without crossings, and a predefined *root node*

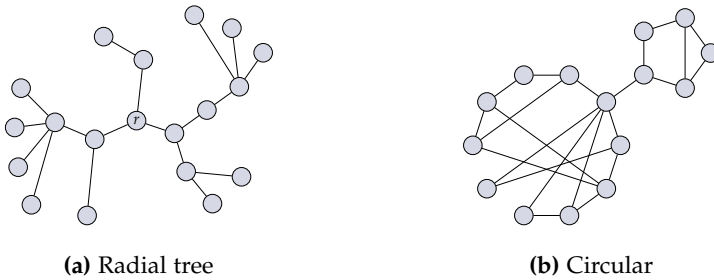


Figure 1.8. (a) A radial tree layout with a root r , (b) a circular layout with two biconnected components.

already determines the coarse structure of the drawing. However, there is still a lot of freedom regarding the style of node placement [Rus13], e. g. *layered* style, *radial* style, and many others. The layered style is similar to general layer-based drawing (Chapter 2), where nodes are assigned to layers such that edges are aligned in the same direction. Radial tree drawings are a variant of layered drawings where the layers are concentric circles with the root in the center. An example is shown in Figure 1.8(a).

The goal of a circular layout algorithm is to arrange the nodes of a biconnected graph G in a circle with few edge crossings. Six and Tollis proposed a method that yields zero crossings if this is possible [ST99]. An alternative has been developed by Baur and Brandes [BB05]. If the input graph is not biconnected, it can be split into its biconnected components, which can each be arranged in circles and then be composed with a tree layout algorithm [ST06], as illustrated in Figure 1.8(b).

1.5 The KIELER Project

“Graphics do not guarantee clarity: ‘good’ graphics relies on secondary notation”—Marian Petre [Pet95]. With this insight in mind, Prochnow and von Hanxleden initiated the project *Kiel Integrated Environment for Layout* (KIEL) in 2004. The goal was to investigate new interaction techniques for graphical modeling using the statecharts notation [PvH06, PvH07] in order to

1. Introduction

enhance the *pragmatics* of model-based design, i. e. the practical aspects of handling models [vH08]. This was done with a Java-based application that included, among other features, a graphical statecharts viewer with automatic layout, a textual notation for statecharts, and a simulator with *SyncCharts* semantics [And03]. Graph layouts were mainly done with the *Graphviz* tool [GN00].

KIELER¹² is a follow-up project of KIEL, started by Fuhrmann in 2008 (ER stands for *Eclipse Rich Client*). One goal was to overcome the limitations of KIEL by building on the Eclipse platform (see Chapter 4) and thus allowing to apply the various techniques addressing modeling pragmatics to a large number of applications instead of only statecharts [FvH10a, FvH10b]. There are three main research tracks in KIELER: *pragmatics*, *semantics*, and *layout*. Regarding pragmatics, the main focus is to eliminate the time-consuming manual creation of graphical models, e. g. through *structure-based editing* [FSMvH10] or automatic view synthesis [SSvH12a, SSvH13]. Automatic layout is the key enabler for these techniques, allowing users to concentrate on the really important tasks and leaving view creation and adaptation to the computer. The semantics track targets infrastructures for simulation and code generation as well as extensions of the synchronous model of computation [vHDM⁺14]. The layout track is largely the content of this thesis: a generic graph layout infrastructure for model-driven engineering that includes high-quality layout algorithms for special notations such as data flow diagrams. Further research directions have evolved in this area during the work on this thesis, e. g. the handling of floating comments [SvH14] and the energy-based layout of data flow diagrams [RKD⁺14].

1.6 Related Work

As explained before, this thesis approaches the automatic layout of graphical models on three separate levels: graph layout algorithms, layout configuration, and integration into modeling tools. Previous work related to these three topics is discussed in the following.

¹²<http://www.informatik.uni-kiel.de/rtsys/kieler/>

Graph layout algorithms. Some graphical notations that are frequently used in MDE have already been studied with respect to their layout requirements, e. g. UML class diagrams [See97, EGK⁺04a] and statecharts [CMT04]. Data flow diagrams, however, to my knowledge have not been directly addressed until 2009 [SFvHM10], although there is a large number of applications employing this kind of notation. Parts of the problem of drawing data flow diagrams have been studied based on other notations with similar requirements. The most important of these requirements are the consistent alignment of edge directions, the handling of port constraints, and the orthogonal routing of hyperedges. Here we focus on the layer-based graph layout approach because it already satisfies the alignment of edge directions by its design (see Section 2.1).

The first contributions to integrating port constraints in the layer-based approach were motivated by the layout of data structures, where certain fields of a structure may contain pointers to other structures. Gansner et al. showed how node positioning can be extended for including offsets derived from port positions [GKNV93]. Sander introduced the idea of handling side ports by adding dummy nodes in order to route the respective edges [San94]. These dummy nodes are positioned with a local approach, which can lead to unpleasant layouts since the number of resulting bend points is possibly higher than necessary. Here we address this problem with a *global* approach, described in Section 2.2.3. The problem of crossing minimization with port constraints was first discussed by Waddle, who adapted the barycenter heuristic to consider port positions [Wad01]. This works for ports constrained to fixed positions, but not for other kinds of port constraints. We discuss a more flexible approach in Section 2.2.2.

Schreiber proposed different solutions in the context of drawing biochemical networks [Sch01]. The crossing minimization phase is adapted by inserting dummy nodes for each port and adding constraints to respect the order of ports. Side ports are handled by routing the incident edges locally for each node, which is done through transformation into a two-layer crossing minimization problem. A similar local approach was proposed by Siebenhaller [Sie09]. However, it supports more flexible port constraints by associating them with individual edges. The consequence is that a node may have some edges that are constrained to ports, and some that are not. This

1. Introduction

flexibility can be useful for the layout of UML diagrams, where it is possible that only a subset of the edges is connected to fixed points of a node. The crossing minimization problem that results from this additional degree of freedom can be partly solved by reducing it to a network flow problem. The local kind of edge routing employed by Schreiber and Siebenhaller is subject to the same problems as mentioned above for Sander’s work [San94].

Klauske and Dziobek introduced a specialized node placement for ports where positions depend linearly on the node sizes [KD10, Kla12] in the context of the Simulink modeling language. Their approach is an extension of the linear program for node placement proposed by Gansner et al. [GKNV93]. This extension does not only determine vertical positions for the nodes, but also modifies their height in order to find an optimal placement of ports. The node placement phase is briefly discussed in Section 2.1, but it is not addressed with further detail in this thesis.

Gutwenger et al. introduced the concept of *embedding constraints* for modeling the port constraints of a node [GKM08], but that model captures only the order of ports, and not their concrete positions. Such constraints are used in the context of planarization-based layout, e. g. the topology-shape-metrics approach [TDB88]. Harrigan and Healy applied embedding constraints to *level planarization* [HH08], which consists in finding a planar subgraph respecting a given layer assignment. The level planarization approach has not yet been compared with the barycenter extensions discussed here. An experimental evaluation of these two methods is an interesting topic for future research.

Eschbach et al. [EGB06] and Sander [San04] proposed methods for the orthogonal routing of edges in the layer-based approach. They both noted that the number of crossings determined during the node ordering phase is only an approximation, but gave no proposals on how to solve this problem. Here two alternative approximations are proposed, and experimental results suggest that they are much more accurate compared to the standard approach (see Section 2.3.3).

Graph layout configuration. Several authors have proposed evolutionary graph layout algorithms where the individuals are represented by lists of coordinates for the positions of the nodes of a graph [BB01, BBS96, EM01,

GMEJ90, RSOR98, Tet98, Vra09]. In the *evolutionary meta layout* approach presented here in Section 3.2, in contrast, we do not include any specific graph in our encoding of individuals, hence we can apply the result of our evolutionary algorithm to any graphs, even if they were not considered during the evolutionary process. Furthermore, we benefit from all features that are already supported by the existing algorithms, while previous approaches for evolutionary layout were usually restricted to draw edges as straight lines and did not consider additional features such as edge labels.

Other works have focused on integrating meta heuristics in existing layout methods. De Mendonça Neto and Eades proposed a system for automatic learning of parameters of a simulated annealing algorithm [dMNE93]. Utech et al. introduced a genetic representation that combines the layer assignment and node ordering steps of the layer-based drawing approach with an evolutionary algorithm [UBSE98]. Such a combination of multiple NP-hard steps is also applied by Neta et al. for the topology-shape-metrics approach [NAGa⁺12]. They use an evolutionary algorithm to find planar embeddings (*topology* step) for which the other steps (*shape* and *metrics*) are able to create good layouts.

Bertolazzi et al. proposed a system for automatic selection of layout algorithms that best match the user's requirements [BDL95]. The system is initialized by evaluating the available algorithms with respect to a set of aesthetic criteria using randomly generated graphs of different sizes. The user has to provide a ranking of the criteria according to her or his preference. When a layout request is made, the system determines the difference between the user's ranking and the evaluation results of each algorithm for graphs of similar size as the current input graph. The algorithms with the lowest difference are offered to the user.

Similarly, Niggemann and Stein proposed to build a database that maps vectors of structural graph features, e.g. the number of nodes and the number of connected components, to the most suitable layout algorithm with respect to some predefined combination of aesthetic criteria [NS00]. These data are gathered by applying the algorithms to a set of "typical" graphs. A suitable algorithm for a given input graph is chosen by measuring its structural features and comparing them with the entries present in the database. Both the approaches of Bertolazzi et al. and Niggemann and Stein

1. Introduction

are restricted to selecting layout algorithms. Here, in contrast, we seek to configure arbitrary parameters of algorithms in addition to their selection.

Archambault et al. combined graph clustering with layout algorithm selection in a multi-level approach [AMA07]. The clustering process is tightly connected with the algorithm selection, since both aspects are based on topological features of the input graph. When a specific feature is found, e. g. a tree or a clique, it is extracted as a subgraph and processed with a layout algorithm that is especially suited for that feature. This kind of layout configuration depends on detailed knowledge of the behavior of the algorithms, which has to be encoded explicitly in the system, while our evolutionary meta layout approach can be applied to any algorithm independently of their behavior.

Graph layout software. There are numerous tools and software libraries that offer automatic graph layout [JM04]. However, most of these are of rather general nature and do not directly address model-driven engineering. In the 1990s, efforts were made to develop generic graph editors with included layout algorithms, e. g. EDGE [Pau93] and GraphEd [Him95] written in C and C++, and Grasper-CL [KLSW94] written in Lisp. Soon after Java became an established programming language, graph layout libraries for Java applications were developed, e. g. JMFGraph [Ste01], GVS [Pri06], and yFiles [WEK04]. The Tulip framework is specialized for large-scale information visualization [AAB⁺12], supporting clustering of huge graphs for reducing complexity. The OGDF library contains implementations of numerous layout algorithms, especially planarity-based and force-based methods [CGJ⁺13]. Graphviz [GN00] is probably the longest standing and most widespread graph layout tool (cf. the results to Question 28 of the survey presented in Section 6.1.2). Gansner attributes this success to the simple and flexible interface and the support of practical features [Gan11], e. g. rendering directives included in textual graph descriptions.

Bull et al. proposed the concept of *model-driven visualization* (MDV) [BF05], where MDE techniques are applied to the process of visualizing information. They also highlighted the importance of integrating such visualization tools into software development platforms, and initiated the Zest project [BBS04], which has become a part of the Eclipse Graphical

Editing Framework (GEF). This is an important forerunner of KLighD, a visualization tool introduced in Section 4.3.3 that employs the KIELER layout infrastructure.

Some authors presented approaches for graph layout support in certain metamodeling tools, i.e. tools for specifying modeling languages. Dubé implemented a number of layout algorithms for AToM³ [Dub06], a Python-based multi-formalism metamodeling tool [dLV02]. The Eclipse-based framework Marama [GHHL08] is a comprehensive alternative to the Eclipse Modeling Framework (EMF) and the Graphical Modeling Framework (GMF), which are introduced in Chapter 4 (cf. the TIGER framework [EEHT05]). Marama was extended with a force-based and a tree layout algorithm by Yap et al. [YHG11], allowing to assign a suitable algorithm on the language specification level. Maier and Minas proposed a constraint-based layout algorithm [MM08] for DIAMETA, a stand-alone Java tool for generating graphical editors [Min06]. This algorithm is combined with graph layout algorithms in a pattern-based approach [MM10a]. From the perspective of these extensions of AToM³, Marama, and DIAMETA, the KIELER layout infrastructure presented here could be seen as an extension of EMF, as it has mostly been applied to EMF-based models. However, KIELER does not strictly depend on any kind of metamodel, and it can also be used to visualize information without a formal metamodel. Here we approach the problem of automatic layout in a more general way compared to previous solutions, targeting to connect arbitrary modeling tools with arbitrary graph layout algorithms. Furthermore, we propose data structures and methods for algorithm selection and configuration such that the capabilities of graph layout libraries can be exploited by tool developers and users, and we emphasize the special requirements of data flow diagrams in the development of layout algorithms.

Part I

Layout Algorithms

The Layer-Based Approach

The *layer-based* graph layout approach was introduced by Sugiyama et al. [STT81]. Given a directed acyclic graph, this approach arranges all edges in the same direction by organizing the nodes in subsequent *layers*, which are also called *hierarchies* [STT81] or *levels* [JLM98] in graph drawing literature. Layer-based algorithms have been very popular both in research [GKNV93, ESK04] and in practical modeling applications (e. g. the “Arrange All” feature provided by the Eclipse GMF project), since many applications require graphs to be directed. Furthermore, the relatively simple structure of this approach allows it to be extended for many special applications, such as run-time data structures [San94, Wad01], biological networks [Sch01], UML class diagrams [See97], or statecharts [CMT02]. The extension to data flow diagrams has been investigated recently [SFvHM10, KD10, KSSvH12, SSVH14] and is the main contribution of this thesis with respect to layout algorithms.

Section 2.1 summarizes the structure, problems, and heuristics of the layer-based approach. Section 2.2 presents extensions for considering port constraints, which are the main challenge for the layout of data flow diagrams. Further relevant aspects that are considered here are hyperedges, discussed in Section 2.3, and the user’s interaction with the layout algorithm, for which a sketch-driven approach is presented in Section 2.4.

Most of the graph drawing literature assumes top-to-bottom orientation of edges [STT81, GKNV93, ESK04]. However, a left-to-right orientation is commonly used for data flow diagrams, thus all algorithms described here follow the assumption that the main orientation of edges is from left to right.

2. The Layer-Based Approach

2.1 Base Algorithms

The layer-based approach solves the graph layout problem by dividing it into five consecutive phases.

1. *Elimination of Cycles.* Directed cycles can be eliminated by reversing a subset of the edges. Usually the aim is to minimize this subset in order to have as many edges as possible pointing in the same direction in the final drawing.
2. *Layer Assignment.* Nodes are assigned to layers L_1, \dots, L_k such that all edges point from layers of lower index to layers of higher index. This is possible because the graph is acyclic after the previous step.
3. *Crossing Minimization.* The nodes of each layer L_i are ordered with the goal of minimizing the number of edge crossings that can occur between pairs of layers.
4. *Node Placement.* The nodes of each layer L_i are assigned a vertical position according to the order determined in the previous step. Positions should be chosen such that the edges can be drawn as straight as possible.
5. *Edge Routing.* This final phase adds bend points to edges, depending on the desired routing style. Popular variants are polyline routing, orthogonal routing, or cubic splines.

The following sections give an overview of the optimization problems and heuristics for each of the five phases. More details are reported by Di Battista et al. [DETT99, Chapter 9] and Bastert and Matuszewski [BM01].

2.1.1 Elimination of Cycles

Given a graph $G = (V, E)$, the cycle elimination phase aims at finding a subset of edges $S \subseteq E$ such that the graph that is derived from G by reversing the edges in S is acyclic. The original direction of these edges is restored after all other phases have been executed. Assuming edges are arranged from left to right, those in the subset S would then point from right to left in the final drawing. The most obvious optimization goal is

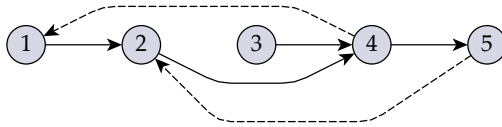


Figure 2.1. Reversing the left-pointing edges $(4,1)$ and $(5,2)$ in the node sequence makes the graph acyclic.

to minimize $|S|$, which implies the maximization of edges pointing in the same direction. This problem, called the *feedback arc set* problem, is NP-hard [GJ79], and many approximations have been studied [BS90, ENSS98, DF03].

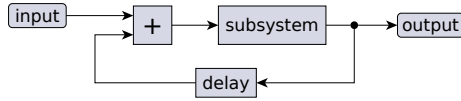
Eades et al. proposed a heuristic that is very fast and easy to implement [ELS93] and is therefore often used in the context of graph layout. This heuristic is based on the observation that a feedback arc set S can be derived from any sequential numbering v_1, \dots, v_n of the nodes by reversing every edge $e = (v_i, v_j)$ for which $i > j$ (see also the work of Brandenburg and Hanau [BH11]). An example is shown in Figure 2.1, where two edges have their target node left of their source node. Note that after reversing these edges the numbering v_1, \dots, v_n is a valid topological numbering.

The data flow diagram shown in Figure 2.2(a) has a delay operator in a feedback loop. Choosing an arbitrary optimal solution to the feedback arc set problem could result in a layout like in Figure 2.2(b), where the meaning of the diagram is much less clear despite the reduced number of feedback edges. The reason for this discrepancy is that for readers of a data flow diagram the direction of an edge is related to the flow of data, which is expected to match the semantic interpretation of the connected nodes. Therefore the fact that the delay operator of Figure 2.2(a) semantically feeds the system’s output data back to its input is adequately reflected by reversing the incident edges of that operator—even though this results in a larger number of reversed edges.

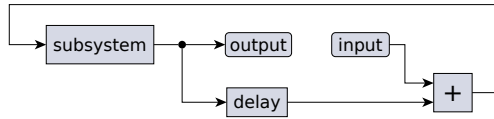
2.1.2 Layer Assignment

The second phase of the layer-based approach is responsible for finding a sequence L_1, \dots, L_k of layers and to assign each node to one of them.

2. The Layer-Based Approach



(a) The expected feedback arc set contains the incident edges of delay



(b) An alternative layout that makes the flow of data more difficult to understand

Figure 2.2. A data flow diagram with a feedback loop.

Definition 2.1 (Layering, long and short edges). A *layering* of a graph is a disjoint decomposition $\mathcal{L} = \{L_1, \dots, L_k\}$. Let $e = (v, w)$ be an edge with $v \in L_i$ and $w \in L_j$. The *layer span* of e is $s_e = j - i$. We call \mathcal{L} a *valid layering* if $s_e > 0$ for all edges. Given a valid layering, we call e a *long edge* if $s_e > 1$, otherwise it is a *short edge*. A valid layering where all edges are short is called a *proper layering*.

The requirement of valid layerings means that the directions of edges must be respected, allowing to draw all edges in the same direction. Proper layerings are useful because in the remaining algorithm phases we can assume that all edges connect only nodes from consecutive layers. This assumption makes the theory and implementation of those phases a lot easier. A proper layering is derived from an arbitrary layering by splitting each long edge e using a series of dummy nodes: if L_i is the source layer and L_j is the target layer, we replace $e = (v, w)$ by new edges $(v, d_{i+1}), (d_{i+1}, d_{i+2}), \dots, (d_{j-1}, w)$ and dummy nodes $d_{i+1} \in L_{i+1}, \dots, d_{j-1} \in L_{j-1}$ (see Figure 2.3).

Provided that the overall layout orientation is left-to-right, the nodes of each layer are arranged vertically. The *width* of a layering is $w(\mathcal{L}) = |\mathcal{L}|$, i. e. the number of layers (traditional literature calls it the *height* because it assumes top-to-bottom orientation). Accordingly, the *height* (resp. *width* in other literature) is the size of the largest layer, defined by $h(\mathcal{L}) = \max\{|L_i| :$

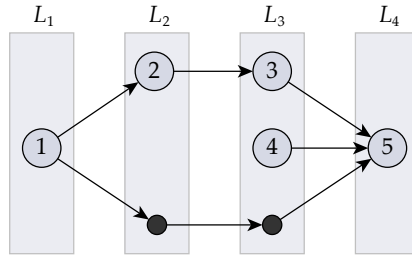


Figure 2.3. A proper layering \mathcal{L} with two dummy nodes in the layers L_2 and L_3 to split the long edge $(1,5)$. The width is $w(\mathcal{L}) = 4$, the height is $h(\mathcal{L}) = 2$, and the proper height is $\hat{h}(\mathcal{L}) = 3$.

$L \in \mathcal{L}$. Sometimes the height is measured including the dummy nodes created for making the layering proper, since they also influence the size of the final drawing. Here we call this the *proper height* $\hat{h}(\mathcal{L})$.

Choosing a good layering is crucial for obtaining a compact layout, especially regarding the aspect ratio. A layering with minimal height $h(\mathcal{L}) = 1$ can be derived from any topological ordering, e. g. as computed by the cycle elimination algorithm of Eades et al. (see Section 2.1.1), by putting each node into its own layer. However, the proper height $\hat{h}(\mathcal{L})$ can be much greater than 1 in this case. It is also easy to compute a layering with minimal width using the *longest path* algorithm: let v_1, \dots, v_k be a longest directed path, then each sink of the graph is put into the layer L_k and each other node v into L_{k-l} , where l is the number of edges on the longest path from v to any sink. The width of the resulting layering is minimal with respect to the feedback arc set determined in the cycle elimination phase, but the height can be very high since all sink nodes are stacked in the last layer. Furthermore, longest path layerings tend to contain a lot of long edges, which makes the layout difficult to read.

More balanced layerings can be obtained with the algorithm of Gansner et al., who use a *network simplex* approach to minimize the length of edges [GKNV93]. While the width of the layerings produced by their algorithm is usually close to the minimum, their height is not controlled, sometimes leading to bad aspect ratios. Layering methods that respect both the width

2. The Layer-Based Approach

and the height have been studied by Healy et al. [HN02], Nikolov et al. [NTB05], and Andreev et al. [AHN07].

2.1.3 Crossing Minimization

Given a specific layering (L_1, \dots, L_k) , the number of edge crossings in the final drawing mainly depends on the vertical order of nodes in each layer. Hence we seek an ordering with minimal number of edge crossings. This problem is NP-hard [GJ83], therefore several approximations have been studied.¹ The *barycenter* heuristic, proposed by Sugiyama et al. [STT81], is probably the most popular, since it is simple, fast, and gives reasonably good results [JM97]. It reduces the ordering problem to consider only two layers L_a and L_b at a time, where L_a is kept fixed and L_b is reordered.

Definition 2.2 (Barycenter value). For each node $v \in L_a$ let $r(v)$ be its position in the fixed order of L_a . The *forward barycenter value* of a node $w \in L_b$ is

$$b(w) = \frac{1}{|E_i(w)|} \sum_{(v,w) \in E_i(w)} r(v) .$$

The *backward barycenter value* is defined accordingly using the outgoing edges $E_o(w)$ instead of $E_i(w)$.

The barycenter value corresponds to the average position of the connected nodes in L_a . The forward barycenter is used if L_a is left of L_b in the ordering of layers, and the backward barycenter is used for the opposite case. The order of L_b is determined by sorting the contained nodes by their barycenter values. The barycenter method can be used to order all layers using the *layer sweep* algorithm:

1. Determine a random order for the nodes of L_1 .
2. Repeat for $i = 2, \dots, k$: Reorder the nodes of L_i such that the number of crossings of edges with their source in L_{i-1} and their target in L_i is minimal.

¹Since the heuristics cannot always produce optimal solutions, this phase is often called *crossing reduction* rather than *crossing minimization*.

3. Repeat for $i = k - 1, \dots, 1$: Reorder the nodes of L_i such that the number of crossings of edges with their source in L_i and their target in L_{i+1} is minimal.
4. Repeat steps 2 and 3 until the total number of crossings is not further decreased.

The reordering in steps 2 and 3 is usually done with the barycenter method (forward barycenter for step 2 and backward barycenter for step 3), but there are also alternatives such as the *median* method [EW86], which takes the position in the middle instead of the average value. A comparison of different crossing minimization heuristics is given by Jünger and Mutzel [JM97]. Generally the result of the layer sweep algorithm can be improved by repeating it with different random initial orderings, and then selecting the result that produced the least number of edge crossings. This selection process as well as step 4 of the layer sweep algorithm require a method for counting the number of crossings that result from a given layering. There are simple and efficient algorithms for counting crossings, e. g. the algorithm of Barth et al. [BMJ04].

A graph with a layering \mathcal{L} is called *level planar* if it can be drawn with straight edges free of crossings respecting \mathcal{L} [JLM98]. *Level planarization* is an approach for crossing minimization that does not directly reorder the nodes of the layered graph, but first reduces the graph to make it level planar [Mut97, GSM11]. The planarization process consists of removing edges to obtain a level planar subgraph, determining a node ordering for that subgraph that admits a drawing without edge crossings, and finally reinserting the missing edges. The implementation of this approach is by far more complex compared to other heuristics, therefore it is rarely employed in practice.

The computation of optimal solutions for the crossing minimization problem has been studied using linear programming [JLMO97], satisfiability solvers [GSM11], and semidefinite programming [CHJM12], which is another method for combinatorial optimization. The computing hardware and optimization tools that are available today allow to obtain optimal solutions for graphs of small or moderate size.

2. The Layer-Based Approach

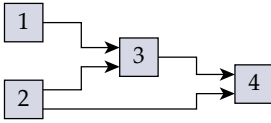
2.1.4 Node Placement

After the order of nodes has been fixed, their concrete coordinates can be computed. The horizontal coordinates are mainly determined by the order of layers. The space required between two consecutive layers depends on the type of edge routing, which is discussed in Section 2.1.5. If nodes have different sizes, their horizontal alignment inside the containing layer has to be chosen, e. g. whether they are centered on a vertical axis or aligned left or right. Since this horizontal placement is straightforward, the graph drawing literature has focused on the vertical placement of nodes.²

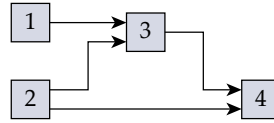
The goal of vertical coordinate assignment is to obtain a *balanced* layout. Usually this means to minimize the length of edges: if for each node v we denote by $y(v)$ the assigned position of v , then the length of an edge $e = (v, w)$ is minimal when $|y(v) - y(w)| = 0$. Gansner et al. showed that the sum of these position differences over all edges can be minimized efficiently [GKNV93]. Sander proposed a *pendulum* method for balanced placement [San94], which starts with an initial unbalanced placement and iteratively improves it by computing the average of adjacent node positions for each node. Other methods that target edge lengths have been proposed by Buchheim et al. [BJL01] and Brandes and Köpf [BK02].

Placement methods that focus on edge lengths mostly do not consider other criteria such as the number of edge bends. This aspect is particularly important when orthogonal edge routing is applied, as is the case in Figure 2.4. Some methods group the dummy nodes created for splitting long edges (see Section 2.1.2) in order to guarantee that the edges connecting these dummy nodes are drawn without bends [San96a, BJL01]. This improves the number of bends for long edges, but not for other edges such as $(1, 3)$ in Figure 2.4(a). A more significant improvement of the number of bends can be achieved by using the *median* of adjacent node positions instead of the average, since that means that at least one incident edge of each node can be drawn straight [Car12].

²This is called *horizontal* placement where the layout direction is top-to-bottom.

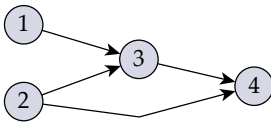


(a) Balanced placement without considering edge bends (8 bends)

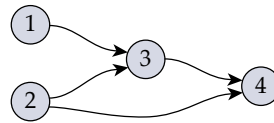


(b) Placement with reduced edge bends (4 bends)

Figure 2.4. Node placement affects both the length of edges and their number of bends when edges are routed orthogonally.



(a) Straight-line routing



(b) Spline based routing

Figure 2.5. Alternative edge routing styles for the graph shown in Figure 2.4(a).

2.1.5 Edge Routing

The simplest form of edge routing is to use straight lines, where bend points are inserted only in place of long edge dummy nodes, as seen for the edge (2,4) in Figure 2.5(a). This is the routing style that is prevalent in the graph drawing literature, including the original publication of Sugiyama et al. [STT81]. In many applications it is preferred to represent edges by smooth curves, i. e. cubic Bézier splines (see Figure 2.5(b)). In this case the layout algorithm has to compute *control points* for the Bézier curves instead of bend points. Methods for the computation of control points have been proposed by Gansner et al. [GKNV93] and Sander [San94].

An orthogonal routing style, which is strictly required in some applications, can be realized by processing each pair (L_a, L_b) of consecutive layers, adding two bend points to each edge that points from L_a to L_b . This introduces one vertical and two horizontal line segments for each edge, except for those where the start and end point are at the same vertical position (see Figure 2.4). Since the line segments of different edges may cross, it is

2. The Layer-Based Approach

important to arrange the vertical segments in such an order that the number of crossings is minimized. Algorithms for ordering vertical segments have been proposed by Sander [San96a] and Baburin [Bab02].

2.2 Port Constraints

Many modeling applications that use ports do not allow layout algorithms to modify the positions of ports relative to the node they are attached to. Other applications allow repositioning ports within certain constraints. Gutwenger et al. introduced the concept of *embedding constraints* for modeling the port constraints of a node [GKM08], but that model only captures the order of ports, and not their concrete position. Eiglsperger et al. [EFK00] and Siebenhaller [Sie09, Section 3.1.5] allow each edge to have individual constraints regarding the side and position of connected ports. This means that a node may have ports that are bound to specific positions as well as free edges that can be attached to arbitrary positions, which can be useful for some applications such as class and component diagrams of the UML.

The port constraints model used in this thesis assigns a constraint level $PC(v)$ to each node v , hence all incident edges of v are subject to the same kind of constraint [KSSvH12, SSvH14]. Extending the approaches presented here to consider mixed constraints, i. e. with both bound and free edges, is not required for most data flow languages and is left for future research.

Definition 2.3 (Port constraints assignment). Let V be a set of nodes. A *port constraints assignment* is a function

$$PC : V \rightarrow \{\text{FREE}, \text{FIXEDSIDE}, \text{FIXEDORDER}, \text{FIXEDRATIO}, \text{FIXEDPOS}\} .$$

The function values $PC(v)$ for $v \in V$ are interpreted as follows.

FREE	Ports can be placed at arbitrary positions on the boundary of node v .
FIXEDSIDE	A side of v , denoted by $\text{side}(p) \in \{\text{north}, \text{east}, \text{south}, \text{west}\}$, is assigned to each port $p \in P(v)$.

<code>FIXEDORDER</code>	The side of each port is fixed, and the ports of each side have to be placed in a specific order.
<code>FIXEDRATIO</code>	Each port is assigned a position; if the containing node is resized, these port positions are scaled accordingly.
<code>FIXEDPOS</code>	Each port is assigned a position that must not be modified by the layout algorithm.

This section describes the challenges faced when handling port constraints in the layer-based graph layout approach and presents previous solutions as well as new insights. Some of the previous solutions were developed in the context of my diploma thesis [Spö09], and some in the diploma thesis of Schulze [Sch11].

2.2.1 Handling Constraint Levels

Let $v \in V$ be a node and $\text{PC}(v)$ be the port constraint level of v . The basic principle for handling this constraint level is to lift it to stricter values during the processing of the five phases of the layer-based approach. One important goal is to modify the algorithms employed in the five phases as little as possible, and to realize most of the extensions in additional pre-processing or post-processing algorithms (see also Section 4.4.1). This allows a modular implementation with a clear separation of concerns and helps to tame the complexity of the problems related to port constraints. In this section we examine the transitions of constraint levels, of which an overview is given in Figure 2.6.

Free \rightarrow Fixed Side. If $\text{PC}(v) = \text{FREE}$, all edges can be aligned to the main layout direction, which we assume to be left-to-right throughout this chapter. This means that ports with incoming edges should be placed on the west side of v , while ports with outgoing edges should be placed on the east side. However, in the rare case that a port p has both incoming and outgoing edges, a compromise has to be found by considering the difference of the number of edges.

- If $|E_i(p)| - |E_o(p)| > 0$, assign the side $s(p) = \text{west}$.

2. The Layer-Based Approach

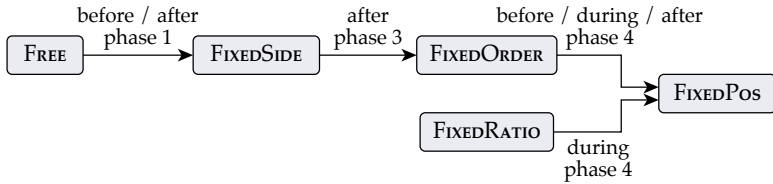
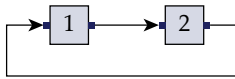
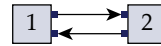


Figure 2.6. Overview of port constraint level transitions in the five phases of the layer-based layout approach. All constraint levels are possible as starting condition for a node.



(a) Before cycle elimination



(b) After cycle elimination

Figure 2.7. Transition from FREE port constraints to FIXEDSIDE before or after the cycle elimination phase.

- If $|E_i(p)| - |E_o(p)| < 0$, assign the side $s(p) = \text{east}$.
- If $|E_i(p)| - |E_o(p)| = 0$, choose an arbitrary side.

The transition to fixed sides must be done before the crossing minimization phase, since the decision which side to assign to each port has an impact on the number of edge crossings. If the transition is done before the cycle elimination phase, the incident edges of each node are consistently attached west or east depending on whether they are incoming or outgoing (see Figure 2.7(a)). In contrast, if the transition is done after cycle elimination, some edges may be reversed, hence they are attached to the opposite sides (see Figure 2.7(b)). The variant shown in Figure 2.7(a) emphasizes more clearly the flow represented by the edges, but the variant in Figure 2.7(b) is more compact. Therefore both options are valid and should be available in an implementation.

Fixed Side \rightarrow Fixed Order. If $PC(v) = \text{FIXEDSIDE}$ after the nodes of each layer have been ordered, it is necessary to order the ports on each side of v

such that the number of crossings is minimized. Since v can contain arbitrarily many ports and each port can have arbitrarily many incident edges, the port ordering problem is equivalent to the two-layer crossing minimization problem where one layer is free and the other is fixed (see Section 2.1.3). As a consequence, ordering the ports of a node optimally is NP-hard [GJ83], but reasonably good solutions can be found with an adapted version of the barycenter heuristic that is used for two-layer crossing minimization. The adapted heuristic is described in Section 2.2.2. If dummy nodes are used to route edges according to prescribed node sides, the order of these dummy nodes must be considered when sorting ports, which is described in Section 2.2.3.

Fixed Order \rightarrow Fixed Position. If $\text{PC}(v) = \text{FIXEDORDER}$, the final constraint level transition consists in setting concrete coordinates for each port. A straightforward method for this is to distribute the ports evenly on the boundary of v before the node placement phase. Then the node placement algorithm is responsible for considering these relative port coordinates in order to straighten the edges as much as possible (see Section 2.1.4).

The extension of a specific node placement method to include port coordinates has been discussed by Gansner et al. [GKNV93], and similar extensions can be done for other methods, too [Car12]. However, there are cases where the edges could be further straightened by modifying port positions: the even distribution of ports in Figure 2.8(a) causes two bend points in the topmost edge, which can be eliminated by moving up the first port of node 1 as shown in Figure 2.8(b). Port placement methods that target edge straightening would have to be realized either as part of the node placement phase or as a post-processing. This problem is left for future research.

Fixed Ratio \rightarrow Fixed Position. The case $\text{PC}(v) = \text{FIXEDRATIO}$ only makes sense if the layout algorithm is allowed to modify the size of v . Let p be a port with initial position (x, y) relative to the upper left corner of v . If the height of v is scaled by λ_h and p is on the east or west side, the vertical position of p is scaled accordingly to $y' = \lambda_h y$. This behavior can

2. The Layer-Based Approach

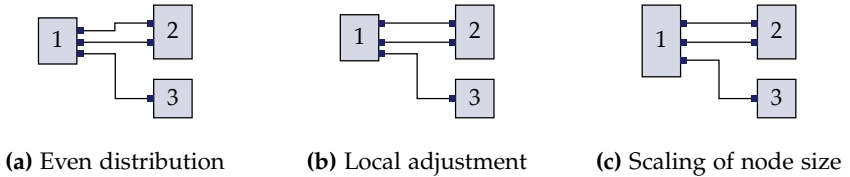


Figure 2.8. Different methods for port placement: (a) even distribution on the node sides, (b) local adjustments to eliminate edge bends, and (c) scaling of node height with `FIXEDRATIO` port constraints.

be exploited for minimizing the number of edge bends: in Figure 2.8(c), the height of node 1 is increased such that the distance between the two topmost ports equals the port distance of node 2, eliminating the bends of the connecting edges that are seen in Figure 2.8(a).

An example for a modeling language that supports `FIXEDRATIO` constraints is Simulink (The MathWorks, Inc.). Klauske and Dziobek presented an ILP-based algorithm for minimizing the number of edge bends in Simulink diagrams [KD10, KD11, Kla12]. Their approach is an extension of the node placement ILP proposed by Gansner et al. [GKNV93]. This extension does not only determine vertical positions for the nodes, but also modifies their height in order to find an optimal placement of ports.

2.2.2 Crossing Minimization

Extending the crossing minimization phase to consider port constraints is crucial, since the number of crossings does not only depend on the order of nodes, but also on the order of ports for each node. The barycenter heuristic used in the layer sweep algorithm, which is introduced in Section 2.1.3, can be modified such that it includes the port order in its computations [SFvHM10].

Definition 2.4 (Port-based barycenter value). Let L_a be the fixed layer and L_b be the free layer to be reordered. For each node $v \in L_a$, let $P'(v) \subseteq P(v)$ be the subset of ports that have connections to nodes in the free layer L_b . For each port $p \in P'(v)$ we require a *rank* value $r(p)$ (Definitions 2.6 and 2.7).

2.2. Port Constraints

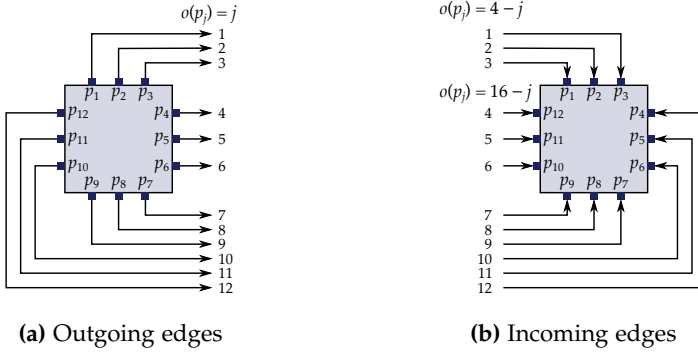


Figure 2.9. Edge order for outgoing edges (for forward layer sweeps) and incoming edges (for backward layer sweeps) according to Definition 2.5. Outgoing edges are ordered clockwise starting with the leftmost north-side port, while incoming edges are ordered counter-clockwise starting with the rightmost north-side port. The port numbering p_1, \dots, p_{12} corresponds to the order of outgoing edges. For inverted ports, i. e. west-side ports in (a) and east-side ports in (b), two alternative routings are feasible, either above or below the node, but this cannot be decided locally (see Section 2.2.3).

The *port-based* forward barycenter value of a node $w \in L_b$ is

$$b(w) = \frac{1}{|E_i(w)|} \sum_{e \in E_i(w)} r(p_s(e)) .$$

The port-based backward barycenter value is defined symmetrically by using the outgoing edges $E_o(w)$ instead of $E_i(w)$ and the target ports $p_t(e)$ instead of $p_s(e)$.

This definition replaces the barycenter values of Definition 2.2. The resulting values are used for sorting the nodes of L_b in the same way as for graphs without ports.

Port ranking. The remaining question is how to determine the rank $r(p)$ for each port p in L_a . Waddle proposed to use the actual coordinates of the ports [Wad01], but that can only be done if the port constraints are

2. The Layer-Based Approach

set to `FIXEDPOS`, since for the other constraint levels the port coordinates are set after the crossing minimization phase has finished. Another option is to determine an index of p considering all other ports in the layer L_a [SFvHM10], which we call the *layer-total* approach. At first we assume all port orders to be fixed. As a convention, we assume these orders to be clockwise, starting with the leftmost port on the north side of each node. As shown in Figure 2.9(a), this convention corresponds to the expected order of outgoing edges of the fixed layer (which are incoming edges of the free layer as used for computing the forward barycenter), hence it can be applied to forward layer sweeps. Backward sweeps are based on the incoming edges of the fixed layer and require a different order, namely counter-clockwise starting with the rightmost port on the north side (see Figure 2.9(b)).

Definition 2.5 (Range of port ranks and edge order with fixed port order). Let $v \in L_a$. The *range* of port ranks with fixed port order occupied by v is $\text{range}(v) = |P'(v)|$. Let $P'(v) = \{p_1, \dots, p_{|P'(v)|}\}$ and let m be the greatest index such that p_1, \dots, p_m are all assigned to the north side of v . The *edge order* with fixed port order is induced by a permutation o of the ports of v : for forward layer sweeps we define $o(p_j) = j$ for all $j \leq |P'(v_i)|$; for backward layer sweeps $o(p_j) = m - j + 1$ if $j \leq m$, and $o(p_j) = m + |P'(v_i)| - j + 1$ otherwise.

The range and edge order values are used for computation of port ranks.

Definition 2.6 (Layer-total rank). Let $v_i \in L_a = \{v_1, \dots, v_{|L_a|}\}$ be a node in the ordered layer and let $p_j \in P'(v_i) = \{p_1, \dots, p_{|P'(v_i)|}\}$ be a port in the ordered port set of v_i . The *layer-total rank* of p_j is

$$r_{\text{LT}}(p_j) = \left(\sum_{k < i} \text{range}(v_k) \right) + o(p_j) .$$

The layer-total rank has unique integer values for all ports in L_a . With this kind of ranking, nodes with many ports occupy a greater range of ranks than nodes with few ports. An alternative is to assign each node an equal range of 1, as done in the following definition. An example for both ranking methods is shown in Figure 2.10.

2.2. Port Constraints

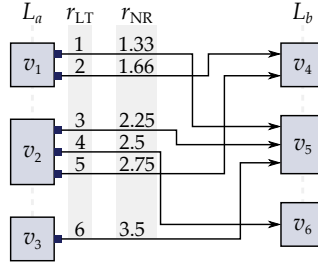


Figure 2.10. Ranks computed with the layer-total (r_{LT}) and the node-relative (r_{NR}) methods. The barycenter values for the layer-total ranks are $b(v_4) = 3.5$, $b(v_5) = 3.33$, and $b(v_6) = 4$, which results in the node order (v_5, v_4, v_6) . In contrast, the node-relative ranks produce $b(v_4) = 2.21$, $b(v_5) = 2.36$, and $b(v_6) = 2.5$, resulting in the node order (v_4, v_5, v_6) . With both variants the drawing has 5 edge crossings.

Definition 2.7 (Node-relative rank). Let v_i and p_j be as in Definition 2.6. The *node-relative rank* of p_j is

$$r_{NR}(p_j) = i + \frac{o(p_j)}{\text{range}(v_i) + 1} .$$

The port ranking methods described above can be adapted to port constraint levels that do not imply a specific order of ports. The basic idea is to create groups of ports of which the order can be chosen freely, and to assign the same rank value to all members of a group. As described in Section 2.2.1, all nodes with constraint level `FREE` are set to `FIXEDSIDE` before or after phase one of the layer-based approach, hence we only need to consider the case `FIXEDSIDE` as alternative to the constraint levels with fixed port order. Since in our model each node can be assigned an individual port constraint, the layer-total rank must be extended such that the range of ranks occupied by a node can be different from that of other nodes.

Given a node $v \in L_a$ for which $PC(v) = \text{FIXEDSIDE}$, all ports $p \in P'(v)$ that are assigned to the same side of v are also given the same rank value. The four sides $\{s_1, s_2, s_3, s_4\}$ are ordered in the same way as already done for fixed-order constraints (see Figure 2.9). For forward layer sweeps it is

2. The Layer-Based Approach

$s_1 = \text{north}$, $s_2 = \text{east}$, $s_3 = \text{south}$, and $s_4 = \text{west}$, while for backward layer sweeps it is $s_1 = \text{north}$, $s_2 = \text{west}$, $s_3 = \text{south}$, and $s_4 = \text{east}$.

Definition 2.8 (Range of port ranks and edge order with `FIXEDSIDE`). Let $v \in L_a$. We denote the set of non-empty sides of v with S_v , i. e. $s \in S_v$ if there exists a port $p \in P'(v)$ such that $\text{side}(p) = s$. The range of port ranks with `FIXEDSIDE` constraints is $\text{range}(v) = |S_v|$. For each side s let $\sigma(s) = 1$ if $s \in S_v$ and $\sigma(s) = 0$ otherwise. Let $p \in P'(v)$ be a port and $\text{side}(p) = s_j$ be its assigned side (j is the side index as described above). The edge order for `FIXEDSIDE` constraints is induced by

$$o(p) = \left(\sum_{k < j} \sigma(s_k) \right) + 1 .$$

By applying these new definitions of the range and edge order functions to Definitions 2.6 and 2.7 we obtain new versions of the layer-total and node-relative ranking methods that assume an arbitrary order of ports on each side.

Counting crossings. In order to effectively use the port-sensitive barycenter heuristic in the layer sweep algorithm for crossing minimization, we need a method for counting the number of crossings with proper consideration of port orders (see Section 2.1.3). In a properly layered graph two edges can only cross if their source nodes are in the same layer, which is equivalent to the condition that their target nodes are in the same layer. As a consequence, the total number of crossings can be determined as the sum of the crossings counted for each pair L_a, L_b of consecutive layers.

Let \mathcal{A} be an algorithm for counting the number of crossings of edges connecting nodes in L_a with nodes in L_b . We can extend \mathcal{A} to consider port constraints by replacing each node v with fixed port order by a set of nodes $v_1, \dots, v_{|P(v)|}$ according to the ports $P(v) = \{p_1, \dots, p_{|P(v)|}\}$. In a similar way, we replace each node v' with `FIXEDSIDE` constraint by nodes v'_n, v'_e, v'_s, v'_w representing the groups of ports located on each of the four sides of v' . After this transformation we execute \mathcal{A} , possibly resulting in a higher number of crossings as compared to the unmodified version.

2.2. Port Constraints

For any pair of nodes v, v' in the same layer L let $\sigma(v, v') = 1$ if the index of v is less than the index of v' in the given order of L , and $\sigma(v, v') = -1$ otherwise. Without taking ports into account, two edges $e = (v_1, v_3)$ and $e' = (v_2, v_4)$ cross if and only if $\sigma(v_1, v_2) \neq \sigma(v_3, v_4)$. Hence the number of crossings for layers L_a, L_b can be determined by comparing the node orders for each pair of edges, which takes $\mathcal{O}(|E_{L_a, L_b}|^2)$ time, where $E_{L_a, L_b} \subseteq E$ is the subset of edges between L_a and L_b . However, there are more efficient algorithms, e. g. as reported by Barth et al. [BMJ04].

Sorting ports. Nodes for which the order of ports is not prescribed have to be processed after an ordering of all layers has been determined. The goal is to find an order of ports with minimal number of edge crossings. Siebenhaller presented an approach for ordering free edges at nodes that can also have fixed edges by transforming the problem into a flow network and finding a minimum cost flow [Sie09, Section 4.5.1.2]. In our model of port constraints this mixed scenario is not allowed, thus the ports of a node are all subject to the same ordering constraint. According to Section 2.2.1 only the case `FIXEDSIDE` has to be considered for crossing minimization, since for nodes with fixed port order the port ordering step is skipped, and the `FREE` constraint level is processed earlier. In order to find a suitable ordering of ports we apply an adapted variant of the barycenter heuristic [SFvHM10].

Definition 2.9 (Port-local barycenter value). Let v be a node with `FIXEDSIDE` constraint. Each port $p \in P(v)$ is assigned a *port-local barycenter value* $b(p)$ defined by

$$b(p) = \frac{1}{|E(p)|} \left(\sum_{e \in E_o(p)} r(p_t(e)) - \sum_{e \in E_i(p)} r(p_s(e)) \right) .$$

The ports $P(v)$ are sorted with a primary and a secondary key: the primary key is the side assigned to each port, and the secondary key is the port-local barycenter value. The ranks $r(p_s(e))$ and $r(p_t(e))$ are computed in the same way as previously described for the ordering of layers, i. e. either with the layer-total or the node-relative approach. Note that the

2. The Layer-Based Approach

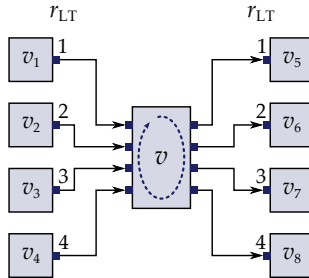


Figure 2.11. Ranks of ports connected by outgoing edges of v (right layer) conform to the clockwise order of ports around v . Ranks of ports connected by incoming edges (left layer), however, are contrary to that clockwise order.

barycenter computation for ports considers ranks for incoming edges as well as outgoing edges. The ranks of incoming edges are considered with a negative sign, because the order of the corresponding ports is contrary to the convention that the ports of v are ordered clockwise around v (see Figure 2.11). Usually a port has either incoming or outgoing edges, but not both,³ hence one of the two sums in Definition 2.9 is zero. Since fixing the order of $P(v)$ influences the ranks of these ports, care must be taken to properly update the rank values. The process of sorting ports for the whole graph is sketched in Algorithm 2.1.

A problem with the approach of using the ranks of both the preceding and the subsequent layer in the barycenter computation is that the rank values of the two layers are determined independently of each other, and thus it makes little sense to compare them with each other in the sorting algorithm. We solve this problem using preprocessing techniques that are described in the following section. As a consequence of this preprocessing, all outgoing edges of v are incident to ports on the east side, and all incoming edges are incident to ports on the west side. This property ensures that only ranks of ports from the same layer are compared with each other by the sorting algorithm.

³As shown in Section 2.2.3, cases where this assumption does not hold are eliminated through preprocessing.

Algorithm 2.1. Sorting ports

Input: a graph with layers L_1, \dots, L_k

```

for  $i = 1 \dots k$  do
  if  $i < k$  then
    Compute ranks of the ports in  $L_{i+1}$ 
  for each  $v \in L_i$  do
    if  $PC(v) = \text{FIXEDSIDE}$  then
      for each  $p \in P(v)$  do
        Compute the barycenter  $b(p)$ 
      Sort  $P(v)$  by assigned sides and barycenter values
       $PC(v) \leftarrow \text{FIXEDORDER}$ 
    // Rank values may now be different due to updated constraints.
  Recompute ranks of the ports in  $L_i$ 

```

2.2.3 Edge Routing

The way the routing of an edge e incident to a port p should be handled depends on the side of p and whether e is an incoming or an outgoing edge. We call p a *regular* port if either $\text{side}(p) = \text{east}$ and $E_i(p) = \emptyset$, or $\text{side}(p) = \text{west}$ and $E_o(p) = \emptyset$. For instance, all ports in Figure 2.11 are regular, which means that they conform to the left-to-right orientation of edges. We call p an *inverted* port if either $\text{side}(p) = \text{east}$ and $E_i(p) \neq \emptyset$, or $\text{side}(p) = \text{west}$ and $E_o(p) \neq \emptyset$. If all ports are regular, we can apply standard routing methods as described in Section 2.1.5. If we have inverted ports or north/south-side ports, however, the standard methods are not sufficient because additional bend points are required (see Figure 2.9).

The first contribution for handling north/south-side ports was done by Sander [San94]. There the affected edges (called left/right anchored edges) of a node v are replaced by dummy nodes that are constrained to be placed next to v , hence the edges are routed locally around v . Similar methods are employed by Schreiber [Sch01, Section 7.1.5] and Siebenhaller [Sie09, Section 4.5.2]. We call this the *local* approach for routing with port constraints. As the term suggests, the approach restricts the routing of

2. The Layer-Based Approach

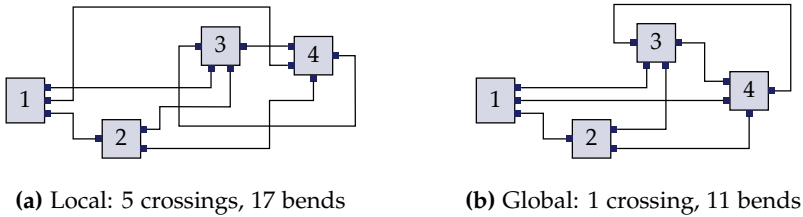


Figure 2.12. Two approaches for routing edges subject to port constraints: (a) the *local* approach reserves an exclusive area around each node without regard to the structure of the graph, while (b) the *global* approach generates dummy nodes that can be placed with constraints that are less strict, and thus enables solutions with fewer edge crossings and bends.

edges to a specific area surrounding the node v and does not take the global graph structure into account. Such a local routing is also implied by the edge order function $o(p)$, $p \in P(v)$, defined in Section 2.2.2. Figure 2.9 reveals that, according to this order function, edges incident to inverted ports are always assumed to be routed below the node. This is not always a good choice, since in some cases routing above the node would yield a more readable drawing, as illustrated in Figure 2.12. Furthermore, the local approach does not allow other nodes to be placed inside the reserved routing area surrounding v .

Schulze proposed a *global* routing approach [Sch11, KSSvH12], which is based on the idea of representing edge segments by dummy nodes that can be placed with certain constraints during the crossing minimization phase. This approach allows a more flexible arrangement of edge segments in order to increase the freedom for minimizing edge crossings and bends. The graph shown in Figure 2.12 has 5 crossings and 17 bends when drawn with the local approach, but only 1 crossing and 11 bends when drawn with the global approach. This is mainly due to two properties of the global method that are exploited in the example: the feedback edge (4,3) is drawn above instead of below the nodes, and the edge (1,4) intersects the area between node 3 and the bend point of edge (2,3). The local approach does not allow that because it constraints bend points of north/south side ports to be put directly beneath their nodes.



(a) An edge connected to inverted ports (b) Dummy nodes inserted into (u, v)

Figure 2.13. An edge (u, v) connecting inverted ports is split with dummy nodes w_l and w_r . The new edges $e_l = (u, w_l)$ and $e_r = (w_r, v)$ are in-layer edges.

More details on Schulze’s global routing approach are given in the remainder of this section.

Inverted ports. The basic scheme for handling inverted ports is illustrated in Figure 2.13: given an edge $e = (u, v)$ for which the source port $p_s(e)$ is inverted, a dummy node w_l is inserted in the same layer as u , and e is split into $e_l = (u, w_l)$ and $e_m = (w_l, v)$. If the target port $p_t(e)$ is inverted, a dummy node w_r is inserted in the same layer as v , and e is split into $e_m = (u, w_r)$ and $e_r = (w_r, v)$. If both $p_s(e)$ and $p_t(e)$ are inverted, as shown in Figure 2.13(b), the edge sequence replacing e is $e_l = (u, w_l)$, $e_m = (w_l, w_r)$, and $e_r = (w_r, v)$. As a result of this preprocessing, the new edge e_m can be treated as a regular edge. However, e_l and e_r have their source and target in the same layer, which breaks the general requirement of a proper layering introduced in Section 2.1.2. We call this new kind of edges *in-layer* edges.

While in-layer edges can be ignored in the node placement phase (phase 4), and it is straightforward to include them in the orthogonal edge routing (phase 5), more intricate adaptations are necessary for crossing minimization (phase 3). The complexity of these adaptations can be greatly reduced by exploiting the fact that for the processing of inverted ports either the source or the target node of an in-layer edge is a dummy node. Let v be a regular node connected to a dummy node w via an in-layer edge $e = (w, v)$. We have to correct the barycenter computation for v in case of a forward layer sweep with v and w both in the free layer L_i : the normal processing would include the rank $r(p_s(e))$ in the sum computed for $b(v)$ (see Definition 2.4), but that rank would be undefined because only ranks of the fixed layer L_{i-1}

2. The Layer-Based Approach

are determined. The dummy node w , however, has only incoming edges with their source in L_{i-1} , hence the value $b(w)$ can be computed normally. The solution is to use $b(w)$ in place of $r(p_s(e))$ in the computation of $b(v)$. This can be written as follows.

Definition 2.10 (Port-based barycenter value with in-layer edges). Let v be a node in the free layer L_i . The port-based forward barycenter value of v considering in-layer edges is

$$b(v) = \frac{1}{|E_i(v)|} \left(\sum_{\substack{e \in E_i(v), \\ v_s(e) \in L_{i-1}}} r(p_s(e)) + \sum_{\substack{e \in E_i(v), \\ v_s(e) \in L_i}} b(v_s(e)) \right).$$

The backward variant is obtained by using the outgoing edges $E_o(v)$, target ports $p_t(e)$, target nodes $v_t(e)$, and the following layer L_{i+1} instead of the preceding layer L_{i-1} .

Applying Definition 2.10 to the example shown in Figure 2.13(b) would mean that $b(v) = b(w_r)$, since (w_r, v) is the only edge incident to v . The barycenter values can be computed by first processing all dummy nodes of L_i , for which we ignore the in-layer edges, then all remaining nodes. As a result, dummy nodes created for inverted ports are placed near their corresponding regular nodes. After all five phases of the layout algorithm have finished, the dummy nodes are removed in the same way as those created to split long edges, thus the original edges are restored.

An additional extension is necessary regarding the port-local barycenter computation for sorting ports with `FIXEDSIDE` constraints given in Definition 2.9. Let $p \in P(v)$ be an inverted east-side port of v connected to a dummy node w via an in-layer edge. At the time when the ports of v are sorted, the relative position of v and w in their respective layer L is already determined, so let $i(v)$ and $i(w)$ be the indices of these nodes. If $i(w) < i(v)$ we want p to be placed at the top of the east side, and otherwise we want it at the bottom. A simple solution is to first compute the barycenters of all regular ports and then use their minimum and maximum to determine barycenters for the inverted ports.

2.2. Port Constraints

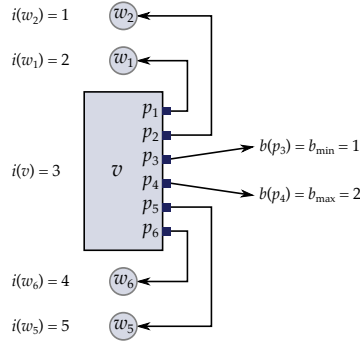


Figure 2.14. Sorting ports with in-layer edges: $b(p_1) = b_{\min} - i(w_1) = -1$, $b(p_2) = b_{\min} - i(w_2) = 0$, $b(p_5) = b_{\max} + |L| + 1 - i(w_5) = 3$, and $b(p_6) = b_{\max} + |L| + 1 - i(w_6) = 4$. Assuming $b(p_3) = 1$ and $b(p_4) = 2$ according to Definition 2.9, we obtain the port order as depicted above.

Definition 2.11 (Inverted port-local barycenter value). Let $v \in L$ be a node and $p \in P(v)$ be an inverted port on the east side of v . Let b_{\min} be the minimum and b_{\max} be the maximum port barycenter value of regular east-side ports of v . Let i_{avg} be the average index of dummy nodes connected to p via in-layer edges (p may have more than one in-layer edge). The *east-side inverted port-local barycenter value* of p is

$$b(p) = \begin{cases} b_{\min} - i_{\text{avg}} & \text{if } i_{\text{avg}} < i(v), \\ b_{\max} + |L| + 1 - i_{\text{avg}} & \text{otherwise.} \end{cases}$$

The handling of west-side inverted ports is symmetric.

As can be seen in Figure 2.14, the effect of the negative sign of i_{avg} is that the order of ports is inverted with respect to the order of dummy nodes, but that is correct if crossings of in-layer edges are to be avoided.

Finally, the algorithm for counting crossings of edges between consecutive layers must be complemented by an algorithm that counts the crossings caused by in-layer edges. As shown by Schulze, this can be done in linear time [Sch11, Section 3.1].

2. The Layer-Based Approach

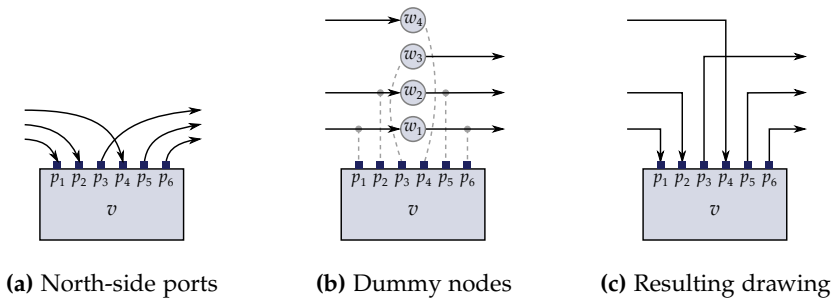


Figure 2.15. Edges connected to the north side are redirected to dummy nodes. In this example four nodes w_1, \dots, w_4 are required, of which w_1 and w_2 are associated with two ports (indicated by the dashed lines in Fig. (b)). In the final drawing, the original edges are restored and the dummy nodes are replaced by bend points.

North/south-side ports. Ports on the north or south side of a node v are handled by adding dummy nodes in order to determine where to draw the necessary bend points. The general idea is illustrated in Figure 2.15 for north-side ports. Each dummy node is associated with either one or two ports. Two ports p_1, p_2 can be assigned to the same dummy node if p_1 has only incoming edges, p_2 has only outgoing edges, and p_1 is placed left of p_2 . The edges are redirected to the generated dummy nodes, hence the node v does not have any connections to the north or south side after this preprocessing. After the five phases of the layer-based algorithm have finished, the original edges are restored and bend points are added at the vertical coordinates that have been assigned to the dummy nodes. More details on the dummy node creation for north/south side ports are given by Schulze [Sch11, Section 3.4].

Let $V_{v,N}$ be the sequence of dummy nodes generated for north-side ports of v and $V_{v,S}$ be the sequence of dummy nodes generated for south-side ports of v . One approach for handling the dummy nodes, called the *fixed-order* approach, requires the relative order of $V_{v,N}$ and $V_{v,S}$ to be preserved by the crossing minimization phase: first $V_{v,N}$ in the given order, then v , then $V_{v,S}$ in the given order. This can be done by extending the layer sweep algorithm, e. g. using the method of Forster [For05]. Furthermore, Schulze

proposed the concept of *layout units* in order to prevent another regular node u or one of its dummy nodes to be placed between the dummy nodes created for v , since that can cause node-edge or edge-edge overlaps. The layout unit of v is $U_v = V_{v,N} \cup \{v\} \cup V_{v,S}$. Every time a layer L is ordered during the execution of the layer sweep algorithm, new node ordering constraints are dynamically inserted and then verified using Forster’s method. Let $u, v \in L$ be regular nodes and let v be the next node following u in the given order of L . Ordering constraints are inserted from all nodes in U_u to all nodes in U_v in order to prevent overlaps between layout units [Sch11, Section 3.3].

When `FIXEDSIDE` constraints are used with the fixed-order approach, the ports on the north and south sides of v have to be sorted before their respective dummy nodes are created. Since these nodes are created before the crossing minimization phase, it is not possible to consider the global graph structure when sorting the ports. As a consequence, they can only be sorted using local information, that is the number of incoming and outgoing edges. Let $\Delta_E(p) = |E_o(p)| - |E_i(p)|$ for each port p , then ports with high Δ_E value should be placed towards the subsequent layer, while those with low Δ_E should be placed towards the preceding layer.

The fixed-order approach fixes the order of dummy nodes and can thereby prevent the crossing minimization phase from finding a globally optimized ordering. A better alternative seems to be to relax the ordering constraints such that the dummy nodes $V_{v,N}$ and $V_{v,S}$ can be ordered arbitrarily, which we call the *variable-order* approach. In this case we still require constraints to ensure that v is placed after $V_{v,N}$ and $V_{v,S}$ is placed after v , plus the constraints derived from layout units. Instead of deriving the dummy node order from the port order, we first apply constrained crossing minimization, then derive the port order from the dummy node order. This approach can only be applied with success if the crossings between edges connected to north/south side ports are included in the total crossing number of a layer sweep, see e. g. the edges connected to p_3 and p_4 in Figure 2.15. This enables the layer sweep algorithm to select the ordering for which the number of crossing is truly minimal. Counting the crossings for a given dummy node order and port order is straightforward: given two north-side edges e_1, e_2 with corresponding dummy nodes w_1, w_2 and ports p_1, p_2 , the edge e_1 crosses the vertical segment of e_2 if w_1 is below w_2 and

2. The Layer-Based Approach

either e_1 is outgoing and p_1 is left of p_2 , or e_1 is incoming and p_1 is right of p_2 . If e_1, e_2 are on the south side, the condition is nearly the same, but w_1 must be above w_2 . Checking these conditions for each pair of edges takes quadratically many computation steps depending on the number of edges on the north and south side of v , but usually that number is rather low.

In order to derive the port order from the dummy node order in the variable-order approach, we assign a port-local barycenter value to each north-side port p with corresponding dummy node $w \in V_{v,N}$: $b(p) = -i(w)$ if p has only incoming edges, $b(p) = i(w)$ if p has only outgoing edges, and $b(p) = 0$ otherwise, where $i(w)$ is the index of w in its containing layer. We treat dummy nodes in $V_{v,S}$ similarly. The resulting barycenter values can be integrated in the sorting process described in Section 2.2.2, where the port side is the primary key and the port barycenter is the secondary key for sorting (see Definition 2.9).

2.2.4 Evaluation

The Ptolemy open source project [EJL⁺03] contains a large number of models for testing and demonstration in its repository.⁴ Ptolemy distinguishes between *atomic actors*, implementing basic functions that are frequently used, and *composite actors*, representing subsystems that can contain other actors. Each composite actor has a specified number of input and output ports for communicating between its content and its surrounding context. Typically each composite actor contains only a medium number of actors, which often can be arranged to fit on one screen, even when the whole model has several hundreds of actors in total.

The layout of composite data flow diagrams is an interesting topic, but it is not addressed here. Therefore the evaluation of the methods presented in the previous section was done on a transformed variant of the Ptolemy demonstration models, where all composite actors were flattened. This was done by moving their contained actors to the outer hierarchy level, connecting them accordingly, and eliminating the composite actors. 216 of the so obtained flattened data flow diagrams were selected for the evaluation. Diagrams that are unsuitable for evaluations were left out, e. g.

⁴<http://ptolemy.eecs.berkeley.edu>

those with very few nodes, or those that have a state machine as top-level actor. The selected diagrams have between 10 and 453 nodes and between 8 and 660 edges; 85% of the diagrams have at most 40 nodes.

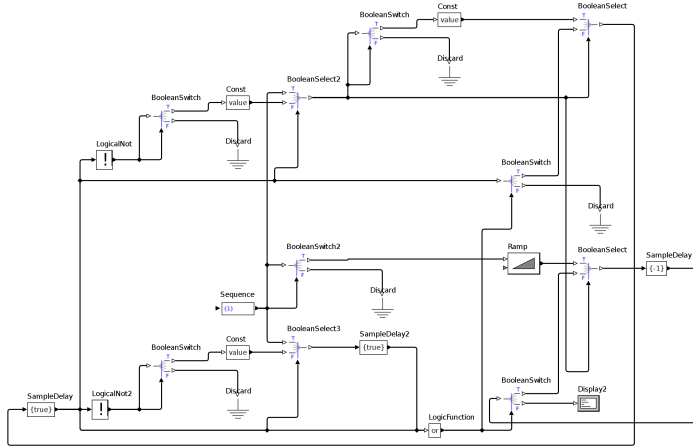
All evaluations involving Ptolemy models have been done with `FIXED-Pos` port constraints, i. e. the relative positions of ports have to be retained by layout algorithms. The approaches presented in this chapter have been implemented in a subproject of KIELER named *KLay* [SSvH14] (see Section 4.4.1). The evaluations were done with node-relative port ranking and variable-order handling of dummy nodes for north/south-side ports. Figure 2.16 illustrates the results of the *KLay* implementation on two flattened diagrams of the Ptolemy demonstration models.

Comparison with Graphviz. In order to demonstrate the importance of considering the prescribed port positions in layout computations, *KLay* was compared with the *Dot* algorithm that is part of the Graphviz tool [GKNV93].⁵ Since Graphviz does not support `FIXEDPos` port constraints, the result of the *Dot* algorithm was modified such that the end points of all edges match the prescribed port positions. This post-processing leads to a high number of edge crossings as well as edges overlapping nodes. When applied to the 216 Ptolemy diagrams mentioned above, the post-processed *Dot* layouts have 27.3 edge crossings and 9.2 edge-node overlaps on average, while the *KLay* layouts have 15.0 edge crossings on average and no edge-node overlaps at all. For 75% of the evaluation diagrams the *KLay* layouts have fewer crossings than the *Dot* layouts, and for 15% the number of crossings is equal. The superior readability of *KLay* layouts for such data flow diagrams can be seen in Figure 2.17.

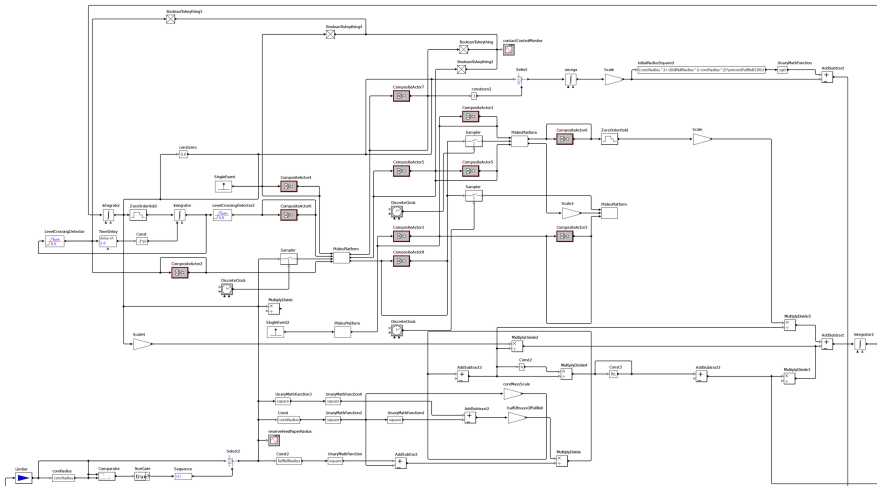
Comparison with local routing. *KLay* implements the global edge routing approach for handling north-south ports and inverted ports as described in Section 2.2.3. The local routing approach has been implemented in a predecessor project named *KLoDD* [Spö09, SFvHM10]. We evaluated the number of edge crossings and the number of edge bends for these two approaches using the flattened Ptolemy models mentioned above. The result

⁵<http://www.graphviz.org/>

2. The Layer-Based Approach



(a) "Stack" model from the *Process Networks* domain (27 nodes, 41 edges)



(b) "T-REX" model from the *PTIDES* project (77 nodes, 115 edges)

Figure 2.16. Flattened diagrams of the Ptolemy demonstration models processed with the KLayout layer-based layout algorithm.

2.2. Port Constraints

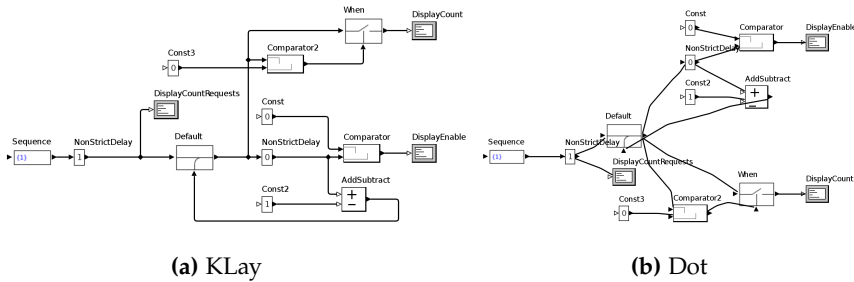


Figure 2.17. Comparison of a layout done using KLayout, which implements the approaches presented here, with a post-processed Graphviz Dot layout on the flattened “GuardedCount” Ptolemy demonstration model. The KLayout layout is obviously more readable, which is due to the proper support of port constraints.

is shown for diagrams with up to 40 nodes in Figure 2.18. It can be clearly seen that the global routing approach outperforms the local approach both in terms of crossings (20% fewer on average) and in terms of bends (22% fewer on average). Regarding statistical significance, t -tests with paired samples result in p -values of 1.9×10^{-4} for the number of crossings and 5.5×10^{-10} for the number of bends. While KLayout achieved zero crossings for 55% of the diagrams, only 32% of the KLoDD layouts have no crossings.

Comparison of port ranking approaches. Both the layer-total and the node-relative port ranking approach (see Section 2.2.2) are very effective for crossing minimization with ports. We evaluated them using a set of 570 randomly generated graphs with between 10 and 94 nodes and an average of 2.4 incident edges per node, which corresponds to the average number of edges found in the demonstration models of the Ptolemy project (a similar value can be derived from the statistics for Simulink models reported by Klauske [Kla12, Section 2.1.2]). The port constraints were set to FIXEDPos for all nodes. For each of the random graphs the best result out of 1000 randomized executions of the layer sweep algorithm was chosen. The average number of edge crossings applying the layer-total ranking approach was $c_{LT} \approx 58.81$, while with the node-relative approach

2. The Layer-Based Approach

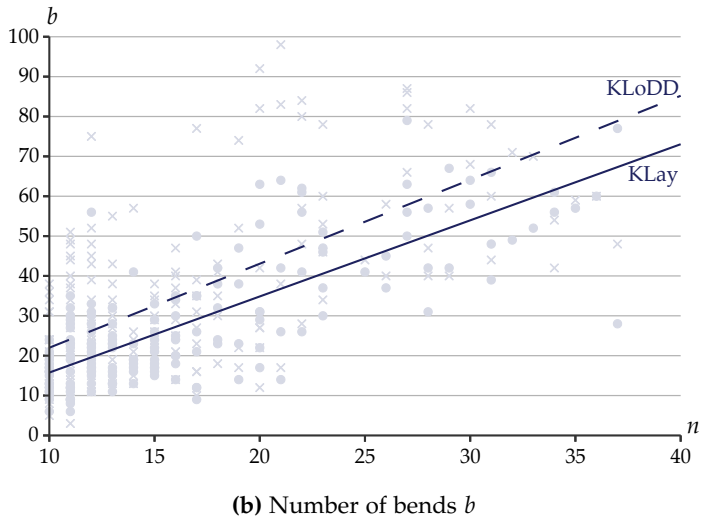
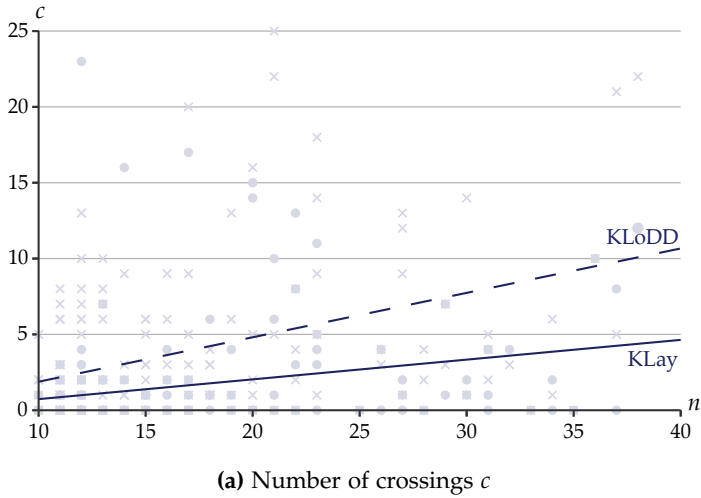


Figure 2.18. Comparison of the number of edge crossings and the number of edge bends between the KLAY algorithm with a global routing approach (gray circles, solid lines) and the KLoDD algorithm with a local routing approach (gray crosses, dashed lines). The horizontal axis represents the number of nodes n in the flattened Ptolemy models used for the evaluation. The lines illustrate linear regression for the measurements.

2.2. Port Constraints

Table 2.1. Average results of the layer-total (c_{LT}), node-relative (c_{NR}), and combined (c_{rand}) port ranking methods, with standard deviations in brackets. The comparison of c_{rand} with the other two methods comprises the relative improvement, the p -value resulting from a t -test with paired samples, and the conclusion on statistical significance.

	Random graphs	Ptolemy diagrams
c_{LT}	58.81 [54.9]	24.11 [88.4]
c_{NR}	59.35 [55.0]	23.70 [88.4]
c_{rand}	58.62 [54.6]	22.97 [82.9]
<i>Comparison of c_{rand} and c_{LT}</i>		
Improvement	0.3%	4.7%
p -value	13.8%	4.7%
Significant	no	yes
<i>Comparison of c_{rand} and c_{NR}</i>		
Improvement	1.2 %	3.1%
p -value	0.004%	18.1%
Significant	yes	no

the value $c_{NR} \approx 59.35$ was measured. Applying the same comparison to the flattened Ptolemy diagrams (excluding those for which the number of crossings is zero regardless of the chosen ranking method) yields a different perspective: $c_{LT} \approx 24.11$ and $c_{NR} \approx 23.70$. We cannot derive a clear winner from these results, since for one set of diagrams $c_{LT} < c_{NR}$ and for another set $c_{LT} > c_{NR}$.

A third possible variant for computing port ranks is to randomly choose one of the two proposed ranking methods in each execution of the layer sweep algorithm. The resulting node order is then taken from the execution with minimal number of crossings. The layouts obtained with this variant have fewer crossings on average: $c_{rand} \approx 58.62$ for the randomly generated graphs and $c_{rand} \approx 22.97$ for the Ptolemy diagrams. Although the improvement of mean values compared to using only one of the ranking methods is rather small, it is partly significant. The relative improvements

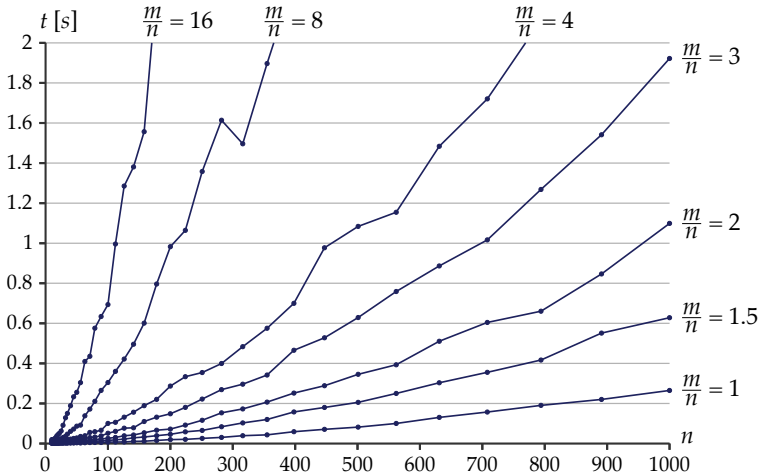
2. The Layer-Based Approach

and t -test results are listed in Table 2.1. In summary, I would recommend using the combined port ranking variant, i. e. using both the layer-total and the node-relative methods and taking the best result. This partly contradicts a previous conclusion [SSvH14], where statistical significance was not considered.

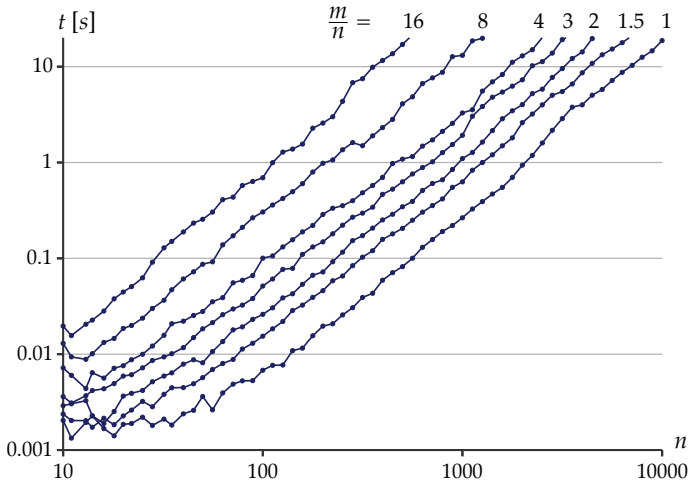
Execution Time. The theoretical time complexity of the layer-based algorithm with the extensions presented here can hardly be determined analytically because it is influenced by the generated dummy nodes, which are not only used for splitting long edges, but also for handling ports with prescribed sides. Eiglsperger et al. gave an analysis of the complexity caused by long edge dummy nodes and proposed to eliminate them in order to process very large graphs [ESK05]. Here the time complexity is investigated by measuring the actual execution time of K Lay with multiple series of randomly generated graphs. Each series of graphs was assigned a fixed ratio $\frac{m}{n}$ of the number of edges to the number of nodes, while the number of nodes n was variable. For each generated graph, the minimal execution time of five subsequent invocations was taken in order to reduce the influence of the operating system and the memory cache. For each value of n , the average time of five random graphs was determined. All edges were connected through ports, where 70% of the ports were regular, 20% were on the north or south side, and 10% were inverted.⁶ All port constraints were set to `FIXEDPos`. Executions were performed with a single thread on an Intel Xeon 2.5 GHz processor using a 64 bit Java Virtual Machine. The results are shown in Figure 2.19: sparse graphs with $\frac{m}{n} \leq 2$ can be processed in less than 200 ms for sizes of up to 350 nodes, while denser graphs with $\frac{m}{n} = 16$ require up to 10 s for the same sizes. Assuming a polynomial complexity $\mathcal{O}(n^\alpha)$, an average exponent $\alpha \approx 1.74$ can be derived from the slope of the curves in logarithmic scale (Figure 2.19(b)).

Klauske examined a set of 1796 data flow diagrams from the automotive industry [Kla12, Section 2.1.2] and identified an average of 22 nodes and 29 edges per diagram, with standard deviations of 25 and 35, respectively. Graphs of these sizes can be processed within 10 ms with our layout methods

⁶ The actual portion of inverted ports could be higher due to cycle elimination.



(a) Linear scale



(b) Logarithmic scale

Figure 2.19. Measured execution time of the KLayout layer-based layout algorithm, in seconds. The time was measured for random graphs with varying number of nodes n in seven series with different ratios $\frac{m}{n}$ of the number of edges to the number of nodes.

2. The Layer-Based Approach

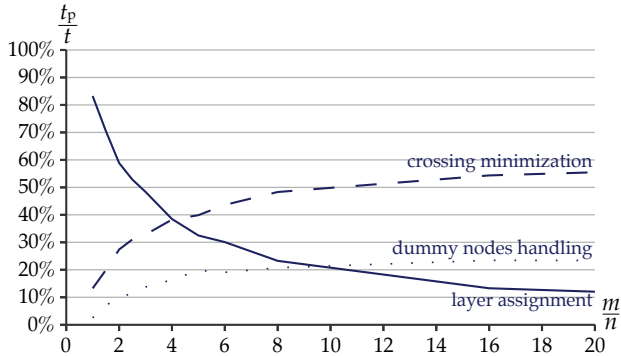


Figure 2.20. Percentage of the execution time t_p of single parts of the layout process relative to the total execution time t . The horizontal axis represents the ratio $\frac{m}{n}$ of the number of edges to the number of nodes.

(time measured for 22 nodes and 33 edges: 2.6 ms), allowing their use in interactive environments where fluidity is a crucial requirement.

Of course there are various factors that determine the execution time, especially the implementation of the five phases of the layer-based approach (Section 2.1). A closer look to the measurement results reveals that two phases accounted for most of the time: layer assignment for sparse graphs, and crossing minimization for dense graphs (see Figure 2.20). An implementation of the network simplex algorithm proposed by Gansner et al. [GKNV93] was used in the layer assignment phase, which is not in the scope of this work. Crossing minimization was done with the usual barycenter heuristic with the port handling extensions of Section 2.2.2. Together these phases took about 80% of the execution time on average. The handling of dummy nodes for long edges and for processing north/south ports and inverted ports (Section 2.2.3) was negligible for sparse graphs and reached 23% for dense graphs.

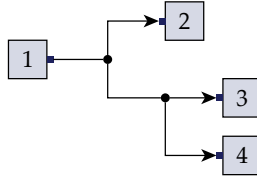


Figure 2.21. A directed hyperedge with source node 1 and target nodes 2, 3, and 4.

2.3 Hyperedges

Directed hyperedges are often used in conjunction with ports, especially in the context of data flow diagrams. A hyperedge h connects a set of source ports $P_s(h)$ to a set of target ports $P_t(h)$, where both port sets are non-empty. Usually hyperedges are drawn in the *tree-based edge standard* [CG07], i. e. they are represented by *junction points* which, together with their connections, form a tree. For instance, the hyperedge shown in Figure 2.21 has one source port, three target ports, two junction points, and five connections.

2.3.1 General Representation

Our general approach for representing a hyperedge h in the layer-based layout algorithm is to replace it by normal edges.

Definition 2.12 (Representing edge). Let (V, H) be a hypergraph and $h = (S, T) \in H$. We call an edge $e = (v, v')$ a *representing edge* of h if $v \in S$, $v' \in T$, $p_s(e) \in P_s(h)$, and $p_t(e) \in P_t(h)$. Given a set of normal edges E , we denote the subset of all representing edges of h with $E_h \subseteq E$.

Definition 2.13 (Representing graph). Let (V, H) be a hypergraph. A graph (V, E) is called a *representing graph* of (V, H) if for all $e \in E$ there is a hyperedge $h \in H$ such that $e \in E_h$ and for all $h \in H$ the graph $(P_s(h) \cup P_t(h), E'_h)$ formed by the source and target ports is connected, where $E'_h = \{(p_s(e), p_t(e)) \mid e \in E_h\}$.

For instance, the hyperedge in Figure 2.21 would be represented by three edges $(1,2)$, $(1,3)$, and $(1,4)$. All edges that are connected to the

2. The Layer-Based Approach

same port are regarded as being part of the same hyperedge, which is a sufficient criterion for identifying hyperedges in our representation, since for any two different hyperedges we generally assume their port sets to be disjoint. In many applications either $|P_s(h)| = 1$ or $|P_t(h)| = 1$ for all hyperedges h , in which case the representing graph is unique: we need exactly $|P_s(h)| + |P_t(h)| - 1$ normal edges to represent each hyperedge h . If multiple sources and multiple targets are allowed, the maximal number of representing edges is $|P_s(h)| \cdot |P_t(h)|$, but the actual number may be lower, as long as the source ports and target ports are connected.

The major benefit of this approach is that it allows to reuse the standard graph-based data structures and most of the algorithms employed in the layer-based drawing approach. More precisely, the first four phases cycle elimination, node layering, crossing minimization, and node placement can be performed with standard algorithms (enhanced by port handling methods as described in Section 2.2) when hyperedges are represented in this manner. The representing edges of one hyperedge may partly overlap each other in the final drawing. All layouts of Ptolemy diagrams generated in Section 2.2.4 have been created with this approach. A more complex solution for representing hyperedges, requiring a dedicated data structure and adapted algorithms, was proposed by Sander [San04].

Edge routing. The fifth and last phase of the layer-based approach, responsible for routing edges, is much more complex when hyperedges are involved. As mentioned in Section 2.1.5, orthogonal edge routing with layers implies that the vertical line segments of edges between each pair of layers must be ordered for obtaining a minimal number of crossings. Eschbach et al. have shown that the vertical segment ordering problem is NP-hard for hyperedges if each hyperedge is constrained to have at most one vertical segment between each pair of consecutive layers [EGB06]. Furthermore, they proposed two heuristics for this problem, one based on greedy assignment and one based on sifting. Sander transformed it to a cycle breaking problem on an auxiliary graph (V^*, E^*) [San04]. Each node in V^* corresponds to a hyperedge, and $(h_1, h_2) \in E^*$ if the line segments of h_1 and h_2 have fewer crossings with each other when the vertical segment of h_1 is drawn left of that of h_2 compared to the inverse order. For instance,

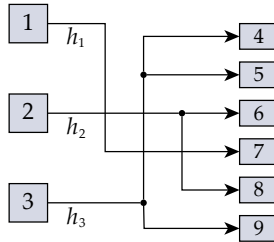


Figure 2.22. Hyperedges may have cyclic dependencies in the auxiliary graph (V^*, E^*) . In this example $E^* = \{(h_1, h_3), (h_3, h_2), (h_2, h_1)\}$, hence we have a cycle $h_1 \rightarrow h_3 \rightarrow h_2 \rightarrow h_1$. No matter how the vertical line segments are ordered, the number of crossings is 4.

the hyperedge h_1 in Figure 2.22 has two crossings with h_2 if h_1 is drawn left of h_2 , but only one crossing if h_1 is drawn right of h_2 . The vertical segments can be ordered by finding a topological order for (V^*, E^*) . However, as observed in Figure 2.22, this auxiliary graph may have cycles, which have to be resolved using a heuristic such as those mentioned in Section 2.1.1. Note that the order and vertical positions of the nodes are fixed, since they are determined in the preceding phases of the layer-based approach.

Merging dummy nodes. If no further measures are taken, the approach of replacing hyperedges by normal edges can lead to layouts like shown in Figure 2.23(a), where the hyperedge represented by the edges $(1,3)$ and $(1,4)$ is assigned two dummy nodes d_1 and d_2 in the second layer in order to obtain a proper layering (see Section 2.1.2). As a consequence, the connections that represent the hyperedge are unnecessarily long. This can be improved by merging adjacent dummy nodes that belong to the same hyperedge as shown in Figure 2.23(b), where the dummy nodes d_1, d_2 have been merged to d' . More details on this method are given by Schulze [Sch11, Section 3.8], cf. [Kla12, Section 3.4.4].

It is possible to apply the merging of dummy nodes immediately after they have been created (after the node layering phase), but that prevents the crossing minimization phase from avoiding crossings caused by the

2. The Layer-Based Approach



(a) Hyperedge with two dummy nodes (b) Hyperedge with one dummy node

Figure 2.23. The hyperedge connecting nodes 1, 3, and 4 is represented by two edges (1,3) and (1,4), which are split by dummy nodes d_1 and d_2 . Merging them to one dummy node d' decreases the total edge length and thus improves readability.



(a) Unmerged, no crossing

(b) Merged, one crossing

Figure 2.24. The merging of dummy nodes of long hyperedges has an influence on the number of crossings: (a) leaving the dummy nodes unmerged allows to draw the hyperedge without crossings, while (b) the merged variant leads to an unavoidable crossing.

hyperedges. For instance, the crossing minimizer algorithm could produce an ordering as shown in Figure 2.24(a), separating the dummy nodes of the two edges between nodes 1 and 3. If the dummy nodes are merged before the crossing minimization phase, an edge crossing is unavoidable, as can be seen in Figure 2.24(b). Depending on the priority given to the aesthetic criteria of edge lengths and edge crossings, the dummy node merging algorithm should be applied before or after crossing minimization: before it if edge lengths have higher priority, and after it if edge crossings have higher priority. In most cases it is probably more desirable to favor the edge crossings criterion, and to merge dummy nodes only if they are adjacent after the nodes of each layer have been ordered.

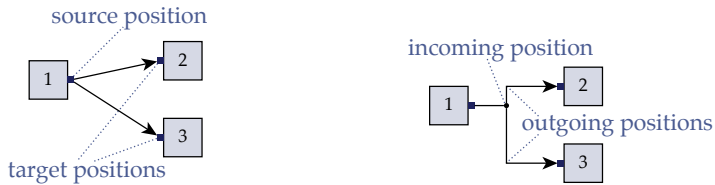
2.3.2 Junction Points

It is important to visualize the junction points of hyperedges, since otherwise it can be hard to distinguish them from edge crossings. The computation of junction point positions can be integrated in the edge routing phase of the layer-based layout algorithm. As mentioned in the preceding section, we represent each hyperedge by a set of normal edges. Let e be a normal edge, h be its corresponding hyperedge, and x_h be the horizontal position assigned to the vertical line segment of h by the edge routing algorithm (see Section 2.3.1). Since vertical node and port positions are already fixed when the edge routing is computed, the source position y_s and target position y_t of e are known. If $y_s \neq y_t$, two bend points (x_h, y_s) (the *incoming* position) and (x_h, y_t) (the *outgoing* position) are added to e (see Figure 2.25). If $y_s = y_t$, the edge e does not require any bend points, but we still assign both an incoming and outgoing position at (x_h, y_s) . Both of these positions are potential candidates for junction points. Let \check{y}_h be the least and \hat{y}_h be the greatest vertical bend point positions of any edge that is part of h . Provided that h is composed of more than one edge, we create a junction point (x_h, y) for each $y \in \{y_s, y_t\}$ if $\check{y}_h < y < \hat{y}_h$ or h contains both an incoming and an outgoing position in y . For instance, the edge $(1, d_1)$ in Figure 2.23(a) has a junction point at its outgoing position because it lies between the vertical bounds \check{y}_h and \hat{y}_h of its hyperedge h . The edge $(2, 4)$ in Figure 2.23(b), in contrast, has a junction point at its incoming position because $(2, 3)$, which belongs to the same hyperedge, has an outgoing position with the same value.

The layouts of the flattened Ptolemy diagrams shown in Figure 2.16 (p. 68) have been created with the junction point computation method described above. The larger diagram in Figure 2.16(b) has 55 hyperedges represented by 115 normal edges, hence each hyperedge is represented by approximately 2 normal edges on average.

Hypernodes. Some modeling environments, e. g. Ptolemy [EJL⁺03], have a concept of *hypernodes* (called *relation vertices* in Ptolemy), which are hyperedge junction points that are modeled explicitly by the user, in contrast to such that are implicitly computed by the modeling tool. If it is acceptable

2. The Layer-Based Approach



(a) A hyperedge with one source and two targets (b) Orthogonal drawing with a junction point

Figure 2.25. Computation of bend points and junction points for orthogonally drawn hyperedges: (a) A hyperedge represented by two normal edges; (b) drawing with two bend points for each representing edge and a junction point at their common incoming position.

to have the layout algorithm add or remove hypernodes, a straightforward approach for optimizing them is to remove all hypernodes and then create a new hypernode for each junction point computed by the algorithm as described above. If such a modification of the model is not acceptable, the hypernodes can be regarded as regular nodes in the layout algorithm, hence they are assigned a position, but not added or removed. However, this approach leads to unpleasant layouts like shown in Figure 2.26(a): the hypernode v_h is assigned to the second layer, and the two edges going to the third layer are given an additional junction point in the edge routing phase. I propose a post-processing step to improve this situation by moving hypernodes such that they replace junction points that have been computed during edge routing. A hypernode v_h can have both incoming edges $E_i(v_h)$ from the preceding layer and outgoing edges $E_o(v_h)$ to the subsequent layer. If $|E_i(v_h)| \leq 1$ and $|E_o(v_h)| \leq 1$, there is no junction point to replace, so we leave v_h unchanged. Otherwise we check which side has more edges: if $|E_i(v_h)| \leq |E_o(v_h)|$, we replace the first junction point of the outgoing edges by v_h , otherwise we do the same with the first junction point of the incoming edges. An example for this procedure is shown in Figure 2.26(b). The additional junction point between the second and third layer has been replaced by v_h , yielding a more concise layout.

2.3. Hyperedges

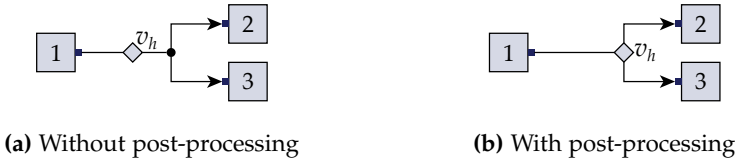


Figure 2.26. Treating hypernodes as regular nodes can lead to unpleasant layouts, since additional junction points are created. This can be improved by moving the hypernodes to one of the junction points in a post-processing step.

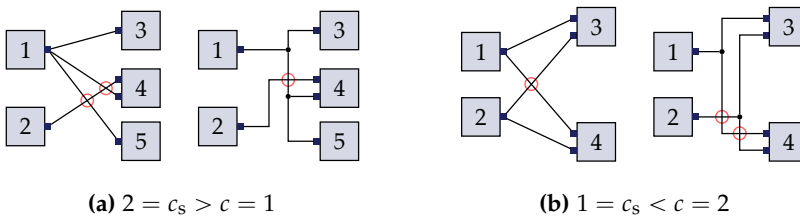


Figure 2.27. The number of crossings c_s resulting from a straight-line drawing can be (a) greater or (b) less than the actual number of crossings c resulting from an orthogonal hyperedge routing.

2.3.3 Counting Crossings

An integral part of the layer sweep heuristic for crossing minimization is an algorithm for counting crossings (Section 2.1.3). Such algorithms usually assume that all edges are drawn as straight lines. A fundamental problem with hyperedges is that the actual number of crossings does not only depend on the order of nodes in each layer, but also on their routing between the layers. This routing in turn depends on the concrete positioning of the nodes, which is unknown at the time the crossing minimization heuristics are executed. The inevitable consequence is that those heuristics work with unreliable crossing numbers, possibly compromising the quality of their results. In this section we discuss approaches to improve this situation [SSRvH14a, SSRvH14b].

We write STRAIGHT to denote a standard straight-line method for counting crossings, and denote its result as c_s . As noted by Eschbach et al.

2. The Layer-Based Approach

[EGB03], there are simple examples where c_s is always different from the actual number of crossings c obtained after applying the usual orthogonal routing methods (see Figure 2.27). In order to quantify this difference, we measured c and c_s for a number of data flow diagrams from the Ptolemy project (see Section 2.2.4). The difference $c - c_s$ averaged -34 with a standard deviation of 190 . There are some examples where the difference amounts to extreme values; one diagram with 194 hyperedges even reaches $c = 269$ and $c_s = 2216$. As a general observation, the STRAIGHT heuristic tends to overestimate the crossing number.

In the following, let $G = (V, H)$ be a hypergraph with a representing graph (V, E) and two layers L_1, L_2 , i.e. $V = L_1 \cup L_2$, $L_1 \cap L_2 = \emptyset$, and all $h \in H$ have their sources in L_1 and their targets in L_2 . Let $\pi_1 : L_1 \rightarrow \{1, \dots, |L_1|\}$ and $\pi_2 : L_2 \rightarrow \{1, \dots, |L_2|\}$ be the permutations of the layers L_1 and L_2 that result from the layer sweep heuristic for crossing minimization.

Lower bound method. Since counting straight-line crossings tends to yield rather pessimistic estimates when hyperedges are involved, we assumed that a more accurate approach might be to use a lower bound of the number of crossings. I propose an optimistic method MINOPT and denote its result as c_m . This method counts the minimal number of crossings to be expected by evaluating each unordered pair $h_1, h_2 \in H$: if any edge e_1 that represents h_1 crosses an edge e_2 that represents h_2 if drawn as a straight line, h_1 and h_2 are regarded as crossing each other once, denoted as $h_1 \bowtie h_2$.

Definition 2.14 (MINOPT heuristic). The crossings number determined by MINOPT is

$$c_m = |\{\{h_1, h_2\} \subset H : h_1 \bowtie h_2\}| .$$

Theorem 2.15. $c_m \leq c_s$.

Proof. Let $e_1, e_2 \in E$ cross each other when drawn as straight lines. There are unique $h_1, h_2 \in H$ such that e_1 represents h_1 and e_2 represents h_2 . By definition of the MINOPT method, h_1 and h_2 cross each other. Hence there is a mapping $\alpha : \{e_1, e_2 \in E : e_1 \bowtie e_2\} \rightarrow \{h_1, h_2 \in H : h_1 \bowtie h_2\}$ that is surjective because for each hyperedge crossing there is at least one crossing of representing edges. This implies $c_s = |\{e_1, e_2 \in E : e_1 \bowtie e_2\}| \geq |\{h_1, h_2 \in H : h_1 \bowtie h_2\}| = c_m$. \square

2.3. Hyperedges

Theorem 2.16. *Let D be a layer-based drawing of G and c be the corresponding number of hyperedge crossings. Then $c_m \leq c$.*

Proof. Let $h = (S, T)$ and $h' = (S', T')$ cross each other as determined by MINOPT. Then there are $v \in S$, $w \in T$, $v' \in S'$, and $w' \in T'$ such that $(v, w), (v', w') \in E$ and $(v, w), (v', w')$ cross each other. Without loss of generality let $\pi_1(v) < \pi_1(v')$ and $\pi_2(w) > \pi_2(w')$. The representation $D(h)$ of h in the drawing D must connect the representations $D(v)$ and $D(w)$. This connection is not possible without crossing $D(h')$, which must connect $D(v')$ and $D(w')$, since $D(v')$ is below $D(v)$, $D(w')$ is above $D(w)$, and both $D(h)$ and $D(h')$ are inside the area between the two layers. Consequently, each crossing counted by MINOPT implies at least one crossing in D . \square

Theorem 2.17. *Let $q = |H|$ and $H = \{h_1, \dots, h_q\}$. The time complexity of MINOPT is $\mathcal{O}\left(\sum_{i=1}^{q-1} \sum_{j=i+1}^q |E_{h_i}| \cdot |E_{h_j}|\right)$. If $|S| = |T| = 1$ for all $(S, T) \in H$, the complexity can be simplified to $\mathcal{O}(|H|^2)$.*

Proof. The result of MINOPT is $|\{\{h_i, h_j\} \subset H : h_i \bowtie h_j\}|$, which requires to check all unordered pairs $U = \{\{h_i, h_j\} \subset H\}$. This is equivalent to $U = \{(i, j) \in \mathbb{N}^2 : 1 \leq i < j \leq q\}$, hence $|U| = \sum_{i=1}^{q-1} \sum_{j=i+1}^q 1$. Whether $h_i \bowtie h_j$ is determined by comparing all representing edges of h_i with those of h_j , which requires $|E_{h_i}| \cdot |E_{h_j}|$ steps. In total we require $\sum_{i=1}^{q-1} \sum_{j=i+1}^q |E_{h_i}| \cdot |E_{h_j}|$ steps.

If for all $h = (S, T) \in H$ the constraint $|S| = |T| = 1$ holds, we can imply $|E_h| = 1$. In this case the number of steps is $\sum_{i=1}^{q-1} \sum_{j=i+1}^q 1 \leq q^2$, hence the complexity is $\mathcal{O}(q^2) = \mathcal{O}(|H|^2)$. \square

Approximation method. Theorem 2.17 shows that MINOPT has a roughly quadratic time complexity. In this section I propose a second method with better time complexity, called APPROXOPT. The basic idea is to approximate the result of MINOPT by checking three criteria explained below, hoping that at least one of them will be satisfied for a given pair of hyperedges if they cross each other in the final drawing.

2. The Layer-Based Approach

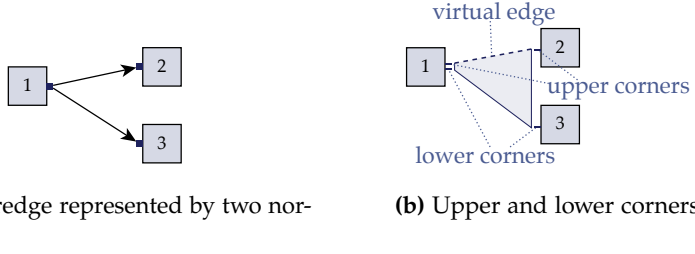


Figure 2.28. Illustration of the four *corners* defined for a hyperedge and the virtual edge between the two upper corners. Here $\kappa_1^\uparrow(h) = 1$, $\kappa_1^\downarrow(h) = 1$, $\kappa_2^\uparrow(h) = 1$, and $\kappa_2^\downarrow(h) = 2$ (note that corners refer to node permutations, not to node labels).

Definition 2.18 (Corners, virtual edges). For each $h = (V_{h,1}, V_{h,2}) \in H$ and $i \in \{1, 2\}$, we define the *upper corners* $\kappa_i^\uparrow(h) = \min \{ \pi_i(v) \mid v \in V_{h,i} \}$ and the *lower corners* $\kappa_i^\downarrow(h) = \max \{ \pi_i(v) \mid v \in V_{h,i} \}$ (see Figure 2.28). The *virtual edges* are defined by $E^* = \{ (\kappa_1^\uparrow(h), \kappa_2^\uparrow(h)) \mid h \in H \}$.

The APPROXOPT method consists of three steps:

1. Compute the number of straight-line crossings caused by virtual edges between the upper corners.
2. Compute the number of overlaps of ranges $[\kappa_1^\uparrow(h), \kappa_1^\downarrow(h)]$ in the first layer for all $h \in H$.
3. Compute the number of overlaps of ranges $[\kappa_2^\uparrow(h), \kappa_2^\downarrow(h)]$ in the second layer for all $h \in H$.

The result c_a of APPROXOPT is the sum of the three numbers computed in these steps. Step 1 checks the first criterion, which aims at “normal” crossings of hyperedges such as h_1 and h_2 in Figure 2.29. The hyperedge corners used in Steps 2 and 3 serve to check for overlapping areas, as shown in Figure 2.29(c). For instance, the ranges spanned by h_4 and h_5 overlap each other both in the first layer (left side) and in the second layer (right side). This is determined using a linear pass over the hyperedge corners,

Algorithm 2.2. Counting crossings with the APPROXOPT method

Input: layers L_1, L_2 with permutations π_1, π_2 and hyperedges H with an arbitrary order ϑ

Output: an approximation for the number of hyperedge crossings

// Step 1

for each $h \in H$ **do**

 | Add $(\kappa_1^\uparrow(h), \kappa_2^\uparrow(h))$ to E^*

$c_a \leftarrow$ number of crossings caused by E^* , counted with a straight-line method

// Steps 2 and 3

for $i = 1 \dots 2$ **do**

 | **for each** $h \in H$ **do**

 | Add $(\kappa_i^\uparrow(h), \kappa_i^\downarrow(h), \vartheta(h), -1)$ to C_i

 | Add $(\kappa_i^\downarrow(h), \kappa_i^\uparrow(h), \vartheta(h), 1)$ to C_i

 | Sort C_i lexicographically

 | $d \leftarrow 0$

 | **for each** $(x, x', j, t) \in C_i$ in lexicographical order **do**

 | **if** $t = -1$ **then**

 | $d \leftarrow d + 1$

 | **else if** $t = 1$ **then**

 | $d \leftarrow d - 1$

 | $c_a \leftarrow c_a + d$

return c_a

where a variable d is increased whenever a top-side corner is found and decreased whenever a bottom-side corner is found (see Algorithm 2.2). This variable indicates how many ranges of other hyperedges surround the current corner position, hence its value is added to the approximate number of crossings.

While MINOPT counts at most one crossing for each pair of hyperedges, APPROXOPT may count up to three crossings, since the hyperedge pairs are considered independently in all three steps. Figure 2.30(a) shows an

2. The Layer-Based Approach

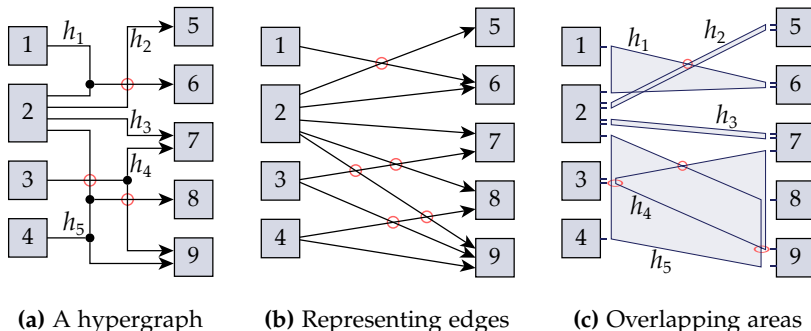


Figure 2.29. The hypergraph (a) can be drawn orthogonally with $c = 3$ crossings. The straight-line crossing number (b) is $c_s = 5$, the result of MINOPT is $c_m = 2$, and the result of APPROXOPT is $c_a = 4$. APPROXOPT counts three crossings between h_4 and h_5 (c) because the virtual edges $(2, 8)$ and $(3, 7)$ cross (Step 1 in Algorithm 2.2) and the ranges spanned by the corners overlap both on the left side and on the right side (Steps 2 and 3).

example where MINOPT counts a crossing and APPROXOPT counts none, while Figure 2.30(b) shows an example where APPROXOPT counts a crossing and MINOPT counts none. These examples show that neither $c_m \leq k \cdot c_a$ nor $c_a \leq k \cdot c_m$ hold in general for any $k \in \mathbb{N}$. However, as determined experimentally in Section 2.3.4, the difference between c_m and c_a is rather small in practice.

Theorem 2.19. Let $b = \sum_{(S,T) \in H} (|S| + |T|)$. The time complexity of APPROXOPT is $\mathcal{O}(b + |H|(\log |V| + \log |H|))$.

Proof. In order to determine the corners $\kappa_i^\uparrow(h), \kappa_i^\downarrow(h)$ for each $h \in H, i \in \{1, 2\}$, all source and target nodes are traversed searching for those with minimal and maximal index π_i . This takes $\mathcal{O}\left(\sum_{(S,T) \in H} (|S| + |T|)\right) = \mathcal{O}(b)$ time. The number of virtual edges created for Step 1 is $|E^*| = |H|$. Counting the crossings caused by E^* can be done in $\mathcal{O}(|E^*| \log |V|) = \mathcal{O}(|H| \log |V|)$ time [BMJ04]. Steps 2 and 3 require the creation of a list C_i with $2|H|$ elements, namely the lower-index and the upper-index corners of all hyperedges. Sorting this list is done with $\mathcal{O}(|C_i| \log |C_i|) =$

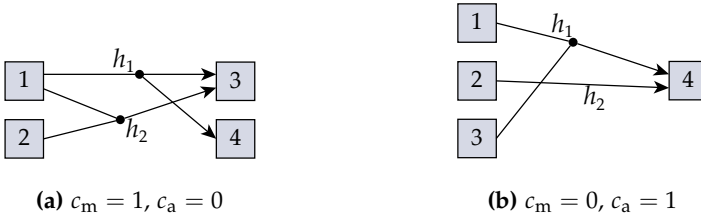


Figure 2.30. Examples revealing the difference between the MINOPT and APPROXOPT methods: (a) $c_m = 1$ due to the crossing of $(1,4)$ and $(2,3)$, but $c_a = 0$ since none of the three steps of APPROXOPT is able to detect the crossing. (b) $c_m = 0$ because $(2,4)$ crosses neither $(1,4)$ nor $(3,4)$; $c_a = 1$ because one crossing is detected in Step 2 of Algorithm 2.2.

$\mathcal{O}(|H| \log |H|)$ steps. Afterwards, each element in the list is visited once. The total required time is $\mathcal{O}(b + |H| \log |V| + |H| \log |H|)$, which is equal to $\mathcal{O}(b + |H|(\log |V| + \log |H|))$. \square

2.3.4 Evaluation

The algorithms for counting crossings were evaluated using the flattened Ptolemy diagrams introduced in Section 2.2.4. The layout algorithm was executed once for each crossings counting algorithm on each of the flattened Ptolemy diagrams. For each execution, the actual number of crossings in the final diagram as well as the number predicted by the three crossings counting algorithms were measured.

The results can be seen in Figure 2.31. The important observation is that the average number of actual crossings is reduced by 23.6% when using MINOPT and by 23.8% when using APPROXOPT instead of STRAIGHT. These differences of mean values are significant: the p -values resulting from a t -test with paired samples are 4.5% for MINOPT and 4.0% for APPROXOPT.

A more detailed view on the experimental results is shown in Table 2.2. The average results of the three counting methods are given for each of the three executions, even if they have not been used in the layer sweep heuristic for crossing minimization during that execution. The table reveals that the accuracy of the counted number of crossings, $|c - c_m|$ and $|c - c_a|$,

2. The Layer-Based Approach



Figure 2.31. Average number of crossings in the resulting drawing when using the given counting algorithm and average number of crossings predicted by that algorithm for the flattened Ptolemy diagrams.

is consistently better with the two methods proposed here compared to the accuracy $|c - c_s|$ obtained with the straight-line method. This does not only apply when comparing the mean values of these differences, but also their standard deviations: the STRAIGHT method leads to more extreme difference values. Furthermore, the difference $|c_m - c_a|$ of the results of the MINOPT and APPROXOPT methods is relatively low, averaging about a third of the total crossing number. This confirms that APPROXOPT yields a good approximation of MINOPT.

The layer sweep heuristic for crossing minimization uses the predicted number of crossings only to compare two possible node orderings with each other. Therefore the predicted values as such are not relevant in this context, but rather their comparison: given two node orderings π_1, π_2 , corresponding predictions $c_{p,1}, c_{p,2}$, and actual numbers of crossings c_1, c_2 , a good prediction must meet

$$\sigma(c_{p,1} - c_{p,2}) = \sigma(c_1 - c_2) , \quad (2.1)$$

where $\sigma : \mathbb{R} \rightarrow \{0, 1, -1\}$ is the sign operator. With the STRAIGHT prediction Equation 2.1 is met in 55% of the cases comparing the values obtained in the three algorithm executions for each graph, which lead to three comparisons per graph (Executions E_s / E_m , E_s / E_a , and E_m / E_a). MINOPT

2.3. Hyperedges

Table 2.2. Average values measured for the flattened Ptolemy diagrams, with standard deviations in brackets. All three methods for counting crossings (c_s , c_m , and c_a) were measured in all three executions, but each execution used only one method for minimizing crossings: c_s in Execution E_s , c_m in Execution E_m , and c_a in Execution E_a . The last column shows the average values over all three executions. All values are normalized by the total average number of crossings $\bar{c} \approx 18.75$.

Variable	Execution E_s (using c_s)		Execution E_m (using c_m)		Execution E_a (using c_a)		Total	
c	1.19	[4.77]	0.91	[3.65]	0.91	[3.95]	1.00	[4.14]
c_s	3.02	[13.94]	3.63	[16.48]	3.66	[16.86]	3.44	[15.79]
c_m	1.12	[4.24]	0.82	[3.33]	0.85	[3.62]	0.93	[3.75]
c_a	1.46	[5.52]	1.17	[4.81]	1.09	[4.77]	1.24	[5.04]
$ c - c_s $	1.90	[10.13]	2.79	[13.49]	2.78	[13.61]	2.49	[12.50]
$ c - c_m $	0.29	[0.69]	0.28	[0.63]	0.26	[0.63]	0.28	[0.65]
$ c - c_a $	0.33	[1.01]	0.36	[1.28]	0.31	[0.94]	0.33	[1.09]
$ c_m - c_a $	0.35	[1.47]	0.34	[1.49]	0.26	[1.17]	0.32	[1.38]

and APPROXOPT performed correctly in 65% and 72% of the comparisons, respectively. These drastic improvements of the ratio of correct comparisons (p -values $< 10^{-6}$ with a t -test) in the context of the layer sweep heuristic explain why the two proposed methods lead to fewer crossings in the actual drawings compared to the straight-line method.

More details on the correctness of comparisons are given in Table 2.3. It can be seen that the correctness rates are extremely different depending on which execution results are compared and which subset of graphs is considered. Each comparison involves two node orderings π_1, π_2 and actual numbers of crossings c_1, c_2 . In general, when constrained to graphs where $c_1 < c_2$, each prediction method M yields high correctness rates if π_1 was determined based on M , but low correctness rates if π_2 was determined based on M . If $c_1 > c_2$ an inverse tendency is observed. For instance, when comparing results made with the STRAIGHT method (Execution E_s) with results made with the MINOPT method (Execution E_m), STRAIGHT has a correctness rate of 74% for graphs with $c_1 < c_2$ (i. e., where the STRAIGHT method yields better results than the MINOPT method), but only 8% for

2. The Layer-Based Approach

Table 2.3. Rate of correctness determined with Equation 2.1 applied to the results of the three executions on Ptolemy diagrams. The execution results are compared in pairs E_s / E_m , E_s / E_a , and E_m / E_a , where Execution E_s used c_s to determine a node order, Execution E_m used c_m , and Execution E_a used c_a . The correctness rates for each prediction method are presented in four rows: three rows constrained to graphs that meet specified criteria and one row showing total average rates for all graphs. The criteria are given in the form $c_1 \sim c_2$, where c_1 and c_2 are the actual numbers of crossing resulting from the first and second compared executions, respectively.

		E_s / E_m	E_s / E_a	E_m / E_a	Total
STRAIGHT	$c_1 < c_2$	74%	82%	79%	
	$c_1 > c_2$	8%	4%	32%	
	$c_1 = c_2$	82%	62%	73%	
	Total	64%	42%	58%	55%
MINOPT	$c_1 < c_2$	46%	66%	73%	
	$c_1 > c_2$	76%	32%	28%	
	$c_1 = c_2$	90%	79%	91%	
	Total	77%	56%	63%	65%
APPROXOPT	$c_1 < c_2$	67%	34%	12%	
	$c_1 > c_2$	82%	92%	91%	
	$c_1 = c_2$	88%	58%	73%	
	Total	82%	67%	68%	72%

graphs with $c_1 > c_2$. This difference is not surprising; the better the drawing created with a particular method is, the higher is the probability that the predictions made by that method were correct.

We performed a second experiment with the same kinds of measurements as for the Ptolemy diagrams, but based on randomly generated bipartite graphs. The algorithms for counting crossings always operate on pairs of consecutive layers, which are bipartite subgraphs, hence the specialization of the experiment to bipartite graphs is valid. Each graph had between 5 and 100 nodes distributed over two layers and between 2 and 319 hyperedges. The results are shown in Figure 2.32 and Table 2.4. These data confirm the general observations made for Ptolemy. The average

2.3. Hyperedges

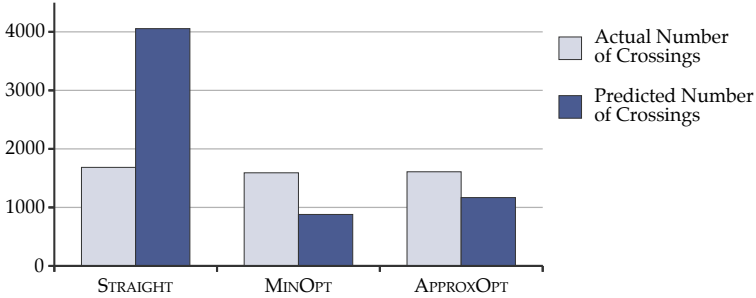


Figure 2.32. Average number of crossings in the resulting drawing when using the given counting algorithm and average number of crossings predicted by that algorithm for the random graphs.

Table 2.4. Average values measured for the random bipartite graphs, with standard deviations in brackets. The table has the same format as Table 2.2. All values are normalized by the total average number of crossings $\bar{c} \approx 1628$.

Variable	Execution E_s (using c_s)	Execution E_m (using c_m)	Execution E_a (using c_a)	Total
c	1.04 [2.00]	0.98 [1.93]	0.99 [1.95]	1.00 [1.96]
c_s	2.49 [4.43]	2.67 [4.68]	2.68 [4.69]	2.61 [4.60]
c_m	0.58 [1.28]	0.54 [1.23]	0.55 [1.24]	0.55 [1.25]
c_a	0.79 [1.69]	0.74 [1.63]	0.72 [1.59]	0.75 [1.64]
$ c - c_s $	1.46 [3.31]	1.69 [3.65]	1.69 [3.64]	1.61 [3.54]
$ c - c_m $	0.46 [0.81]	0.44 [0.78]	0.44 [0.78]	0.45 [0.79]
$ c - c_a $	0.24 [0.47]	0.24 [0.44]	0.27 [0.48]	0.25 [0.46]
$ c_m - c_a $	0.22 [0.43]	0.20 [0.41]	0.17 [0.36]	0.19 [0.40]

number of actual crossings is reduced by 5.6% when using MINOPT and by 4.6% when using APPROXOPT instead of STRAIGHT. Although the relative difference of mean values is lower compared to the Ptolemy diagrams, their significance is much higher: in both cases $p < 10^{-31}$. Compared to the absolute difference $|c - c_s|$ of the straight-line method, $|c - c_m|$ is 72% lower and $|c - c_a|$ is 84% lower.

2. The Layer-Based Approach

Table 2.5. Rate of correctness determined with Equation 2.1 applied to the random bipartite graphs. The table has the same format as Table 2.3.

		E_s / E_m	E_s / E_a	E_m / E_a	Total
STRAIGHT	$c_1 < c_2$	97%	100%	59%	
	$c_1 > c_2$	1%	0%	37%	
	$c_1 = c_2$	66%	62%	70%	
	Total	24%	28%	51%	34%
MINOPT	$c_1 < c_2$	7%	31%	88%	
	$c_1 > c_2$	98%	89%	21%	
	$c_1 = c_2$	73%	70%	81%	
	Total	79%	75%	59%	71%
APPROXOPT	$c_1 < c_2$	21%	5%	18%	
	$c_1 > c_2$	93%	100%	84%	
	$c_1 = c_2$	67%	64%	71%	
	Total	78%	76%	50%	68%

With respect to Equation 2.1, 32% of the comparisons made with the STRAIGHT prediction were correct, while MINOPT and APPROXOPT performed correctly in 71% and 68% of the cases, respectively. It is worth noting that the rate of correct comparisons with the straight-line method is very close to the expected result of a function randomly choosing between 0, 1, and -1 , which would make a correct decision in 33% of the cases. More details are presented in Table 2.5.

2.4 Interactive Layout

When graph layout algorithms are employed in interactive environments, new requirements arise that are not captured by the usual aesthetic criteria. While these criteria describe properties of a single layout, in many situations a user may be confronted with multiple layouts of the same or a similar graph, e. g. when the graph is modified through editing operations, or the view is restricted or extended to a different subset of the whole graph. In

order to facilitate the user's comprehension of the graph at all points in time during this sequence of layouts, it is important to minimize the difference between two consecutive layouts, which is referred to as maintaining *stability* of the layouts [BP90], or preserving the user's *mental map* [ELMS91, PHG06].

Some solutions specifically address the *dynamic* graph layout scenario, where stability must be maintained for a graph that is subject to slight modifications. The layer-based approach has been adapted to this scenario by extending its main phases such that layout stability is considered [BP90, Nor96, NW02]. Other approaches use a difference metric for graph layouts combined with a standard layout method in a meta-heuristic such as an evolutionary algorithm [Bra01, Ism12], or define constraints that preserve the topology of the layout after each editing operation [DMW09].

In this section I present a much simpler approach for maintaining stability, called *sketch-driven* layout. The idea is to take an arbitrary layout of the input graph, called the *sketch*, and to produce a layer-based layout with the same topology as the sketch. A similar method has been proposed by Brandes et al. for the topology-shape-metrics approach by adapting the orthogonalization phase [BEKW02]. The given layout can be created by a human, allowing an interactive scenario where the algorithm reacts to manual layout modifications. Another scenario is to use a completely different layout algorithm to create a sketch, e. g. a force-based or planarization-based algorithm. The sketch-driven method preserves the topology computed by these other layout algorithms and transfers it to a layer-based drawing. This can be useful to obtain drawings with certain properties, e. g. layer-based node placement with few edge crossings. In the following we will look at the adaptations required to realize sketch-driven layout in the layer-based approach.

2.4.1 Layer-Based Sketch-Driven Layout

The layer-based approach has three phases that determine the topology of the layout, i. e. the relative ordering of graph elements, and two phases that set concrete coordinates. Here we consider sketch-driven methods for the topology phases (cycle elimination, layer assignment, and crossing minimization). These methods can be used as replacements for the respec-

2. The Layer-Based Approach

tive static optimization methods according to the *strategy* pattern (see also Section 4.4.1). The final node placement and edge routing phases are applied with standard methods as described in Section 2.1.4 and Section 2.1.5, although sketch-driven methods could be also developed for these phases in future extensions.

Cycle elimination. The basic idea for sketch-driven cycle elimination is simple: reverse all edges that point against the main layout direction in the given sketch (see Algorithm 2.3). The main question here is how to derive the horizontal positions of nodes from the sketch. While this is trivial if nodes are just drawn as points, nodes with a predefined size allow multiple options for choosing a reference point to be compared with that of other nodes:

- the center point,
- one of the corners of the bounding box (e. g. the top left corner), or
- the anchor point where the respective edge touches the node.

It is not obvious a priori which of these options to choose, therefore they should be configurable for the specific application or by the user.

Algorithm 2.3. Sketch-driven cycle elimination

Input: a graph (V, E) with a sketch S
for each node $v \in V$ **do**
 $x_v \leftarrow$ horizontal position of v in S
 for each outgoing edge $(v, w) \in E$ **do**
 $x_w \leftarrow$ horizontal position of w in S
 if $x_w < x_v$ **then**
 Reverse the edge (v, w)

// If multiple nodes have the same horizontal position, cycles can remain.

Search and eliminate cycles with a standard heuristic

2.4. Interactive Layout

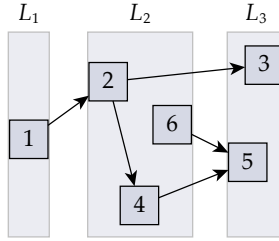


Figure 2.33. Sketch-based layer assignment: nodes 2, 4, and 6 have overlapping horizontal coordinates, hence they are assigned to the same layer L_2 in the first step of Algorithm 2.4. Due to the edge $(2,4)$, the node 4 is moved to L_3 in the second step, forcing node 5 to be moved to a new layer L_4 .

Layer assignment. A layer assignment can be derived from a sketch by requiring that if the area occupied by two nodes overlaps horizontally, they should be assigned to the same layer. The *span* of a layer L is (L, l, r) , where l and r are the minimal and maximal horizontal sketch positions occupied by any node in L , accordingly. Algorithm 2.4 constructs such spans following the principle above. For each node v it either inserts v into an existing span $s = (V_s, l_s, r_s)$ that overlaps with v , or it assigns v to a new span. The resulting layering may contain edges (v, w) , $v \in L_j$ and $w \in L_k$, for which $k \leq j$, violating the requirement of valid layerings (see Definition 2.1). This can be corrected by traversing all nodes v according to a topological sorting and moving all target nodes of violating outgoing edges into the layer directly following the one assigned to v (see Algorithm 2.4). Due to the topological sorting all predecessor nodes of v have already been processed at the time v is processed, hence in subsequent iterations the layer index of v is not modified, and the layer index of successor nodes of v is only increased or left unmodified. Consequently, for all edges (v, w) the layer index of w is higher than the layer index of v after the algorithm has finished. Figure 2.33 shows an example where six nodes are assigned to three layer spans, resulting in four layers after the correction step.

Finding a topological sorting and correcting the layering is done within $\mathcal{O}(|V| + |E|)$ steps. The worst case for deriving layers from the sketch is that the horizontal spans of all nodes are non-overlapping, in which case

2. The Layer-Based Approach

Algorithm 2.4. Sketch-driven layer assignment

Input: an acyclic graph (V, E) with a sketch S
 $L \leftarrow$ empty list of spans // *The spans are always sorted by their positions.*
for each node $v \in V$ **do**
 $(l_v, r_v) \leftarrow$ min./max. horizontal position occupied by v in S
 $(V_s, l_s, r_s) \leftarrow (\emptyset, \perp, \perp)$
 for each span $(V_t, l_t, r_t) \in L$ **while** $V_s \neq \perp$ **do**
 if $r_v \leq l_t$ **then** // *The current span is further right, so break iteration.*
 if $V_s = \emptyset$ **then**
 └ Insert a new span $(\{v\}, l_v, r_v)$ before (V_t, l_t, r_t) in L
 $V_s \leftarrow \perp$
 else if $l_v < r_t$ **then** // *The node overlaps with the current span.*
 if $V_s = \emptyset$ **then** // *This is the first overlapping span.*
 $(V_s, l_s, r_s) \leftarrow (V_t \cup \{v\}, \min\{l_t, l_v\}, \max\{r_t, r_v\})$
 Replace (V_t, l_t, r_t) by (V_s, l_s, r_s) in L
 else // *Merge the previously found span with the current one.*
 $(V_s, l_s, r_s) \leftarrow (V_t \cup V_s, l_s, \max\{r_t, r_s\})$
 Replace (V_t, l_t, r_t) and its preceding span by (V_s, l_s, r_s)
 if $V_s = \emptyset$ **then** // *All existing spans are further left, so create a new one.*
 └ Append a new span $(\{v\}, l_v, r_v)$ to L
// *Create layers from the spans derived from the sketch.*
for each span $(V_t, l_t, r_t) \in L$ **with index** i **do**
 └ Assign all nodes V_t to layer V_i
// *Correct the layering so all edges point from left to right.*
Find a topological sorting v_1, \dots, v_n of the nodes V
for $i = 1 \dots n$ **do**
 $V_j \leftarrow$ layer to which v_i is assigned
 for each outgoing edge $(v_i, w) \in E$ **do**
 $V_k \leftarrow$ layer to which w is assigned
 if $k \leq j$ **then** // *Violation detected – move target node to the next layer.*
 └ Reassign w to layer V_{j+1}
Eliminate layers that are left empty after reassignment

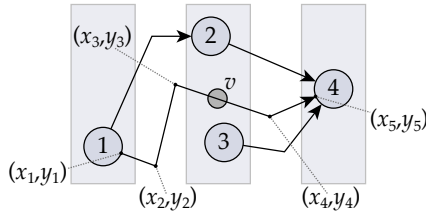


Figure 2.34. Computation of a virtual position for the dummy node v of the long edge $(1, 4)$: given the sketch $(x_1, y_1), \dots, (x_5, y_5)$, the average layer position of v (determined by nodes 2 and 3) is between x_3 and x_4 , hence $\text{pos}(v)$ is scaled accordingly between y_3 and y_4 (see Algorithm 2.5). This results in the node order $2, v, 3$ for the second layer.

each node must be compared with all previously processed nodes. This leads to a running time of $\mathcal{O}(|V|^2)$ for deriving layers, and a total running time $\mathcal{O}(|V|^2 + |E|)$ for the algorithm.

Crossing minimization. Sketch-driven crossing minimization aims at reproducing the node order induced by a sketch in each layer. If the layering contains no long edges, this can be done by sorting the nodes of each layer by their vertical positions. These positions can be derived in different ways, as explained above for sketch-driven cycle elimination. However, the process is less obvious when the layering contains dummy nodes created to split long edges. For such nodes we have to derive a virtual position from the sequence of points $(x_1, y_1), \dots, (x_m, y_m)$ that represent the sketch of the corresponding edge. Algorithm 2.5 does this by finding a point index i such that $x_i < \bar{x} < x_{i+1}$, where \bar{x} is the average horizontal position of the regular nodes in that layer. The virtual position assigned to the dummy node is scaled between y_i and y_{i+1} . An example is shown in Figure 2.34.

2.4.2 Evaluation

Since the interactive layout approach presented here requires positioning information for all elements of the graph, it cannot be applied when elements are added without initial positions. However, it is simple to implement and

2. The Layer-Based Approach

Algorithm 2.5. Sketch-driven node ordering

Input: a graph (V, E) with proper layering L_1, \dots, L_k and a sketch S

for $i = 1 \dots k$ **do**

- $\bar{x} \leftarrow$ average horizontal position of the nodes L_i in S
- for each** $v \in L_i$ **do**
 - if** v is a long edge dummy **then**
 - $e \leftarrow$ the original edge in E for which v was created
 - $(x_1, y_1), \dots, (x_m, y_m) \leftarrow$ sketch of e in S ($m \geq 2$)
 - if** $x_1 \geq \bar{x}$ **then**
 - $\lfloor \text{pos}(v) \leftarrow y_1$ // Take the source point for sorting.
 - else if** $x_m \leq \bar{x}$ **then**
 - $\lfloor \text{pos}(v) \leftarrow y_m$ // Take the target point for sorting.
 - else**
 - $i \leftarrow 1$ // Find point i such that $x_i < \bar{x} < x_{i+1}$.
 - while** $i \leq m - 2$ and $x_{i+1} < \bar{x}$ **do**
 - $\lfloor i \leftarrow i + 1$
 - $\lfloor \text{pos}(v) \leftarrow y_i + \frac{\bar{x} - x_i}{x_{i+1} - x_i} (y_{i+1} - y_i)$
 - else**
 - $\lfloor \text{pos}(v) \leftarrow$ vertical position of v in S

\lfloor Sort the nodes L_i by their $\text{pos}(v)$ values

very fast, hence it provides a useful alternative to more complex solutions based on constraints [NW02, DMW09].

A second use case of sketch-driven layout, besides the layout of dynamic graphs, is to consider user hints in the generated layouts. Do Nascimento and Eades proposed an extension of the layer-based method that considers user hints by focusing subgraphs that need improvement and by adding layout constraints [dNE02]. Here, in contrast, we give the user the option of manually modifying the layout and use the modified layout as a sketch for the next layout computation. Again, this approach is less flexible, but much easier to realize compared to previous solutions. As a further advantage, all

three sketch-driven phases can be employed independently of each other, allowing to limit the user's control on the generated layout. For instance, one could use sketch-driven crossing minimization together with standard algorithms for the first two phases, allowing to reorder nodes vertically, but not horizontally.

The proposed algorithms for sketch-driven layout have been evaluated towards two criteria: stability and responsiveness. An algorithm is regarded as stable if it does not modify the layout when applied to its own output, i. e. it always reaches a fixed point after one application. This property is important to verify the correctness of the algorithms, because it means that the layout detected from the sketch is consistent with the generated layout. The second property, responsiveness, refers to the consistency of the generated layout with the user's expectation when the graph or its layout is modified. This is important to ensure readability in a dynamic graph scenario and to enable effective layout generation based on user hints.

Stability. 50 random graphs with between 4 and 46 nodes and between 3 and 157 edges have been created for evaluating stability of the sketch-driven algorithms. The evaluation involved the following process for each graph:

1. Execute the standard layer-based algorithm for static layout.
2. Execute the layer-based algorithm with sketch-driven cycle elimination, layer assignment and crossing minimization.

In all 50 cases step 2 did not modify the layout that was generated in step 1. These results suggest optimal stability of the proposed algorithms.

Responsiveness. Since the second evaluated property refers to the interaction of users with the sketch-driven layout algorithms, the evaluation was based on an experiment with human participants. Each participant had to fulfill 18 tasks using a graph editor (KEG, see Section 4.5.1) on three prepared graphs. The graphs and corresponding tasks are shown in Appendix A. For each task the participants

1. manually modified the layout with drag-and-drop operations,

2. The Layer-Based Approach

2. pressed a button to invoke sketch-driven automatic layout,
3. verified the resulting layout according to the given task, and
4. repeated these steps if the requirements given in the task were not met.

The sketch-driven algorithms were not explained to participants beforehand because the experiment aimed at their intuition.

Eight students and members of the research group participated in the experiment. On average they performed 1.96 drag-and-drop operations and 1.28 layout invocations per task. For 91.9% of the layout invocations the participants were satisfied with the computed sketch-driven layouts. This high portion indicates that the overall responsiveness of the proposed sketch-driven algorithms is good. However, two weaknesses were observed, which provoked most of the remaining 8.1% of the invocations where users reported that they expected different results. The first weakness is the missing responsiveness to vertical node positioning, which is due to the absence of a sketch-driven node placement algorithm. Such an algorithm could be developed in order to complement the approach. The second weakness is that for long edges the order of dummy nodes is recomputed in each layer independently of the other layers. On the one hand this gives high flexibility for routing long edges, but on the other hand it leads to confusing layouts when multiple long edges cross each other more than once. Some users found it tedious to change the routing of long edges in each single layer and would have preferred to move the whole long edge at once. Such a functionality would have to be implemented as an alternative method for sketch-driven crossing minimization, where the order of dummy nodes is determined.

Sketches from layout algorithms. The sketch-driven approach is not restricted to user interaction, but can also be applied to drawings generated by other layout algorithms. In this way the coarse topology of the original drawing is preserved, but organized into layers, potentially retaining some of its characteristics. This approach can serve as a useful alternative to the normal node ordering methods. For instance, planarization-based algorithms usually produce fewer edge crossings than layer-based algorithms.

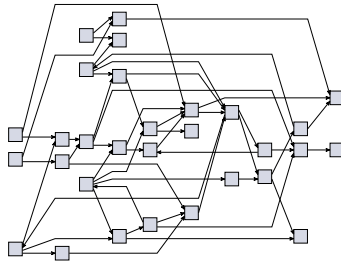
2.4. Interactive Layout

By using a such a drawing as a sketch, the resulting layer assignment and node ordering is likely to yield fewer crossings, too. I evaluated this assumption with a set of 50 randomly generated graphs. The standard layer-based algorithm resulted in 73.6 crossings on average, while the sketch-driven algorithm applied to a topology-shape-metrics layout gave 50.2 crossings on average, which is a reduction by 32%. However, the price for this improvement is high: the layer-based algorithm arranged 93% of the edges from left to right, while the sketch-driven algorithm achieved this only for 60%. An example is shown in Figure 2.35.

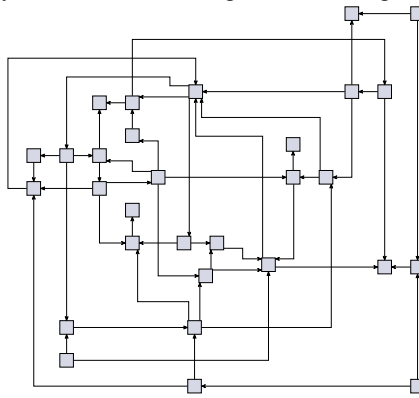
Another effect is achieved if the sketch-driven approach is combined with force-based drawings. When applied to the same set of random graphs as above, the standard deviation of edge lengths was 84.0 with the sketch-driven algorithm applied to drawings of the method of Fruchterman and Reingold [FR91], which emphasizes uniform edge lengths. This is a reduction by 57% compared to the value 194.5 obtained with the standard layer-based algorithm. Interestingly, the average number of crossings was nearly equal for these two algorithms. The orientation of edges, however, is totally random for the force-based sketches: only about 50% of the edges point from left to right.

In summary, the approach of using existing layout algorithms to produce sketches and to transform these into layer-based drawings can be used to obtain layouts where certain characteristics shall be emphasized. If the employed layout algorithms are designed for undirected graphs, the sketch-driven algorithm yields low consistency of edge directions. In return, other aesthetic criteria such as the number of edge crossings or the uniformity of edge lengths can be promoted.

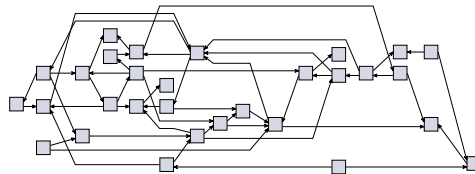
2. The Layer-Based Approach



(a) Layer-based (34 crossings, 46 left-to-right edges)



(b) Planarization-based (10 crossings)



(c) Sketch-driven (17 crossings, 29 left-to-right edges)

Figure 2.35. The drawing (c) was created by applying the sketch-driven algorithm to drawing (b), which was made with the *Planarization* algorithm of the OGDF library. This combination results in only half of the crossings of the standard layer-based drawing (a), but also fewer edges pointing from left to right (the graph has 50 edges in total).

Meta Layout

There are many different approaches for drawing graphs, and all have their strengths and weaknesses. Therefore successful graph drawing libraries include multiple algorithms, and usually they offer a bulk of configuration options to allow users to tailor the generated layouts to their needs. However, the proper choice of layout algorithms and parameter settings often require detailed knowledge of the background of these algorithms. Obtaining such knowledge or simply testing all available configuration options is not feasible for users who require quick results. Consequently, methods for automatic configuration are sought after.

Abstract layout denotes the annotation of graphs or graph elements with directives for layout algorithm selection and configuration. *Concrete layout* is a synonym for the *drawing* of a graph and is represented by the annotation of graph elements with specific values for their position and size. When a layout algorithm is executed on a graph, it transforms the associated abstract layout into a concrete layout. By *meta layout* we denote a process of generating abstract layout.

In this chapter I contribute methods for the successful management of large collections of layout algorithms. The foundation for these methods is laid by a meta model, introduced in Section 3.1, that allows the description of graph structures as well as their annotation with abstract layouts and concrete layouts. The meta model is used in a scheme for the integration of multiple layout algorithms and layout configuration methods. The implementation and evaluation of these general concepts are described in Part II of this thesis. With this premise we will discuss an evolutionary algorithm for optimal layout configuration in Section 3.2 [SDvH14a, SDvH14b].

3. Meta Layout

3.1 A Generic Layout Interface

The main component of a graph layout interface is a data structure for transferring graphs to the layout algorithms and receiving their results. In the context of model-driven engineering, it is advantageous to express the involved data with the same means as the models that are to be visualized, i. e. to have a meta model for the layout interface. This allows to employ MDE techniques such as model transformation and validation in the layout process, facilitating its integration into modeling applications (see Chapter 4).

The usual approach employed in graph layout software [JM04] is to offer a data structure that is directly implemented in the respective programming language, e. g. in Java for yFiles [WEK04], in C for Graphviz [EGK⁺04b], or in C++ for OGDF [CGJ⁺13]. Some tools also offer textual formats such as GraphML [BEP13] for the graph transfer. Maier and Minas presented a *graph layout meta model* (GLMM) that allows to create the graph structure by transforming it from the abstract syntax or the concrete syntax representation [MM10b]. This is a good step towards applying MDE concepts to graph layout. However, their meta model does not include any representation of the abstract layout.

Bertolazzi et al. proposed to configure the layout automatically by evaluating the available layout algorithms w.r.t. aesthetic criteria, which have to be prioritized by the user [BDL95]. Niggemann and Stein proposed to choose layout algorithms depending on the structural properties of a subgraph (e. g. the number of nodes and edges) and used a learning process to develop the mapping of structural properties to the most suitable algorithms [NS00]. A similar approach is applied by Archambault et al., who select specific layout algorithms when the corresponding structural patterns are found in a subgraph, e. g. a tree layout algorithm for trees [AMA07]. Maier and Minas proposed *layout patterns* for the integration of multiple layout algorithms in graphical editors [MM07, MM10a]. All these methods are restricted to the selection and combination of layout algorithms, and do not solve the more general problem of producing abstract layouts that consider all parameters supported by the available layout algorithms.

3.1. A Generic Layout Interface

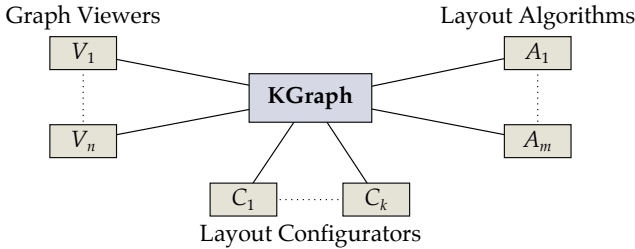


Figure 3.1. The KGraph meta model is the main interface for combining multiple graph viewers, layout configurators, and layout algorithms. Viewers provide the graph structure, configurators provide an abstract layout, and layout algorithms provide a concrete layout.

Here I propose a metamodel called *KGraph* that combines the representation of graph structures, concrete layout, and abstract layout. Its role is shown in Figure 3.1: the graph structure is derived from graph viewer components, which provide the abstract syntax and concrete syntax of models, layout configurators supplement the graph with abstract layout data, and layout algorithms generate concrete layout data. Multiple viewers, configurators, and algorithms operate on the same graph layout meta model, hence the meta model serves as the main interface through which these components can be combined. The most important advantage of this approach is that it allows to completely specify both the input and the output of any layout algorithm, hence layout algorithms can be regarded as functions that modify GLMM instances, and their behavior can be controlled by adapting the parameters embedded in these instances.

Furthermore, I propose a general concept for the integration of multiple layout configuration methods, including the automatic selection of layout algorithms as well as practical interfaces that allow users and tool developers to adapt the configuration to their needs. The details of the proposed graph layout meta model and layout configuration approach are discussed in the remainder of this section.

3. Meta Layout

3.1.1 The KGraph Meta Model

KGraph is an extensible graph layout meta model (GLMM) [SSvH12a, SSvH12b]. Figure 3.2(a) shows a class diagram of KGraph created in the *Ecore* notation of the Eclipse Modeling Framework (EMF). The core classes in KGraph are KNode, KEdge, and KPort, representing nodes, edges, and ports, respectively. The associations between these classes are analogous to the graph notation defined in Section 1.3. Ports have an explicit representation because their presence has important implications on the layout (see Section 2.2). The same applies to labels (KLabel class), of which arbitrarily many can be attached to the three core graph element types.

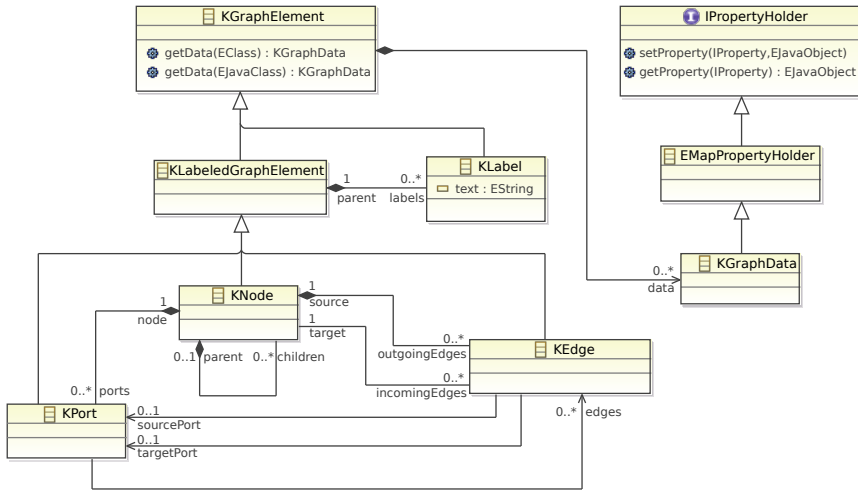
The meta model supports compound graphs, in which each node may contain a nested subgraph. This is expressed by the children and parent references of the class KNode. The graph itself is also represented by an instance of KNode.

Concrete layout data. All graph elements are extensible via the data reference, which points to instances of KGraphData. Such instances are identified by their concrete subclass; the most important subclasses are KShapeLayout and KEdgeLayout, defined in a supplementary meta model named *KLayout-Data* (see Figure 3.2(b)). These two classes hold the concrete and abstract layout data. By convention, an instance of KEdgeLayout is attached to each KEdge instance, and an instance of KShapeLayout is attached to each KNode, KPort, and KLabel instance. A shape layout consists of horizontal and vertical coordinates for the position (xpos and ypos) as well as width and height values for the size of the respective element. An edge layout contains a series of points that form the line segments for drawing the edge: sourcePoint and targetPoint are the points where the edge touches the source and target nodes or ports, respectively, and bendPoints is the list of points where two consecutive line segments touch each other.

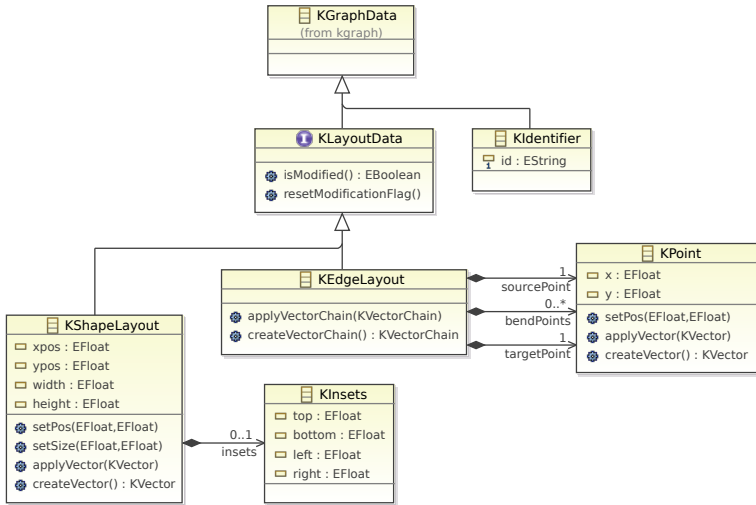
The KIdentifier graph data class is used to provide a unique text for the identification of graph elements, e. g. for cross-referencing elements in textual formats of the KGraph meta model (see Section 4.5.2).

The children of a compound node *C* are drawn inside the boundary of *C*. If *C* has a reserved area where children must not be positioned, KInsets

3.1. A Generic Layout Interface



(a) KGraph



(b) KLayoutData

Figure 3.2. Ecore class diagrams of the KGraph and KLayoutData meta models.

3. Meta Layout

can be used to specify the size of that area. For instance, the compound node C in Figure 3.3(a) has an inset of width i_l on the left side, i_r on the right side, i_t on the top side, and i_b on the bottom side. The area in which children are positioned, called the *child area* of C , is represented by a dashed rectangle.

The coordinates of each graph element in the concrete layout could be stored as absolute values, i. e. all elements would have the origin of the graph drawing as reference point. However, this very simple approach implies that whenever the position of an element x is modified, all elements that are directly or indirectly contained by x must also be updated in order to preserve their position relative to x . For instance, moving a node implies moving all its labels and ports by the same amount. Therefore I propose a relative positioning scheme that is used as convention for all coordinates of the `KLayoutData` meta model. This scheme includes the following rules (see Figure 3.3(a)).

- ▷ The positions of nodes, ports, and labels identify their upper left corner.
- ▷ The top-level `KNode` instance representing the whole graph marks the origin position $(0, 0)$.
- ▷ The position of a node v is relative to its parent $v_p(v)$ plus the parent's left and top insets.
- ▷ The position of a port p is relative to the containing node v for which $p \in P(v)$.
- ▷ The position of a node or port label is relative to the respective node or port.

In compound graphs an edge may connect two nodes that are contained by different parent nodes, therefore it is not clear which element to use as reference point. The convention chosen for the `KGraph` / `KLayoutData` meta model is to use the parent of the source node as reference to which all points of an edge (source point, bend points, and target point) are relative to. For flat graphs this convention means that edges have the same reference point as nodes. For compound graphs the reference points can be different, as

3.1. A Generic Layout Interface

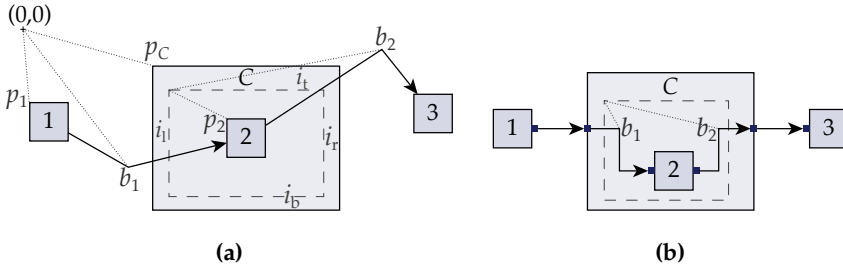


Figure 3.3. Different reference points of graph elements, shown by dotted lines: (a) Nodes 1 and C have positions p_1 and p_C relative to the origin $(0,0)$, while the position p_2 of node 2 is relative to $p_C + (i_L, i_T)$; the bend point b_1 has the same reference point as the source node 1, and b_2 has the reference point of the source node 2. (b) Both bend points b_1 and b_2 are relative to the compound node C plus its insets due to the exception applied to the edge $(C, 2)$. The dashed rectangles represent the child areas of compound nodes, surrounded by their insets.

seen in Figure 3.3(a). The labels of an edge always have the same reference point as the edge itself.

An exception is made for edges (v, w) for which w is a direct or indirect child of v . In this case the reference point is v plus the left and top insets of v (see the edge $(C, 2)$ in Figure 3.3(b)). The reason for this exception is that it makes edges more consistent that are fully contained in one compound node, simplifying layout computations for port-based graphs where such connections between compound nodes and their children occur frequently.

Abstract layout data. All subclasses of `KGraphData` inherit from the `IPropertyHolder` interface, which allows to modify *properties* of graph data. A property assignment is a key-value pair with a specific type. Although properties can be used to attach arbitrary supplementary data, their primary purpose is to specify the abstract layout. Each property assignment to a `KShapeLayout` or `KEdgeLayout` instance is used to control a parameter of a layout algorithm. Such a parameter can be specific for a given implementation of a layout algorithm, or it can be a shared parameter that is understood by multiple algorithms, e. g. the spacing between nodes, the main direction of edges, or the edge routing style.

3. Meta Layout

3.1.2 Layout Configuration

The basic building block for our layout configuration concept is a *layout option*, representing a single configuration parameter that is understood by one or more layout algorithms. Let Ω be the set of available layout options. Each $o \in \Omega$ is assigned a *type*, that is a set $T(o)$ of admissible values.

Definition 3.1 (Layout option mapping). A *layout option mapping* is a function $\alpha : \Omega \rightarrow \mathcal{T} \cup \{\perp\}$, where $\mathcal{T} = \bigcup_{o \in \Omega} T(o)$ is the union of all types. Such a mapping is called *type-consistent* if for all options $o \in \Omega : \alpha(o) \in T(o)$ or $\alpha(o)$ is undefined, denoted as $\alpha(o) = \perp$.

Definition 3.2 (Abstract layout). Let G be a graph and $\mathcal{E}_G = V \cup E \cup P \cup L$ be the set of its elements, consisting of the nodes V , the edges E , the ports P , and the labels L . An *abstract layout* of G is a mapping $\lambda : \mathcal{E}_G \rightarrow [\Omega \rightarrow \mathcal{T} \cup \{\perp\}]$, where $[\Omega \rightarrow \mathcal{T} \cup \{\perp\}]$ denotes the set of type-consistent layout option mappings.

Given a pair (G, λ) , a layout algorithm produces a concrete layout for G by interpreting the layout option mappings stored in λ for each graph element. The choice of layout algorithm is also encoded as a layout option o_A , hence it can be processed in the same way as other options.

Each layout option is targeted to a specific kind of graph elements. Therefore the layout option mapping $\lambda(e)$ for an element $e \in \mathcal{E}_G$ is usually undefined for options o that are not targeted to e : $\lambda(e)(o) = \perp$. For instance, the value of an option that determines the minimal width of nodes is ignored by layout algorithms when assigned to anything that is not a node, hence its value can be undefined in the mappings of all graph elements $\mathcal{E}_G \setminus V$.

Definition 3.3 (Combined abstract layout). Multiple configurations can be combined with the operator \circ : given two abstract layouts λ_1 and λ_2 , $\lambda_1 \circ \lambda_2 = \lambda'$ is a combined abstract layout such that for all graph elements $e \in \mathcal{E}_G$ and options $o \in \Omega$ the result is $\lambda'(e)(o) = \lambda_1(e)(o)$ if $\lambda_2(e)(o) = \perp$ and $\lambda'(e)(o) = \lambda_2(e)(o)$ otherwise.

The combination operator gives priority to the second configuration, hence with a sequence $\lambda_1, \dots, \lambda_k$ the combination $\lambda_1 \circ \dots \circ \lambda_k$ yields an

3.1. A Generic Layout Interface

abstract layout in which all configurations in the sequence are considered, but those with higher index are prioritized in case of conflict.

Definition 3.4 (Layout configurator). A *layout configurator* is an algorithm that computes abstract layouts (a *meta layout algorithm*).

Configurators can be integrated in the layout process with the following scheme.

1. Execute the configurators C_1, \dots, C_k for the graph G in order of increasing priority, yielding abstract layouts $\lambda_1, \dots, \lambda_k$.
2. Determine the combined abstract layout $\lambda_C = \lambda_1 \circ \dots \circ \lambda_k$.
3. Determine the layout algorithm $A = \lambda_C(G)(o_A)$.
4. Execute the layout algorithm A on G with the configuration λ_C .

According to the KGraph data format, the combined abstract layout λ_C is attached to the graph G using property assignments, as described in Section 3.1.1. Given a graph element $e \in \mathcal{E}_G$, an option $o \in \Omega$, and the assigned value $x = \lambda_C(e)(o)$, the key-value pair (o, x) is assigned to the layout data of e .

Compound graphs. Several authors have proposed to decompose graphs and to apply different layout algorithms to their components [HH91, KLSW94, NS00, AMA07]. Each component is wrapped into a new compound node v_C ; the general procedure is to first apply a layout algorithm to the content of v_C , determine its required size, and then apply a layout algorithm to the container of v_C , treating v_C as a regular node with fixed size. For many applications the graph structure is inherently hierarchical, so it is not required to compute an artificial decomposition. Among many others, this applies to statecharts and actor models.

As mentioned in Section 3.1.1, the KGraph format directly supports compound graphs. The layout process can be extended to handle such compound graphs with a recursive scheme as shown in Algorithm 3.1. However, this divide-and-conquer approach fails when the graph contains cross-hierarchy edges, i. e. edges that connect nodes with different parents

3. Meta Layout

Algorithm 3.1. Layout of compound graphs

Input: a graph G and layout configurators C_1, \dots, C_k
 $\lambda_C \leftarrow \lambda_\perp$ // λ_\perp is the abstract layout that maps all options to \perp .
for $i = 1 \dots k$ **do**
 // The algorithm C_i computes an abstract layout that is combined with λ_C .
 $\lambda_C \leftarrow \lambda_C \circ C_i(G)$
RecursiveLayout(G, λ_C)
// Two special layout options are used here: o_A selects a layout algorithm, and
// o_H activates processing of all hierarchy levels of the contained subgraph at once.
procedure RecursiveLayout(v, λ)
 $A \leftarrow \lambda(v)(o_A)$
 if $\neg(\lambda(v)(o_H) = \text{true})$ and A supports o_H **then**
 for each $w \in V_c(v)$ **do** // $V_c(v)$ is the set of children of v .
 if $V_c(w) \neq \emptyset$ **then**
 RecursiveLayout(w, λ)
 // The algorithm A sets the concrete layout of the content of v .
 $A(V_c(v), \lambda)$

in the compound hierarchy. Such edges must be ignored by the layout algorithms because the corresponding nodes are positioned at different stages of the RecursiveLayout procedure. Some algorithms are capable of processing a whole compound graph at once including cross-hierarchy edges [For02, DGC⁺05]. In Algorithm 3.1, this behavior is controlled with the layout option o_H : if it is set to true and the active layout algorithm supports that option, the processing of deeper hierarchy levels contained in compound nodes is delegated to the layout algorithm, otherwise the recursive scheme is applied.

Standard configurators. The integration of multiple layout configurators in the layout process allows to consider many different requirements and to keep a modular separation of their implementations. Priorities are assigned to configurators according to their generality: more general configurators

3.1. A Generic Layout Interface

get lower priorities, while those that are more specific to certain applications get higher priorities. The following list summarizes a selection of configurators that are very useful for the practical integration of automatic layout in modeling applications [SSM⁺13]. More details on the realization of these configurators and their user interface are given in Chapter 4.

Defaults. In many cases it makes sense to assign default values to layout options such that for a large number of graphs good layouts are produced. For each layout algorithm these default values can be overwritten by algorithm-specific values. Of course the configurator for default values has the lowest priority.

Tool-specific settings. Tool developers need to customize the layout configuration to meet the requirements of the application. For this purpose, layout option mappings are associated with specific model classes, which can either represent components of the concrete syntax or components of the abstract syntax. The option mappings are applied to all graph elements that correspond to instances of the associated classes. These associations can be specified in an abstract format such as XML. In some cases, however, such a static association is not sufficient, since the actual value of some option may depend on certain properties of the class instance. Such a dynamic configuration can be realized by associating a *semantic* configurator to the model class, that is a specialized component into which the required property checks can be coded.

User preferences. Users may want to modify some parameters globally for their own environment. This can be done by offering a user interface in which the same kind of associations between layout option mappings and model classes as already used for tool-specific settings can be made. The associations entered through this interface are kept in the local preference storage.

Diagram-bound settings. In order to allow a customized configuration for each diagram, layout option mappings must be linked directly with diagram elements. This can be done with annotations of either the concrete syntax model or the abstract syntax model. In the former case the options are applied only to the respective diagram, while in the

3. Meta Layout

latter case they are applied to all diagrams derived from the model. This distinction only makes sense if the model and view are separated, like in most UML editors. The diagram-bound configuration can be displayed and modified in the user interface in form of a table that contains the layout option mapping for the currently selected diagram element.

View management. The handling of graphical views can be improved with the concept of *view management* [FvH10b], where a simple interface for the combination of triggers and effects of the modeling environment is provided. In this context it is often necessary to perform automatic layout as an effect on a graphical view, and to apply different configurations to the layout depending on the state of the overall system. This is done with an interface for setting layout option mappings in a single layout execution; these mappings are held in a hash map, which is discarded after the layout is applied.

3.1.3 Meta Data of Layout Algorithms

The generic handling of layout algorithms requires meta data of these algorithms and their supported parameters, allowing automatic processing in user interfaces and meta layout algorithms. In this section I present a meta data format that supports the standard configurators mentioned in Section 3.1.2 as well as optimizing methods as discussed in Section 3.2. The basic element of this format is a layout option, which is assigned the following attributes.

Layout Option

- ▷ identifier
A unique string for referencing the option.
- ▷ name
The name to be shown in user interfaces.
- ▷ description
A concise explanation of what the option does.

3.1. A Generic Layout Interface

- ▷ type
The data type, that is one of boolean, string, integer, floating point, enumeration, enumeration set, or object. For options with enumeration type, one value from a specific set of possible values can be assigned, while for the enumeration set type an arbitrary subset of such a given set is chosen. The object type is used for extension to types that are not considered here, e. g. for assigning class instances.
- ▷ class
The types enumeration, enumeration set, and object require more specific information from which the set of assignable values is derived. For enumeration typed options, for instance, the name of a specific enumeration must be given in this attribute. For other types it can be left empty.
- ▷ applies to
Which kinds of graph elements the option can be applied to. This is specified as a subset of parents, nodes, edges, ports, and labels. The keyword parents relates to compound nodes, i. e. instances of KNode for which the children reference is not empty, including the graph itself.
- ▷ default
The default value of the layout option, which is applied when no other value is determined.
- ▷ lower bound
The lower bound on possible values for integer and floating point types.
- ▷ upper bound
The upper bound on possible values for integer and floating point types.
- ▷ variance
A measure that determines how much values of integer and floating point typed options may be modified in an automatic process. If variance is zero, the option must not be considered in optimizing meta layout approaches (see Section 3.2).
- ▷ dependencies
A list of references to other layout options with associated expected

3. Meta Layout

values. The option should be made visible in the user interface only if all its dependencies are met, i. e. the referenced options are assigned the given expected values.

Some layout options only have an influence on the behavior of a layout algorithm if certain other options are set to specific values. For instance, a parameter of a submodule has no effect unless the respective submodule is activated. The dependencies list can be used to model this behavior by specifying the dependencies between layout options. The graph formed by all layout options and their dependencies must be acyclic.

We classify layout algorithms into *layout types* according to their basic graph drawing approach. This classification is very useful for meta layout because algorithms of the same type often produce layouts with similar properties, while different layout types generally lead to different layouts. The standard layout types considered here are “layered” for the layer-based approach as discussed in Chapter 2, “force” for force-directed and energy-based methods, “orthogonal” for planarization-based methods that produce orthogonal or quasi-orthogonal layouts, “planar” for other methods that build on planarity, “circular” for the circular layout approach, and “tree” for methods that work on spanning trees. Algorithms that do not fit into this classification are gathered under the default type “other.”

The meta data of layout algorithms include the following attributes.

Layout Algorithm

- ▷ identifier
A unique string for referencing the algorithm.
- ▷ name
The name to be shown in user interfaces.
- ▷ description
A description of the functioning and the behavior of the algorithm.
- ▷ class
Name of a component (usually a class) in which the algorithm is implemented. In order to execute the algorithm, an instance of the referenced component is created and its layout method is invoked.

3.1. A Generic Layout Interface

- ▷ category
Categories serve to group multiple algorithms into bundles so they are recognized as being part of the same library in the UI.
- ▷ layout type
Reference to the layout type the algorithm belongs to.
- ▷ known options
A list of references to layout options that are supported by the layout algorithm.
- ▷ supported features
A list of graph features that are supported by the algorithm. Typical examples for features that are not supported by all layout algorithms are self-loops, multi-edges, edge labels, ports, or nested subgraphs. Specifying which kinds of features can be processed by an algorithm may help optimizing meta layout methods to decide which algorithm is most suitable for a given graph based on the specific features found in that graph.
- ▷ supported diagrams
A list of references to diagram types for which the algorithm is known to be suitable (see below). Each referenced type can be assigned a priority in order to express the quality of layouts created for diagrams of that type.

Some layout algorithms are specifically targeted to certain types of diagrams by considering domain-specific layout requirements that cannot be applied to general graphs. For instance, the algorithm of Gutwenger et al. [GJK⁺03] targets UML class diagrams, while the algorithm of Klauske et al. [KSSvH12] targets data flow diagrams. Such a specialization can be considered by defining a diagram type D , referencing it with a certain priority in the meta data of the layout algorithm, and associating D with the instances of the respective diagram viewer. Whenever a layout is requested for that viewer, the layout algorithm with the highest priority for the type D is chosen and executed. This procedure can be integrated in the standard configurators introduced in Section 3.1.2.

3. Meta Layout

3.2 Optimal Layout Configuration

In this section we discuss the optimization problem of computing an abstract layout that yields a high-quality concrete layout for a given graph [SDvH14a]. As usual, the quality of concrete layouts is judged according to aesthetic criteria [BRSG07]. The search space for this problem is huge: each layout option $o \in \Omega$ has $|T(o)|$ possible values (see Section 3.1.2), hence the total number of layout option mappings for one element of a graph is $\prod_{o \in \Omega} |T(o)|$. Obviously the brute-force approach of testing all possible abstract layouts can only be applied to a small subset of the available options Ω , and only if all options o in that subset have small type sets $T(o)$.

Here we consider a heuristic based on an evolutionary algorithm. Several authors have proposed evolutionary algorithms where the individuals are represented by lists of coordinates for the positions of the nodes of a graph [BB01, BBS96, EM01, GMEJ90, RSOR98, Tet98, Vra09]. Other works have focused on integrating meta heuristics in existing layout methods. De Mendonça Neto and Eades proposed a system for automatic learning of parameters of a simulated annealing algorithm [dMNE93]. Utech et al. introduced a genetic representation that combines the layer assignment and node ordering steps of the layer-based drawing approach with an evolutionary algorithm [UBSE98]. Such a combination of multiple NP-hard steps is also applied by Neta et al. for the topology-shape-metrics approach [NAGa⁺12]. They use an evolutionary algorithm to find planar embeddings (*topology* step) for which the other steps (*shape* and *metrics*) are able to create good layouts. All these approaches, however, are fundamentally different from the one presented here, since they target the problem of creating concrete layouts instead of abstract layouts.

As this is—to my knowledge—the first generic approach to applying meta heuristics to the layout configuration problem, we will begin with a discussion of a genetic representation of abstract layouts that is independent of the applied heuristic. This representation and the according fitness function can serve as a basis for many different meta layout heuristics. A further aspect considered here is how to involve the user in the search process. An inspiring idea was communicated by Biedl et al. [BMRW98]: by displaying multiple layouts of the same graph, the user may select those

3.2. Optimal Layout Configuration

that best match her or his expectations. We build on that idea and adapt parameters of the fitness function according to the user’s selection. The evolutionary algorithm and the fitness function adaptation are evaluated in a user study.

Although the abstract layouts resulting from the evolutionary meta layout method are generated using a given graph, they can as well be applied to other graphs. Therefore this meta layout method can be understood as *training*, generating templates to be reused for a class of diagrams.

Parts of the results presented here are based on the research thesis of Duderstadt [Dud11].

3.2.1 Genotypes and Phenotypes

The *genotype* of an individual is its genetic code, while the *phenotype* is the total of its observable characteristics. In biology a phenotype is formed from its genotype by growing in a suitable environment. We propose to use abstract layouts (configurations) as genotypes, and concrete layouts (drawings) as phenotypes. The “environment” for this kind of phenotypes is a graph. We generate the concrete layout $L(\lambda)$ that belongs to a given abstract layout λ by applying all parameters encoded in λ to the chosen layout algorithm A , which is also encoded in λ , and executing A on the graph given by the environment. This encoding of parameters and algorithm selection is done with a set of *genes*, which together form a *genome*. A gene consists of a *gene type* with an assigned value. The gene types are derived from the meta data described in Section 3.1.3. A gene type has an identifier, a data type, optional lower and upper bounds, and an optional parameter controlling the standard deviation of Gaussian distributions. Here we consider only the data types integer, floating point, Boolean, and enumeration; options with other data types mentioned in Section 3.1.3 are omitted.

Each genome contains a gene g_T for selecting the layout type as introduced in Section 3.1.3, a gene g_A for selecting the layout algorithm, and a gene g_o for each layout option $o \in \Omega$. It is possible to use only a subset of these genes, as long as all genomes contain the same subset. Such a restriction can serve to focus on the selected layout options in the optimization process, while other options are kept constant. For the sake of simplicity,

3. Meta Layout

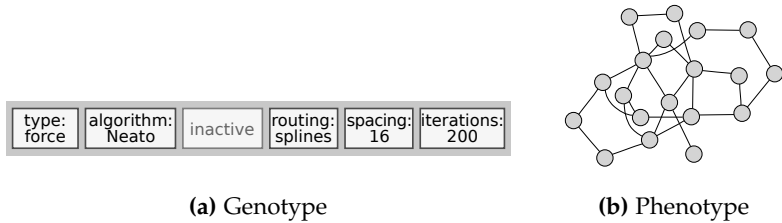


Figure 3.4. (a) A genome with six genes. The layout type gene is set to force-based algorithms, the layout algorithm gene is set to a specific algorithm named “Neato”, and three parameters of that algorithm are set with the remaining genes. One gene is inactive because the corresponding layout option is not supported by Neato. (b) A phenotype of the genome, represented by a layout generated by Neato for an arbitrary graph.

we consider only layout options that are applied to the whole graph, i. e. options for which the applies to attribute in the meta data is set to *parents*. Other kinds of options are associated with specific graph elements such as nodes or edges, hence they could only be included with full detail in the genetic representation if the representation was linked to a given graph. The representation discussed here, in contrast, is independent of any particular graph and can be reused for multiple graphs.

Some genes of a genome are dependent of each other. The gene g_A , for instance, is constrained to be set to a layout algorithm that belongs to the layout type selected in g_T . Furthermore, the layout algorithm A selected in g_A may not support all layout options in Ω , therefore the options that are not supported by A are marked as *inactive*. A genome with six genes and a possible phenotype are shown in Figure 3.4.

Inactive genes of a genome X do not contribute to the characteristics of the phenotype of X , i. e. of its drawing, hence two genomes that differ only in their inactive genes may produce the same drawing. On the other hand, some layout algorithms are randomized and produce different drawings when executed twice with the same configuration, even when applied to the same graph. However, we assume that drawings that result from the same configuration tend to be similar with respect to our fitness function, hence this ambiguity is probably not noticeable in practice.

3.2. Optimal Layout Configuration

Fitness function. Our genotypes have a completely different representation compared to previous evolutionary layout algorithms. The phenotypes, in contrast, are commonly represented by graph layouts, hence we can apply the same fitness evaluation methods as in previous work. The most obvious approach is the evaluation of aesthetic criteria [Pur02]. This process requires an evaluation graph G , to which all layout algorithms are applied, and a selection of criteria for judging the generated layouts.

Some authors used a linear combination of specific criteria as fitness function [BB00, BBS96, EM01]. For instance, given a graph layout L , the number of edge crossings $\kappa(L)$, and the standard deviation of edge lengths $\delta(L)$, the optimization goal could be to minimize the cost function $f(L) = w_c\kappa(L) + w_d\delta(L)$, where suitable scaling factors w_c and w_d are usually determined experimentally. The problem of this approach is that the values resulting from $f(L)$ have no inherent meaning apart from the general assumption “the smaller $f(L)$, the better the layout L .” As a consequence, the cost function can be used only as a relative measure, but not to determine the absolute quality of layouts.

Huang et al. proposed to compute the difference of aesthetic criteria x to their mean value \bar{x} among all considered layouts and to scale it by their standard deviation $\sigma(x)$ [HLH12]. The fitness is then determined as $f(L) = \sum_{i=1}^q \frac{x_i(L) - \bar{x}_i}{\sigma(x_i)}$, where x_1, \dots, x_q are the included criteria. This approach has the disadvantage that the mean values and standard deviations can vary greatly depending on the set of considered layouts. Therefore it cannot be used as an absolute quality measure.

An improved variant, proposed by several authors, is to normalize the criteria to the range between 0 and 1 [DS09, Pur02, RSOR98, Tet98, Vra09]. However, this is still not sufficient to effectively measure absolute layout quality. For instance, Tettamanzi normalizes the edge crossings $\kappa(L)$ with the formula $\mu_c(L) = \frac{1}{\kappa(L)+1}$ [Tet98]. For the complete graph K_5 , which is not planar, even the best layouts yield a result of $\mu_c(L) = 50\%$, suggesting that the layout is only half as good as it could be. Purchase proposed to scale the number of crossings against an upper bound κ_{\max} defined as the number that results when all pairs of edges that are not incident to the same node cross each other [Pur02]. Her formula is $\mu_c(L) = 1 - \frac{\kappa(L)}{\kappa_{\max}}$ if $\kappa_{\max} > 0$ and

3. Meta Layout

$\mu_c(L) = 1$ otherwise. Purchase herself notes that this definition “is biased towards high values.” For instance, the graph N14 used in her evaluations has 24 nodes, 36 edges, and $\kappa_{\max} = 558$. All layouts with up to 56 crossings would result in $\mu_c(L) > 90\%$. When tested with a selection of 28 layout algorithms, all of them resulted in layouts with less than 56 crossings (the best had only 11 crossings), hence the formula of Purchase would assign a very high fitness to all these generated layouts.

We propose new normalization functions that aim at well-balanced distributions of values among typical results of layout algorithms.

Definition 3.5 (Layout metric). A *layout metric* is a function μ that maps graph layouts L to values $\mu(L) \in [0, 1]$. Given layout metrics μ_1, \dots, μ_k with weights $w_1, \dots, w_k \in [0, 1]$, we compute the *fitness* of a graph layout L by

$$f(L) = \frac{1}{\sum_{i=1}^k w_i} \sum_{i=1}^k w_i \mu_i(L) .$$

In the following we describe some of the metrics we have used in conjunction with our proposed genotype representation and evolutionary algorithm. The goal of these metrics is to allow an intuitive assessment of the respective criteria, which means that the worst layouts shall have metric values near 0%, the best ones shall have values near 100%, and moderate ones shall score around 50%. The metrics should be parameterized such that this spectrum of values is exhausted for layouts that are generated by typical layout algorithms, allowing to clearly distinguish them from one another. Additionally to the metrics presented here, we adopt previously proposed metrics that already meet our requirements, e. g. for the number of edge bends and the number of edges pointing in a specific direction as given by Purchase [Pur02].

The basic idea behind each of our formulae is to define a certain *input split value* x_s such that if the value of the respective criterion equals x_s , the metric is set to a defined *output split value* μ^* . Values that differ from x_s are scaled towards 0 or 1, depending on the specific criterion. The advantage of this approach is that different formulae can be applied to the ranges below and above the split values, simplifying the design of metrics that meet the goals stated above. The approach involves several constants, which we

3.2. Optimal Layout Configuration

determined experimentally. Examples are shown in Table 3.1.

Let $G = (V, E)$ be a directed graph with a layout L . Let $n = |V|$ and $m = |E|$.

Number of crossings. Similarly to Purchase we define a virtual upper bound $\kappa_{\max} = m(m-1)/2$ on the number of crossings [Pur02]. We call that bound virtual because it is valid only for straight-line layouts, while layouts where edges have bend points can have arbitrarily many crossings. Based on the observation that crossings tend to be more likely when there are many edges and few nodes, we further define an input split value

$$\kappa_s = \min \left\{ \frac{m^3}{n^2}, (1 - \mu_c^*)\kappa_{\max} \right\}. \quad (3.1)$$

μ_c^* is the corresponding output split value, for which we chose $\mu_c^* = 10\%$. The exponents of m and n are chosen such that the split value becomes larger when the m/n ratio is high. We denote the number of crossings as $\kappa(L)$. Layouts with $\kappa(L) < \kappa_s$ yield metric values above μ_c^* , while layouts with $\kappa(L) > \kappa_s$ yield values below μ_c^* . This is realized with the formula

$$\mu_c(L) = \begin{cases} 1 & \text{if } \kappa_{\max} = 0, \\ 0 & \text{if } \kappa(L) \geq \kappa_{\max} > 0, \\ 1 - \frac{\kappa(L)}{\kappa_s}(1 - \mu_c^*) & \text{if } \kappa(L) \leq \kappa_s, \\ \left(1 - \frac{\kappa(L) - \kappa_s}{\kappa_{\max} - \kappa_s}\right) \mu_c^* & \text{otherwise.} \end{cases} \quad (3.2)$$

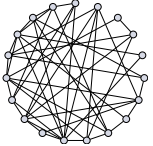
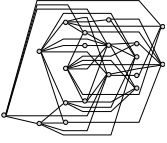


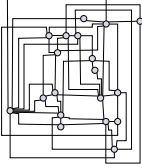
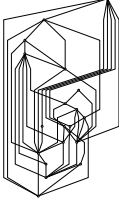
Area. Let $w(L)$ be the width and $h(L)$ be the height of the drawing L . The area required to draw a graph depends on the number of nodes and edges, hence we define a *relative area*

$$\alpha(L) = \frac{w(L)h(L)}{(n+m)^2} \quad (3.3)$$

that takes into account the number of elements in the graph. We square that number because we observed that many drawings of larger graphs require a disproportionately high area. We split the output values

3. Meta Layout

Table 3.1. Results of the five layout metrics presented here for six drawings of the same graph. The drawings have been generated with different layout algorithms and parameters: (a) circular, (b) layer-based, (c)/(d) two force-based methods, (e) planarization-based, and (f) upward planarization. The drawings are shown with different scales.

	Crossings μ_c	Area μ_a	Asp. Ratio μ_r	Length μ_l	Uniformity μ_u
(a) 	30.6%	91.4%	62.0%	18.5%	49.2%
(b) 	68.6%	82.7%	93.3%	12.5%	41.9%
(c) 	82.0%	96.0%	65.9%	55.5%	84.4%
(d) 	82.0%	97.6%	64.4%	87.5%	65.9%
(e) 	91.0%	90.8%	61.5%	19.4%	20.0%
(f) 	86.0%	12.4%	26.2%	5.0%	30.3%

3.2. Optimal Layout Configuration

at two points $\mu_a^* = 10\%$ and $\mu_a^{**} = 95\%$, with corresponding input split values α_{s1} and α_{s2} . Values below μ_a^* are met when $\alpha(L) > \alpha_{s1}$, values above μ_a^{**} are met when $\alpha(L) < \alpha_{s2}$, and values in-between are scaled proportionally. The constants α_{s1} and α_{s2} have been determined experimentally as 1000 and 50, respectively. We define the area metric as

$$\mu_a(L) = \begin{cases} \frac{\alpha_{s1}}{\alpha(L)} \mu_a^* & \text{if } \alpha(L) > \alpha_{s1}, \\ 1 - \frac{\alpha(L)}{\alpha_{s2}} (1 - \mu_a^{**}) & \text{if } \alpha(L) < \alpha_{s2}, \\ \left(1 - \frac{\alpha(L) - \alpha_{s2}}{\alpha_{s1} - \alpha_{s2}}\right) (\mu_a^{**} - \mu_a^*) + \mu_a^* & \text{otherwise.} \end{cases} \quad (3.4)$$

Aspect ratio. The aspect ratio $r(L)$ of a drawing is the ratio of its width $w(L)$ to its height $h(L)$. This measure is important to effectively display graphs on typical media such as computer screens or sheets of paper. We choose the *golden ratio* $r_g \approx 1.618$ as the optimal value for aspect ratios. The computation of a metric from this optimum is rather simple:

$$\mu_r(L) = \begin{cases} \frac{r_g}{r(L)} & \text{if } r(L) \geq r_g, \\ \frac{r(L)}{r_g} & \text{otherwise.} \end{cases} \quad (3.5)$$

Edge length. Many force-based layout algorithms aim at ideal edge lengths [Ead84]. Similar formulae as those used in these algorithms could be applied to compute a layout metric for the edge length. Here we propose a formula based on the average edge length $\bar{\lambda}(L)$ and an ideal edge length λ_{opt} :

$$\mu_l(L) = \begin{cases} \frac{\lambda_{opt}}{\bar{\lambda}(L)} & \text{if } \bar{\lambda}(L) \geq \lambda_{opt}, \\ \sqrt{\frac{\bar{\lambda}(L)}{\lambda_{opt}}} & \text{otherwise.} \end{cases} \quad (3.6)$$

We chose $\lambda_{opt} = 60$ experimentally. The square root for values below the ideal length serves to raise the metric result for drawings with very short edges.

Edge length uniformity. We measure this criterion with the standard deviation $\sigma_\lambda(L)$ of edge lengths and compare it against the average edge length

3. Meta Layout

$\bar{\lambda}(L)$, which we use as input split value. We define

$$\mu_u(L) = \begin{cases} \frac{\bar{\lambda}(L)}{\sigma_\lambda(L)} \mu_u^* & \text{if } \sigma_\lambda(L) \geq \bar{\lambda}(L), \\ 1 - \frac{\sigma_\lambda(L)}{\bar{\lambda}(L)} (1 - \mu_u^*) & \text{otherwise,} \end{cases} \quad (3.7)$$

where the output split value $\mu_u^* = 20\%$ corresponds to the metric value that results when the standard deviation equals the average.

Distance function. The difference between two solutions can be determined either on the genotype level or on the phenotype level. The latter means comparing the drawings of the two individuals, which can be done with *difference metrics* [BT00]. Such metrics, however, are rather expensive in terms of computation time. Therefore we propose a distance function that compares solutions by their genomes, which is much more efficient because it is independent of the size of the graphs given by the environment. The price for this speed improvement is a lower reliability of the computed distance values. As already mentioned, a low distance on the genotype level does not imply a low distance on the phenotype level because some algorithms are randomized and are likely to produce drawings with completely different topologies on each invocation. Furthermore, a high distance of genomes does not necessarily mean the corresponding drawings are different, since that depends on how the layout algorithms interpret their parameters. Some parameters may only have an influence on the drawings for graphs with specific properties.

The distance $d(X_1, X_2)$ of two genomes X_1 and X_2 is determined by the sum of the differences of all genes. We assume that for each gene in X_1 a corresponding gene of the same type is contained in X_2 . Given values g_1 and g_2 of two genes with the same type t , we define their difference as

$$d(g_1, g_2) = \frac{|g_1 - g_2|}{\sigma_t} \quad (3.8)$$

if t has integer or floating point values, where σ_t is the standard deviation assigned to the type t . Genes with other data types have no inherent ordering, therefore we define their difference as $d(g_1, g_2) = 0$ if $g_1 = g_2$, and $d(g_1, g_2) = 1$ otherwise.

3.2.2 Evolutionary Process

In this section we consider operations for an evolutionary algorithm, a popular method for searching large solution spaces. There are numerous variations of such algorithms. The method proposed here is meant as an example of a meta heuristic for meta layout and is used for a first experimental evaluation. A possible alternative is described in Section 3.2.4.

A *population* is a set of genomes. An *evolution cycle* is a function that modifies a population with four steps, which are explained below. The evolutionary algorithm executes the evolution cycle repeatedly, checking some terminating condition after each execution. Simple conditions for fully automatic optimization are to limit the number of iterations and to check whether the fitness of the best individual exceeds a certain threshold. Alternatively, the user can be involved by manually controlling when to execute the next evolution cycle and when to stop the process. The four steps of the evolution cycle are discussed in the following and exemplified in Figure 3.5.

1. *Recombination.* New genomes are created by crossing random pairs of existing genomes. A crossing of two genomes is created by crossing all their genes. Two integer or floating point typed genes are crossed by computing their average value, while for other data types one of the two values is chosen randomly. Only a selection of the fittest individuals is considered for mating.

When the parent genomes are different in the layout algorithm gene, the child is randomly assigned one of the parent values. As a consequence, the *active / inactive* statuses of the other genes of the child must be adapted such that they match the chosen algorithm A : each gene g is made active if and only if A supports the layout option associated to g .

2. *Mutation.* Genomes have a certain probability of mutating. A mutation is done by randomly modifying its genes, where each gene g has an individual mutation probability p_g depending on its type. We assign the highest p_g values to genes with integer or floating point values, medium values to genes with Boolean or enumeration values, and the lowest values to the layout algorithm and layout type genes. Let g be a

3. Meta Layout

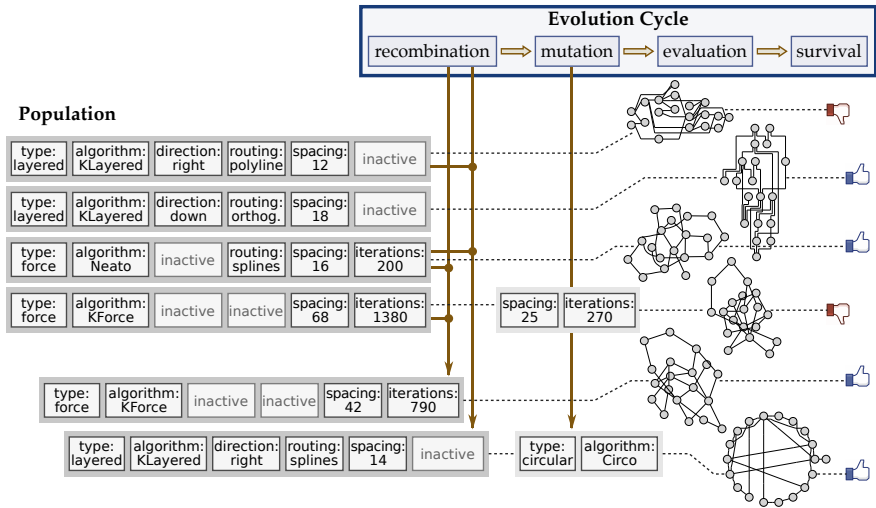


Figure 3.5. Evolutionary layout example: starting with a population of four genomes, two new genomes are created through recombination, two genomes are mutated, and four of the resulting genomes survive after their evaluation.

gene with value x . If the data type of g is integer or floating point, the new value x' is determined using a Gaussian distribution using x as its average and the standard deviation assigned to the gene type of g . If x' exceeds the upper or lower bound assigned to the gene type of g , it is corrected to a value between x and the respective bound. For genes with other types, which have no specific order, a new value is chosen based on a uniform distribution over the finite set of values, excluding the previous value. When the layout algorithm gene mutates, the *active* / *inactive* statuses of other genes must be updated as described for the recombination step.

3. *Evaluation.* A fitness value is assigned to each genome that does not have one yet (see Section 3.2.1), which involves executing the encoded layout algorithm in order to obtain a corresponding phenotype. The population is sorted using these fitness values.

3.2. Optimal Layout Configuration

4. *Survival.* Only the fittest individuals survive. Checking all genomes in order of descending fitness, we include each genome X in the set of survivors if and only if it meets the following requirements: (i) its fitness exceeds a certain minimum, (ii) the maximal number of survivors is not reached yet, and (iii) the distance of X to other individuals is sufficient. The latter requirement serves to support the diversity of the population. Comparing all pairs of individuals would require a quadratic number of distance evaluations, therefore we apply the distance function d introduced in Section 3.2.1 only to some random samples X' from the current set of survivors. In order to meet the third requirement, $d(X, X') \geq d_{\min}$ must hold for a fixed minimal distance d_{\min} .

Choosing metric weights. The fitness function discussed in Section 3.2.1 uses layout metrics μ_1, \dots, μ_k and weights $w_1, \dots, w_k \in [0, 1]$, where each w_i controls the influence of μ_i on the computed fitness. The question is how to choose suitable weights. Masui proposed to apply genetic programming to find a fitness function that best reflects the user's intention [Mas94]. The computed functions are evolved as Lisp programs and are evaluated with layout examples, which have to be rated as "good" or "bad" by the user. A similar approach is used by Barbosa and Barreto [BB01], with the main difference that the fitness function is evolved indirectly by modifying a set of weights with an evolutionary algorithm. Additionally, they apply another evolutionary algorithm to create concrete layouts of a given graph. Both algorithms are combined in a process called *co-evolution*: the results of the weights evolution are used for the fitness function of the layout evolution, while the fitness of the weights is determined based on user ratings of sample layouts.

We have experimented with two much simpler methods, both of which involve the user: (a) the user directly manipulates the metric weights with sliders allowing values between 0 and 1, and (b) the user selects good layouts from the current population and the metric weights are automatically adjusted according to the selection. This second method builds on the assumption that the considered layout metrics are able to compute meaningful estimates of the absolute quality of any given layout (see Section 3.2.1). The higher the result of a metric, the higher its weight shall be.

3. Meta Layout

Let $\bar{\mu}_1, \dots, \bar{\mu}_k$ be the average values of the layout metrics μ_1, \dots, μ_k for the selected layouts. Furthermore, let w_1, \dots, w_k be the current metric weights. For each $i \in \{1, \dots, k\}$ we determine a *target weight*

$$w_i^* = \begin{cases} 1 - \frac{1}{2} \left(\frac{1 - \bar{\mu}_i}{1 - \mu_w^*} \right)^2 & \text{if } \bar{\mu}_i \geq \mu_w^*, \\ \frac{1}{2} \left(\frac{\bar{\mu}_i}{\mu_w^*} \right)^2 & \text{otherwise,} \end{cases} \quad (3.9)$$

where μ_w^* is a constant that determines which metric result is required to reach a target weight of 50%. We chose $\mu_w^* = 70\%$, meaning that mediocre metric results are mapped to rather low target weights. The square functions in Equation 3.9 are used to push extreme results even more towards 0 or 1. The new weight of the layout metric μ_i is $w'_i = \frac{1}{2}(w_i + w_i^*)$, the mean value between the current weight and the target weight. For instance, if the weight is currently at $w_i = 50\%$ and the metric result is $\bar{\mu}_i = 90\%$, the target weight computes to $w_i^* = 94\%$ and the new weight is $w'_i = 72\%$, i. e. the weight is increased by 22% due to the very good result of the layout metric.

User interface. We have experimented with a user interface that includes both variants for modifying metric weights, shown in Figure 3.6. The window visualizes populations by presenting up to 16 small drawings of the evaluation graph, which represent the fittest individuals of the current population. 13 metrics are shown on the side of the window. The user may use the controls in the window to

- view the computed values of the layout metrics for an individual,
- directly set the metric weights,
- select one or more favored individuals for indirect adjustment of weights,
- change the population by executing an evolution cycle (“Evolve” button),
- restart the evolution with a new initial population (“Restart” button), and
- finish the process and select the abstract layout encoded in a selected individual (“Apply” button).

3.2. Optimal Layout Configuration

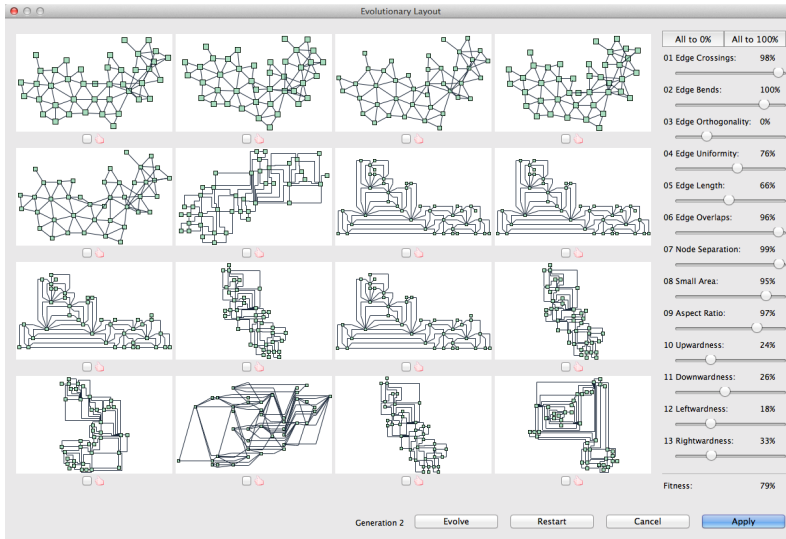


Figure 3.6. User interface for evolutionary meta layout, showing drawings for 16 individuals of the current population. The check box below each proposed graph drawing is used to select favored layouts for automatic adaption of metric weights. The sliders on the right offer direct manipulation of the weights.

The indirect method for choosing weights, which adapts them according to the user's selection of favored layouts, is in line with the *multidrawing* approach introduced by Biedl et al. [BMRW98]. The main concept of that approach is that the user can select one of multiple offered drawings without the need of defining her or his goals and preferences in the first place. The multidrawing system reacts on the user's selection and generates new layouts that are similar to the selected ones. In our proposed method, this similarity is achieved by adjusting the fitness function such that the selected layouts are assigned a higher fitness, granting them better prospects in the competition against other layouts.

3. Meta Layout

3.2.3 Evaluation

The evolutionary meta layout algorithm has been implemented and evaluated in KIELER (see Chapter 4). Our experiments included four layout algorithms provided by KIELER as well as five algorithms from the Graphviz library [GN00] and 22 algorithms from the OGDF library [CGJ⁺13]. The total number of genes in each genome was 79.

Execution time. We tested the performance of evolutionary meta layout on the set of 1277 graphs collected by North [DGL⁺97]. The tests have been executed with an Intel Xeon 2.5 GHz CPU. The population initially contained 16 genomes. The number of genomes created in the recombination operation was 84% of the population size, the mutation operation affected 60% of the total population, and 60% of the resulting population survived for the next evolution cycle. We measured the average execution time of a single evolution cycle by applying five iterations to each graph, which led to the results shown in Figure 3.7. The vast majority of time is spent in the evaluation step: on average 92% is taken by layout algorithm execution, and 8% is taken by metrics evaluation, while the computation time of the actual evolutionary operations is negligible. The average number of layout algorithm and layout metrics evaluations per evolution cycle was 27. The rather high execution time of between 0.5 and 4.5 seconds limits the number of evolution cycles that can be performed in an interactive environment. The consequence is that the evolutionary algorithm has to converge to an acceptable solution within few iterations. However, the evaluation step is very suitable for parallelization, since the algorithm evaluations are all independent. As seen in Figure 3.7, the execution time is significantly lower when run with multiple threads on a multicore machine (eight cores in this example). The average speedup was 2.6, with minimal and maximal execution times of 0.3 and 1.7 seconds.

Layout metrics. We evaluated the layout metrics proposed in Section 3.2.1 using the same set of graphs as for the execution time measurements. For each of these graphs, we created 100 random layout configurations, executed the respective layout algorithms to obtain concrete layouts, and computed

3.2. Optimal Layout Configuration

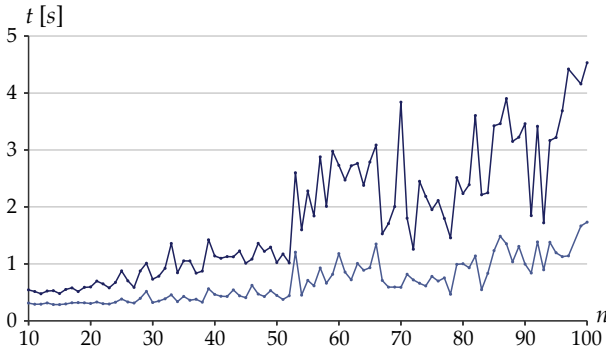


Figure 3.7. Execution time t of evolutionary meta layout plotted by number of nodes n with single threaded execution (darker line) and multithreaded execution (brighter line). For each value of n , the average of the results for all graphs with n nodes is shown.

the metric values for those layouts. In order to fulfill the goals stated in Section 3.2.1, the metrics should yield low values for the worst layouts and high values for the best layouts, hence, with an ideal formula, all values between 0 and 1 should occur when applied to layouts generated by layout algorithms. We evaluated this by measuring the standard deviations, minima, and maxima of the layout metric results. A uniform distribution over the range $[0, 1]$ has the standard deviation 0.289 (28.9%), the minimum 0%, and the maximum 100%. The values measured for the North graphs are shown in Table 3.2. These values are quite close to the ideal values of the uniform distribution, in particular those of the edge crossings and the edge length metrics. We conclude that our proposed formula for layout metrics computation are suitable as absolute quality measures for drawings generated by typical graph layout algorithms.

Evolutionary algorithm. We carried out three experiments in order to verify the effectiveness of the evolutionary approach. The experiments had different optimization goals: minimal number of edge crossings, maximal number of edges pointing from right to left, and optimal uniformity of

3. Meta Layout

Table 3.2. Results of layout metrics evaluations for the North graphs. $\bar{\mu}$ is the total average of the respective metric, $\sigma(\mu)$ is its standard deviation, $\bar{\sigma}(\mu)$ is the average of the standard deviations determined for each graph, $\bar{\mu}_{\min}$ is the average of the minimum values for each graph, and $\bar{\mu}_{\max}$ is the average of the maximum values for each graph.

Metric	$\bar{\mu}$	$\sigma(\mu)$	$\bar{\sigma}(\mu)$	$\bar{\mu}_{\min}$	$\bar{\mu}_{\max}$
Crossings	82.3%	27.6%	23.3%	9.4%	99.9%
Area	82.8%	23.7%	22.8%	5.1%	99.8%
Aspect Ratio	58.0%	21.5%	20.9%	7.7%	97.5%
Edge Length	47.7%	28.4%	26.3%	5.0%	98.1%
Uniformity	45.6%	24.1%	22.2%	13.0%	88.7%

edge lengths. In each experiment the corresponding layout metric was given a weight of 100%, while most other metrics were deactivated (except some basic metrics avoiding graph elements overlapping each other). The optimization goals were chosen such that they can be mapped to certain kinds of layout algorithms, allowing to validate the results according to prior knowledge of their behavior. 30 randomly generated graphs were used as evaluation graphs.

In the crossing minimization experiment, for 60% of the graphs a planarization-based algorithm was selected as the genome with highest fitness after three or less iterations. This confirms the intuitive expectation, since planarization methods are most effective in minimizing edge crossings. In the experiment that aimed at edges pointing from right to left, for 90% of the graphs a layer-based algorithm was selected as the genome with highest fitness after three or less iterations. Additionally, the layout option that determines the main direction of edges had to be set to *left*, which was accomplished in 83% of the cases. In the edge uniformity experiment, a force-based algorithm was selected for 63% of the graphs after three iterations, and for 73% of the graphs after six iterations (see Figure 3.8). This result matches the expectation, too, because force-based methods aim at drawing all edges with uniform length. In all experiments it could be observed that the average rating of genomes was consistently increasing

3.2. Optimal Layout Configuration

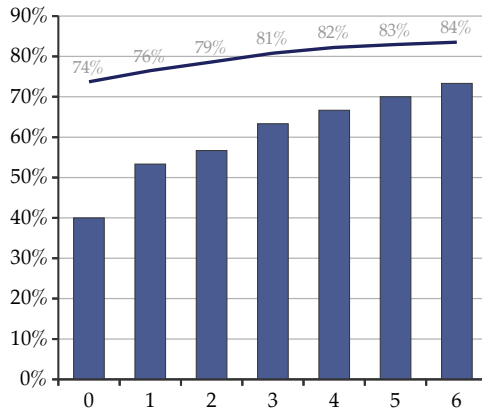


Figure 3.8. Result of the edge uniformity experiment. The line on top shows the fitness values of the best genomes for iterations 0 to 6 (horizontal axis), while the bars show the fractions of genomes that are set to force-type algorithms.

after each iteration, but this increase became smaller with each iteration. We conclude that our proposed evolutionary meta layout approach can effectively optimize given aesthetic criteria, and in most cases the kind of layout algorithm that is automatically selected is consistent with the intuition. A very relevant observation is that the process tends to converge very quickly, often yielding good solutions after few iterations, e.g. as illustrated in Figure 3.8. On the other hand, in some cases the computation is trapped in local optima, which could possibly be avoided by improving the parameters of the evolutionary computation.

User study. We have conducted a user study to determine the practical usefulness of our approach. The study is based on a set of 8 graphs, inspired by examples found on the web, with between 15 and 43 nodes and 18 to 90 edges. 25 persons participated in the study: four members of our research group, 17 computer science students, and four persons who were not involved in computer science. For novice users we expected that the evolutionary meta layout approach would lead to higher efficiency in graph readability compared to direct manipulation of layout configurations. We

3. Meta Layout

did not expect such an improvement for the research group members, who are experts in graph layout technology and are likely to have predetermined opinions about which layout configurations to use in certain contexts.

For each graph, the participants were presented three tasks regarding connectivity, e. g. finding the shortest path between two given nodes. The participants then had to find a layout configuration which they regarded as useful for working on the tasks. The test instructions encouraged the participants to improve the layout configuration until they were sure they had found a well readable layout.

Four of the graphs were treated with the user interface of the evolutionary meta layout, presented in Section 3.2.2 and named *EVOL* in the following, which evolves a population of layout configurations and lets users pick configurations by their previews. They were free to use both the direct and the indirect method for modifying weights. For the other four graphs, the participants were required to find layout configurations manually by choosing from a list of available layout algorithms and modifying parameters of the chosen algorithms (the *Layout View* presented in Section 4.2.1). For each participant we determined randomly which graphs to treat with *EVOL* and which to configure with the manual method, called *MANUAL* in the following. After the participants had accepted a layout configuration for a graph, they worked on the respective tasks by inspecting the drawing that resulted from the configuration. More details on the user experiment and its results are given in Appendix B.

After all graphs were done, the participants were asked 6 questions about their subjective impression of the evolutionary approach. The overall response to these questions was very positive: on a scale from -2 (worst rating) to 2 (best rating), the average ratings were 1.0 for the quality of generated layouts, 0.8 for their variety, 1.2 for the time required for finding suitable layouts, 0.6 for the effectiveness of manually setting metric weights, and 1.5 for the effectiveness of adjusting metric weights by favoring individuals. Most notably, the indirect adjustment of metric weights was rated much higher than their direct manipulation. This indicates that most users prefer an intuitive interface based on layout proposals instead of manually setting parameters of the fitness function, since the latter requires to understand the meaning of all layout metrics.

3.2. Optimal Layout Configuration

The average rate of correct answers of non-expert users to the tasks was 77.4% for MANUAL and 79.8% for EVOL. The average time used to work on each task was lower by 7.5% with EVOL (131 seconds) compared to MANUAL (142 seconds). These differences are not statistically significant: the p -values resulting from a t -test on the difference of mean values are 29% for the correctness of answers and 23% for the working time. A more significant result ($p = 8.3\%$) is obtained when comparing the differences of EVOL and MANUAL working times between expert users and non-expert users. In contrast to the non-experts, expert users took more time to work on the tasks with EVOL (126 seconds) compared to MANUAL (107 seconds). Furthermore, the average rate of correct answers of expert users was equal for both methods. This confirms the assumption that the method proposed in this paper is more suitable in applications used by persons without expert knowledge on graph drawing.

Many participants commented that they clearly preferred EVOL over MANUAL. It could be observed that novice users were overwhelmed by the number of configuration parameters shown for the manual method. In many cases, they stopped trying to understand the effects of the parameters after some unsuccessful attempts to fine-tune the layout. Therefore the average time taken for finding a layout was lower for MANUAL (129 seconds) compared to EVOL (148 seconds). For the EVOL interface, on the other hand, similarly frustrating experiences were observed in few cases where the evolutionary algorithm apparently ran into local optima that did not satisfy the users' expectations. In these cases users were forced to restart the process with a new population.

The average number of applied evolution cycles was 3.1, which means that in most cases the participants found good solutions after very few iterations of the evolutionary algorithm. Furthermore, we measured the index of the layout chosen for working on the tasks on a scale from 0 to 15. The layout with index 0 has the highest fitness in the current population, while the layout with index 15 is the one with lowest fitness from the 16 fittest individuals. The average selected index was 2.3, a quite low value, suggesting that the computed fitness has a high correlation with the perceived quality of the layouts.

A further result could be drawn from the experiment by analyzing the

3. Meta Layout

frequency of selected layout types. 54% of the selected layouts were drawn with layer-based methods, mostly with *KLay Layered* from the KIELER library, *Dot* from Graphviz, and *Sugiyama* from OGDF. The second group of most popular layouts, selected in 32% of the cases, were the orthogonal algorithms *Planarization* and *Mixed Model* from OGDF. Force-based algorithms were selected only in 9% of the cases.

3.2.4 An Alternative Method: Successive Optimization

In this section we discuss an alternative meta layout method, named *successive optimization*, and compare it with the evolutionary approach. The basic principle is simple: find a suitable ordering o_1, \dots, o_k of the available layout options, then find an optimal value for each option in this order by testing a number of different values. Optimality is evaluated with respect to a chosen set of layout metrics or through direct selection by the user.

The first option o_1 shall always equal the special option for selecting a layout algorithm. Furthermore, the ordering shall satisfy $i > j$ for all options o_i that depend on another option o_j . This can be done with a topological sorting of the dependency graph (see Section 3.1.3). Given an ordering o_1, \dots, o_k , we generate *initial genes* g_1, \dots, g_k according to the representation introduced in Section 3.2.1. The value of each gene g_i is set to the default value of o_i . Alternatively, the initial values can also be derived from any given configuration, e. g. one computed with the evolutionary algorithm described in Section 3.2.2.

The layout algorithm gene g_A has no default value. Its value is determined in the first step of the optimization process, where all available layout algorithms are executed once with their parameters set to the initial values encoded in g_2, \dots, g_k . This results in a set of different layouts, of which one is selected either automatically using the fitness function of Section 3.2.1, or interactively following the *multidrawing* approach [BMRW98]. Similarly to the user interface of the evolutionary method, the interactive selection can be backed by the results of the fitness computations. The layout algorithm that generated the selected layout is stored in g_1 .

All other options o_i are processed using the layout algorithm determined in the first step. Options that are not supported by that algorithm and those for which the dependencies are not satisfied are omitted. Such dependencies

3.2. Optimal Layout Configuration

refer to values of layout options that have already been processed. If the data type of o_i is Boolean or enumeration, all possible values are tested, while for integer and floating point types a fixed number of K test values is generated randomly. These random values are chosen with a normal distribution around the initial value encoded in g_i , similarly to the mutation operation described in Section 3.2.2. A concrete layout is created for each test value using the already determined option values g_1, \dots, g_{i-1} and the initial values g_{i+1}, \dots, g_k . In the same way as done for the layout algorithm selection, the fittest layout is selected either automatically or interactively, and the option value that corresponds to the selected layout is stored in g_i .

Comparison. The main advantage of the successive optimization method, called *SucOPT*, compared with the evolutionary method, called *EvOL*, is that the search method is more structured and less randomized, and thus more comprehensible and reproducible for users. However, when used with an interactive interface, it requires a rather high number of choices from the user. While *EvOL* shows one selection dialog (Figure 3.6) after each evolution cycle, *SucOPT* requires a dialog for each layout option that is supported by the layout algorithm chosen in the first step. Using the algorithms from the three libraries considered in the evaluation, the average number of selection dialogs is 8.9, the minimum is 4, and the maximum is 18. In contrast, the average number of dialogs shown for one graph in the user study of *EvOL* was 4.1 (min. 1, max. 15). In that context users were free to stop the search as soon as they found a suitable layout. In a similar way the *SucOPT* user interface could offer an option to stop the search after step i and apply the default values of the initial genes g_{i+1}, \dots, g_k in order to shorten the selection process.

The computation speed of *SucOPT* depends on the number of layout algorithm invocations for generating the concrete layouts to be either presented to the user or analyzed automatically. For integer and floating point typed options this number is determined by the constant K . For $K = 16$, the average number of invocations of the considered algorithms is 121, which corresponds to approximately 4 evolution cycles with *EvOL* using the parameters as applied for the user study. The minimal number of invocations is 66 (2 evol. cycles) and the maximum is 276 (8 evol. cycles). Hence the computation speed of *SucOPT* is similar to that of *EvOL*.

Part II

From Theory to Practice

Integration in Eclipse

Eclipse is widely known as an integrated development environment (IDE) for Java, but it is much more than that: Gamma and Beck state the goals to “give the users an empowering computing experience and provide a learning environment as a path to greater power” [GB04]. With this premise, The Eclipse Foundation¹ now hosts numerous projects extending Eclipse towards different programming languages, platforms, and development processes. Among these, the Eclipse Modeling Project² provides tools for all stages of model-driven engineering: abstract syntax development, concrete syntax development, model creation, and model transformation [VKEH06].

The foundation is laid by the Eclipse Modeling Framework (EMF), which enables to create meta models in a standard form that is supported by all other Eclipse-based modeling tools [SBPM09]. The Graphical Modeling Framework (GMF) and Graphiti both offer graph-based concrete syntax development using the Graphical Editing Framework (GEF), an abstract implementation of the model-view-controller (MVC) pattern. Where text-based concrete syntax is required, Xtext offers automatic generation of parsers and serializers from grammar specifications. Such textual formats can be used as a replacement for the XML Metadata Interchange (XMI), the standard format of EMF models, and are particularly attractive in the context of domain-specific languages (DSLs) [FEK08].

Despite the success of the Eclipse Modeling Project, its tools offer only little support for automatic graph layout. A layer-based algorithm implementation with very few options and features is included in Draw2D, the basic drawing layer of GEF, and offered in the user interface of GMF. A few

¹<http://www.eclipse.org/>

²<http://www.eclipse.org/modeling/>

4. Integration in Eclipse

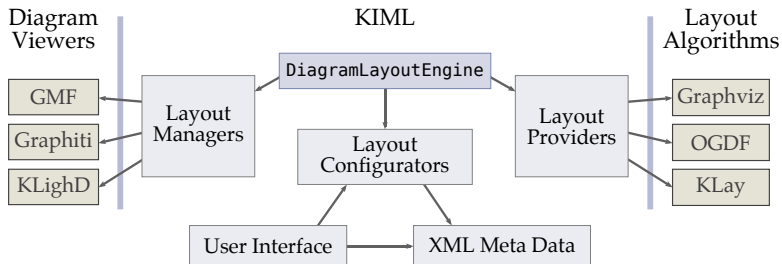


Figure 4.1. Main components of KIML and their interdependencies; cf. the general concepts depicted in Figure 1.1 (p. 4) and Figure 3.1 (p. 105). The components are explained in the remainder of this chapter.

further algorithms are offered by *Zest*, a subcomponent of GEF for visualizing data [BBS04]. The usefulness of these algorithms is rather limited when it comes to models with domain-specific requirements such as data flow diagrams (see Chapter 2) or class diagrams.

One principal goal of the KIELER project is to satisfy the demand for high-quality automatic layout in the context of Eclipse-based modeling tools. In this chapter I present my contributions towards this goal, including implementations of the concepts discussed in Part I of this thesis. The infrastructure is implemented in the project KIELER Infrastructure for Meta Layout (KIML), of which an overview is shown in Figure 4.1. This is a realization of the multiple front-ends / multiple back-ends concept outlined in Section 1.1: arbitrary diagram viewers can be connected to a large number of layout algorithms through a consistent interface. In contrast to previously proposed automatic layout extensions of tools such as Marama [YHG11] or DIAMETA [MM10a], here we do not target a specific visual modeling framework, but we provide automatic layout to the whole landscape of Eclipse-based software. The major challenge in the realization of this goal is to find suitable levels of abstraction that enable the required generality and at the same time tame the complexity of the system.

Section 4.1 begins with a description of the API of KIML, followed by an overview of the UI in Section 4.2. We discuss the connection of existing diagram viewers and editors to the layout infrastructure in Section 4.3, and

the connection of existing layout algorithm libraries as well as concepts for successful Java-based implementations in Section 4.4. The development of layout algorithms in Eclipse is supported by additional tools, which are described in Section 4.5.

4.1 Programming Interface

The driving software quality criteria for the development of KIML were *efficiency*, *simplicity*, and *flexibility*. This should enable to have fast response times even for very large graphs, to obtain good results after short development times, and to adapt the layouts to the specific needs of an application.

Efficiency is a topic that must be considered in all stages of the layout service, from the graph extraction over layout configuration to the actual layout algorithms. Measurements showing the high efficiency of the KIML interfaces are presented in Section 4.1.3.

Here the main approach to simplicity is to offer one common interface for the automatic layout of diagrams, irrespective of the particular kind of diagram or the layout algorithm to be applied. This is realized in the class `DiagramLayoutEngine`, which provides the following method:

```
public LayoutMapping layout(IWorkbenchPart workbenchPart,
    Object diagramPart, ILayoutConfig... configurators)
```

The argument `workbenchPart` indicates for which part of the Eclipse workbench the layout is requested, i. e. for which diagram viewer or editor. The layout algorithms can be focused on a specific subgraph by passing an element `diagramPart` of the respective diagram. If that argument is `null`, the whole graph is processed, otherwise only the subgraph corresponding to the given element is processed. The variable-length argument `configurators` can be used to specialize the layout configuration by passing one or more layout configurators (see Section 3.1.2). If this argument is omitted, only the default configurators are applied, which is elaborated further in Section 4.1.2. The result of the `layout` method is a mapping between diagram elements and graph elements, which is explained in Section 4.3. Invoking automatic layout on the diagram contained in a workbench part `p` can be as simple as

4. Integration in Eclipse

```
DiagramLayoutEngine.INSTANCE.layout(p, null);
```

The Eclipse platform offers sophisticated concepts for ensuring the flexibility of applications. Most importantly, the component model of the OSGi Service Platform is used to manage a set of *plug-ins* [GHM⁺05], called *bundles* in the OSGi specification [OSG03]. Each plug-in may contribute library code, services, or additions to the user interface. Multiple plug-ins are independent of each other unless they declare dependencies in order to access the code and services of other plug-ins. Furthermore, Eclipse allows to define *extension points*, which are XML-based interfaces to which arbitrarily many plug-ins may contribute extensions [GB04, dRB06, Bol03]. Extension points are used by KIML for several services, such as registering layout algorithms (Section 4.1.1) or specifying layout configurations (Section 4.1.2). The basic plug-ins of the layout infrastructure are the following.

- ▷ de.cau.cs.kieler.core
A collection of useful classes that are reused in many KIELER plug-ins.
- ▷ de.cau.cs.kieler.core.kgraph
An EMF-based implementation of the KGraph meta model introduced in Section 3.1.1.
- ▷ de.cau.cs.kieler.kiml
Basic interfaces and extension points for layout algorithms and layout configuration.
- ▷ de.cau.cs.kieler.kiml.service
Service classes for access to meta data contributed to the extension points, DiagramLayoutEngine class, and interfaces for connection of diagram viewers and editors.

4.1.1 Meta Data

In order to ensure flexibility, the meta data on layout algorithms and layout options, specified in Section 3.1.3, must not be hard-coded into the software, but should be available in a generic and extensible form. Which concrete form to choose depends on the platform in which the

Listing 4.1. XML declaration of the “Layout Algorithm” option.

```

<extension
  point="de.cau.cs.kieler.kiml.layoutProviders">
  <layoutOption
    appliesTo="parents"
    description="Select the layout algorithm to execute"
    id="de.cau.cs.kieler.algorithm"
    name="Layout Algorithm"
    type="string">
  </layoutOption>
</extension>

```

layout algorithms are integrated. In the Eclipse integration of KIML, meta data are collected through an extension point with the identifier `de.cau.cs.kieler.kiml.layoutProviders`. Contributions are written in the plug-in XML manifest file named `plugin.xml`. For instance, the declaration of the layout option for selecting the layout algorithm is shown in Listing 4.1. This form of meta data is a very natural choice for Eclipse-based applications, hence it is quickly understood by developers who are experienced with the Eclipse platform.

The contributions to the extension point are made available by the class `LayoutMetaDataService`. The meta data on layout algorithms, options, and types is stored therein with separate hash maps using their identifier strings as keys. As depicted in Figure 4.2, the class `LayoutAlgorithmData` represents layout algorithms, and `LayoutOptionData` represents options.

Although the required information on layout options can be derived completely from their XML specifications, handling the options on the Java code level is much easier if the most important parts of that information are duplicated such that they are available already at compile time. This is realized with the *property* mechanism, which publishes the data type, the identifier string, the default value, and the lower and upper bound of each layout option. The generic interface `IProperty<T>`, shown in Figure 4.3, is the core of this mechanism, where the generic parameter `T` corresponds to the data type. Due to the type erasure of Java generics, that data type

4. Integration in Eclipse

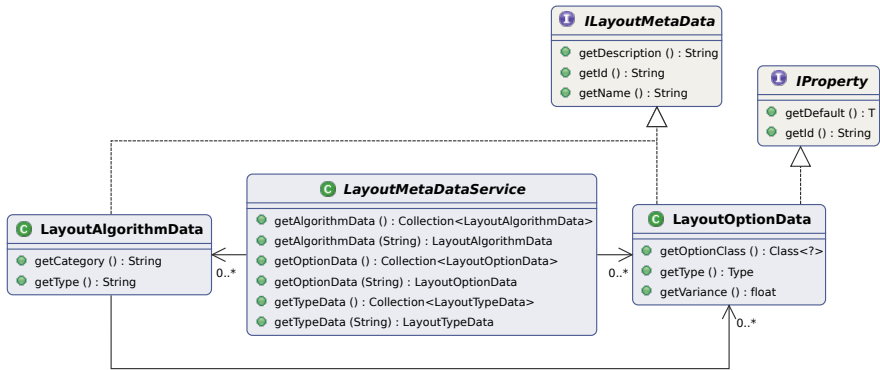


Figure 4.2. Class diagram of the meta data on layout algorithms and options (cf. Section 3.1.3).

can be checked at compile time, but not at runtime, which is sufficient in most situations. The class `LayoutOptions` contains static declarations of the general layout options defined by KIML using instances of `Property<T>`, the base implementation of the `IProperty<T>` interface. Each layout algorithm may extend these general options by defining its own properties in a similar way. The value of a property can be queried from a *property holder*, represented by the interface `IPropertyHolder`, which declares the following two methods:

```

<T> T getProperty(IProperty<T> property);
<T> void setProperty(IProperty<? super T> property, T value);

```

While `getProperty(...)` retrieves the current value of the given property, `setProperty(...)` stores a new value for it. Each of these methods has a local type parameter `T` for ensuring type safety: the return value of `getProperty(...)` as well as the argument value passed to `setProperty(...)` must conform to the data type of the given property at compile time.

As seen in Figure 3.2(a) (p. 107), `EMapPropertyHolder`, which is part of the KGraph meta model, implements the `IPropertyHolder` interface. The meta model class that stores the abstract layouts of graph elements, `KLayoutData`, inherits from that property holder implementation. Given a `KNode` instance

4.1. Programming Interface

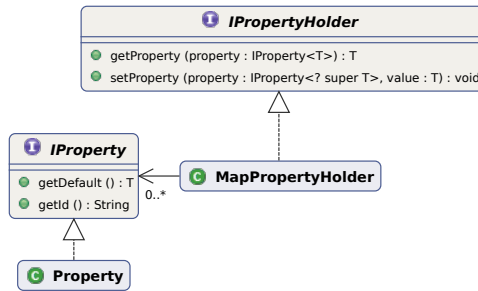


Figure 4.3. Class diagram of the property mechanism.

named node, its abstract layout can be accessed through its attached layout data holder, e. g. setting the active layout algorithm with the statement

```
node.getData(KLayoutData.class).setProperty(
    LayoutOptions.ALGORITHM,
    "de.cau.cs.kieler.klay.layered");
```

The current value of the option for the overall spacing between nodes can be read with the statement

```
float spacing = node.getData(KLayoutData.class).getProperty(
    LayoutOptions.SPACING);
```

Since for this property a default value is defined, the statement is guaranteed to never return `null` and thus the unboxing of the `Float` value to the type `float` is valid.

The class `LayoutOptionData`, representing meta data on layout options read from XML-based extensions, implements the `IProperty<Object>` interface, hence the following statement has the same effect as the last one:

```
float spacing = (Float) node.getData(KLayoutData.class)
    .getProperty(LayoutMetaDataService.getInstance()
        .getOptionData("de.cau.cs.kieler.spacing"));
```

The type cast to `Float` is necessary because the generic type of the property returned by `getOptionData(...)` is unknown at compile time. From this example it is evident that the static declarations of properties in the `LayoutOptions` class, used in the first variant of the above statement, can help making more

4. Integration in Eclipse

readable code compared to the access to meta data read from XML files, used in the second variant. According to my experience, this advantage outweighs the additional effort of duplicating parts of the meta data of layout options.

4.1.2 Layout Configuration

The layout configuration concept introduced in Section 3.1.2 is based on *layout configurators*, i. e. algorithms that compute abstract layouts. The main interface for such configurators is `ILayoutConfig`, shown in Figure 4.4. It declares a method

```
Collection<IProperty<?>> getAffectedOptions(  
    LayoutContext context);
```

for listing the layout options for which the configurator provides specific values, and a method

```
Object getOptionValue(LayoutOptionData optionData,  
    LayoutContext context);
```

for retrieving the value for a given option. `LayoutContext` specifies a focus on one element of a diagram using references to the graph element and the corresponding representation in the diagram viewer. An instance of `LayoutContext` corresponds to a graph element $e \in \mathcal{E}_G$ of the mathematical definition in Section 3.1.2, while an instance of `LayoutOptionData` corresponds to a layout option $o \in \Omega$. The method `getOptionValue(...)` delivers the result $\lambda(e)(o)$ of the abstract layout λ computed by the layout configurator.

The combination $\lambda_1 \circ \dots \circ \lambda_k$ of multiple abstract layouts $\lambda_1, \dots, \lambda_k$ is realized by `CompoundLayoutConfig`, which holds a list of configurators sorted by their priorities and delegates all requests to the contained configurators. The results of `getAffectedOptions(...)` is the union of the sets of options returned by the contained configurators, while `getOptionValue(...)` gives the value of the configurator with highest priority that returns a non-null value.

The application of a set of configurators to a specific graph is done by the class `LayoutOptionManager`. Given a `KGraph`, it collects the applicable configurators into a `CompoundLayoutConfig` instance and transfers the resulting

4.1. Programming Interface

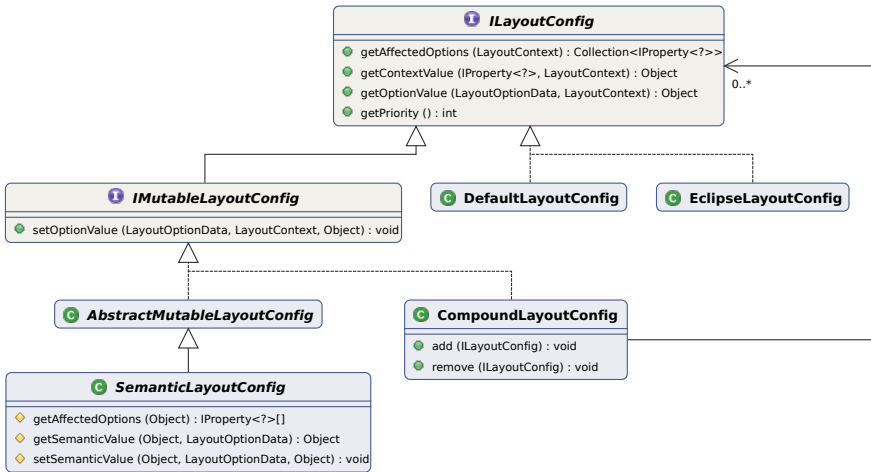


Figure 4.4. Class diagram of the main layout configuration interfaces.

combined abstract layout λ_C to the graph. The abstract layout is evaluated by calling $\text{getAffectedOptions}(e)$ for each graph element $e \in \mathcal{E}_G$, which results in a set O_e of affected options, and then calling $\text{getOptionValue}(o, e)$ for each $e \in \mathcal{E}_G$ and each $o \in O_e$. The returned values x are attached to the KLayoutData of each graph element using $\text{setProperty}(o, x)$.

Some configurators allow to modify their abstract layout by implementing the `IMutableLayoutConfig` interface, a subinterface of `ILayoutConfig`. Among some others, this additional interface declares a method

```
void setOptionValue(LayoutOptionData optionData,
    LayoutContext context, Object value);
```

allowing to store a new value for the given layout option and the graph element represented by the given context descriptor. The Layout View presented in Section 4.2.1 uses this interface for modifying the active configuration for the currently viewed model.

Configuration of default values. A default layout configuration that produces good results for a large range of applications is very important, since

4. Integration in Eclipse

it is responsible for the first impressions many users will have of the layout infrastructure. However, tool developers often have to adapt the configuration to the specific needs of their applications. The combination of these requirements can be realized with layout configurators.

The class `DefaultLayoutConfig` implements the configurator with lowest priority, providing fixed default values for all layout options. Layout algorithms may override these values with their own defaults, which are captured in their meta data and considered in the default configurator. Adaptations for specific applications can be done using an extension point with the identifier `de.cau.cs.kieler.kiml.layoutConfigs`. Contributions to the extension point consist of three parts: the identifier of a layout option, the value to assign, and the qualified name of a class specifying to which elements the configuration is applied. This class can either represent elements of the abstract syntax or elements of the concrete syntax. In the former case, the name of a meta model class is given, which can be related with multiple concrete syntax elements. For instance, when two diagram viewers operate on the same meta model, the layout configuration is applied to both viewers if it is assigned to a meta model element. By specifying a concrete syntax element as target of the configuration, in contrast, the option value is applied only to the diagram viewer that uses that concrete syntax. For GMF *edit parts* are used as concrete syntax elements (see Section 4.3.1), while for Graphiti *Pictogram elements* are used (Section 4.3.2). An example is shown in Listing 4.2.

An alternative to directly specifying classes from the abstract or concrete syntax is to use *diagram types*. A diagram type consists of an identifier string and is assigned through a special layout option. The diagram type identifier can be used in the `class` attribute of static configurations in place of a concrete class name. Configurations specified for a diagram type are applied to all diagram elements to which that same diagram type is assigned. Typical diagram types are class diagrams, data flow diagrams, or statecharts.

Another use of diagram types is to decouple declarations of layout configurations for diagram viewers from the implementations of layout algorithms. A layout algorithm may specify a *support priority* for a diagram type in order to indicate that it is specially suited for graphs of that type.

Listing 4.2. XML declaration of layout configurations with an abstract syntax class (first entry) and a concrete syntax class (second entry, shortened package name).

```

<extension
  point="de.cau.cs.kieler.kiml.layoutConfigs">
  <staticConfig
    class="org.eclipse.emf.ecore.EPackage"
    option="de.cau.cs.kieler.spacing"
    value="40">
  </staticConfig>
  <staticConfig
    class="org.eclipse.emf...EClassESuperTypesEditPart"
    option="de.cau.cs.kieler.edgeType"
    value="GENERALIZATION">
  </staticConfig>
</extension>

```

The priority value, given as an integer number, is used to determine the most suitable algorithm for a diagram type. If a diagram type is assigned to a graph viewer and no specific layout algorithm is configured for it, the most suitable layout algorithm is selected automatically. With this technique, developers of a diagram viewer do not need to specify any direct reference to a layout algorithm.

Configurations contributed to the `de.cau.cs.kieler.kiml.layoutConfigs` extension point are managed by the class `LayoutConfigService` and made available in the meta layout process with `EclipseLayoutConfig`.

Semantic configurators. In some situations a static assignment of layout option values to diagram element classes is not sufficient, but an analysis of the domain model, a.k.a. *semantic* model, is required instead. This is realized with specialized subclasses of `SemanticLayoutConfig`, which are assigned to classes of the domain meta model using the same extension point as for the static configurations. Such a subclass must be implemented by the tool developer; the concrete implementation may include arbitrary checks on instances of the domain model. The domain model analysis is performed dynamically upon each invocation of the layout configuration.

4. Integration in Eclipse

Listing 4.3. Annotation of a textually specified actor with layout directives.

```
@portConstraints FIXED_SIDE
@minWidth 20.0
@minHeight 15.0
entity IdentityActor
{
    @portSide WEST
    port Input;

    @portSide EAST
    port Output;
}
```

One important use of semantic configurators is to transfer annotations of the domain model into the layout configuration. This is very useful for models that also have a textual concrete syntax in addition to the graphical variant. As shown in the example of Listing 4.3, directives for controlling layout algorithms can be written as annotations of textual model elements. This approach can be used to optimize the layout of graphical notations that are automatically synthesized from the textual specification (see also Section 4.3.3). The abstract layouts are stored together with the original models, which is very intuitive and does not require any additional user interface.

4.1.3 Performance

We measured the performance of the KIML layout interface by executing the `layout(...)` method of `DiagramLayoutEngine` for 100 random graphs generated with the KEG editor (see Section 4.5.1). The result is shown in Figure 4.5. All graphs were processed with the KLayout Layered layout algorithm (see Section 4.4.1), which took 46.6% of the execution time on average. 4.6% of the time were taken for deriving the graph structure from the graph editor, 6.9% were taken for the layout configuration with `LayoutOptionManager`, and 13.5% were taken for applying the computed layout to the editor. The remaining time was used for further intermediate steps of the layout process.

4.1. Programming Interface

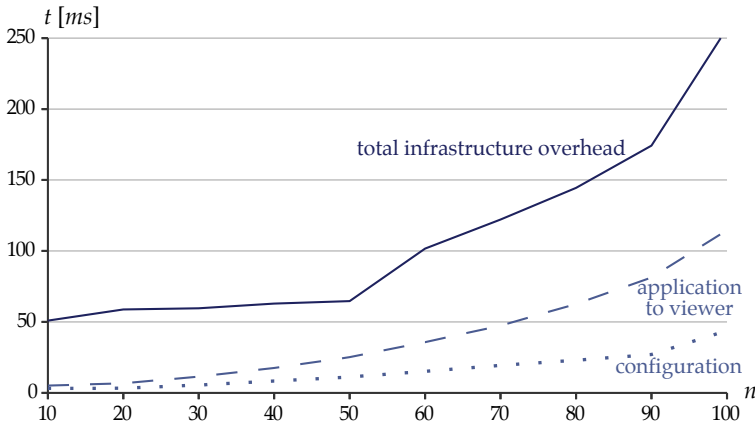


Figure 4.5. Execution time t in milliseconds of the KIML layout interface plotted by number of nodes n . The solid line shows the execution time of a layout invocation through the `DiagramLayoutEngine` class, without the time required by the actual layout algorithm. The dashed line shows the time for applying the computed layout to the diagram viewer. The dotted line shows the execution time of `LayoutOptionManager` for layout configuration.

The layout configuration involved six configurators.

The results demonstrate that the overhead of the KIML layout interface is small enough for use in interactive applications, where fast responses to layout requests are required. Even for graphs with 100 nodes the average overhead was only a quarter of a second. In particular, the time spent for layout configuration is negligible: the minimal and maximal measured values were 3 ms and 88 ms, respectively. A considerably higher portion of time is used for the interaction with the graph editor, which is based on GMF (see Section 4.3.1). This interaction can be improved with the *transient views* concept (see Section 4.3.3), which aims at minimizing the effort of layout algorithm invocations.

4. Integration in Eclipse

4.2 User Interface

The user interface of Eclipse is based on the *workbench*, which is a window composed of so-called *workbench parts*. A workbench part is either an *editor part* or a *view part*. Editor parts are linked with resources such as files and present the resource contents in a way that users can edit them.

The programmatic interface of KIML is separated from its user interface so that tool developers can define specialized interfaces for their applications. A standard UI is provided in the plug-in `de.cau.cs.kieler.kiml.ui`. This plug-in is not required by the other parts of KIML, so it can be loaded optionally if the standard UI is desired. The main goal of the KIML UI is to integrate seamlessly into the Eclipse workbench by offering elements that are quickly understood by Eclipse users.

The main UI element is a simple button with the icon



which is available in the Eclipse toolbar. The button is linked with the key combination `ctrl+R L` (`cmd+R L` on Mac OS), i. e. first `ctrl+R`, then `L`. When the button is activated, the `layout(...)` method of `DiagramLayoutEngine` is invoked passing the currently active workbench part and the selected diagram elements as arguments.

4.2.1 Layout View

Eclipse has a special view named *Properties* which lists properties of the currently selected element, e. g. for modifying attribute values of the corresponding domain model element. A similar view, named *Layout*, is offered by KIML for directly manipulating the abstract layout of a graph. As shown in Figure 4.6, the view has two columns, one with names of layout options and one with their assigned values. The values are of different data types, which are represented with blue icons. When an option is selected, its description is shown in the status bar of the workbench. The view lists only options that are supported by the currently active layout algorithm. Furthermore, it reacts to changes of the selection in the workbench. Whenever a diagram viewer or a contained element thereof is selected, the content of

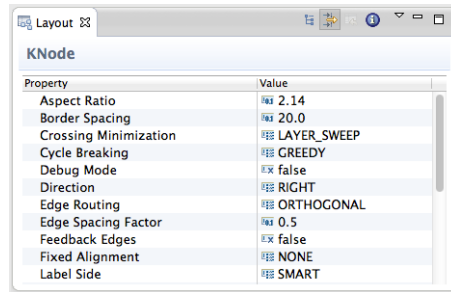


Figure 4.6. The Layout View shows the currently active layout configuration.

the Layout View is updated according to the active layout configuration of that element.

The displayed values of layout options are computed with a compound layout configurator as explained in Section 4.1.2. Therefore, these values may originate from different sources: some may be directly attached to the selected diagram element, some may be inferred from general configurations for the corresponding classes of elements, and some may be left at the default values of the respective options. Which values are displayed depends on the relative priorities of the involved layout configurators. A value can be modified by clicking on it and either editing it with the keyboard or selecting one of the possible values (e. g. for enumerations). The new value is passed to the compound configurator, which searches for contained instances of `ImmutableLayoutConfig` allowing to store the value in some way. Usually the storage is carried out by a configurator class contributed specifically for the selected diagram viewer (see Section 4.3).

When the “Layout Algorithm” option is selected in the Layout View, the dialog shown in Figure 4.7 is presented to the user. It lists the available layout algorithms grouped by their layout types and displays detailed descriptions and preview images of the algorithms. By this means, users are equipped with a useful overview with integrated documentation.

4. Integration in Eclipse

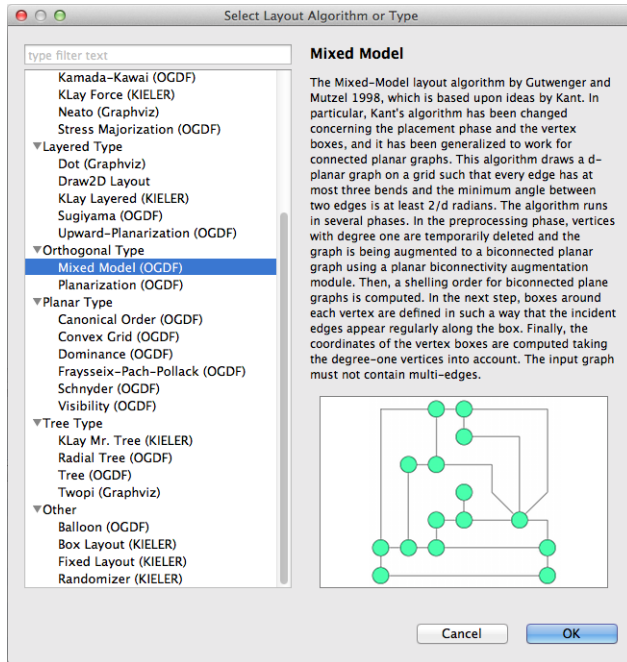


Figure 4.7. This dialog lists the available layout algorithms grouped by their layout types.

4.2.2 Preference Page

Eclipse allows to adapt its behavior and appearance in an extensible set of *preference pages*. Figure 4.8 shows the preference page of KIML, titled “KIELER Layout”. It allows to set certain general options, e.g. whether to animate the transitions to new computed layouts or whether to show the progress of layout algorithms during their execution, and to set default values for layout options. These values have higher priority than those configured in the extension points (see Section 4.1.2), but lower priority than those set directly in the Layout View.

A layout option is configured by adding an entry to the table shown in

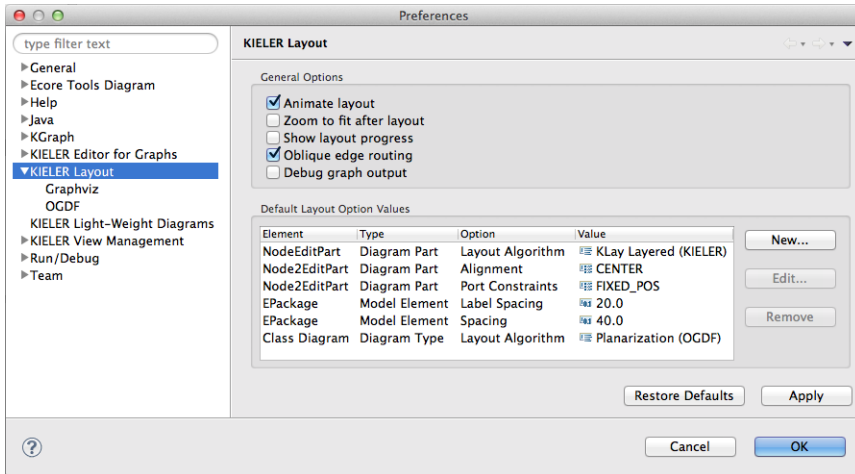


Figure 4.8. The preference page for KIML allows to modify default values of layout options.

Figure 4.8. An entry corresponds to one row of the table and consists of an element identifier, an element type, a layout option identifier, and the assigned value. Similarly to the extension point, the element type can be chosen from

- *model elements*, referring to classes of the domain meta model (the abstract syntax),
- *diagram parts*, referring to classes of the diagram viewer (the concrete syntax), and
- *diagram types* contributed to the extension point.

For a model element or a diagram part, the element identifier is the qualified class name, while for a diagram type it is the diagram type identifier. Once configured in the preference page, the layout option values are applied to all layouts invoked on instances of the specified element classes or diagram types.

4. Integration in Eclipse

4.3 Connection to Diagram Viewers

A diagram viewer is part of the *front-end* of the application. Its purpose is to graphically represent the underlying domain model. In our context, such a representation is based on a graph structure, but usually it contains more kinds of elements that may or may not be relevant for the automatic layout process, e. g. text labels, comments, symbols, decorative shapes, or other visual artifacts. The main problem is how to extract the actual graph structure from all this information. Of course a generic solution would be desirable, allowing to connect automatic layout to any viewer without further adaptation, but even the two diagramming frameworks based on GEF, namely GMF and Graphiti, have so different concepts that they must be treated with different approaches. Therefore we need a mechanism that allows to bridge the technological gaps as easily as possible.

Diagram viewers can be connected to KIML by implementing the interface `IDiagramLayoutManager` and registering the implementing class to the extension point `de.cau.cs.kieler.kiml.service.layoutManagers`. Such a class, called a *layout manager*, is responsible for extracting a graph structure out of any instance of the diagram viewer and for applying a computed layout back to the viewer instance. The graph structure extraction is done with the interface method

```
LayoutMapping<T> buildLayoutGraph(IWorkbenchPart workbenchPart,  
    Object diagramPart);
```

which creates an instance of the `KGraph` meta model based on the given workbench part and the selected element contained in the workbench part (`diagramPart` argument). The `KGraph` instance, called the *layout graph*, is stored in the returned instance of `LayoutMapping` together with additional information required by the layout manager, e. g. a mapping of diagram viewer elements to layout graph elements. After the layout algorithms have been executed on the layout graph, the resulting concrete layout is applied to the diagram viewer with the method

```
void applyLayout(LayoutMapping<T> mapping, boolean zoomToFit,  
    int animationTime);
```


4.3. Connection to Diagram Viewers

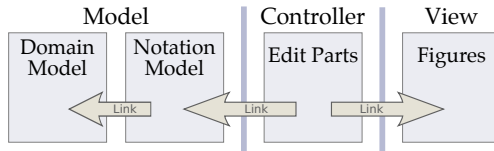


Figure 4.9. Basic components of GMF implementing the model-view-controller (MVC) pattern.

The combination of layout managers and layout algorithms is organized in the class `DiagramLayoutEngine`, which is the main programmatic entry point for the layout of diagram viewers.

The connection of KIML to the two most important diagramming frameworks, GMF and Graphiti, is discussed in the following.

4.3.1 GMF Editors

The Graphical Modeling Framework (GMF) is employed by many Eclipse-based modeling applications, including IBM Rational Software Architect³ and Papyrus [LTE⁺09]. It uses the MVC implementation of the Graphical Editing Framework (GEF), where *edit parts* represent the controller and *figures* represent the view [RWC11]. GMF complements these two components by using domain models defined with EMF and an additional *Notation* model. The four components are connected as shown in Figure 4.9: edit parts have links to figures and elements of the Notation model, and the latter have links to elements of the domain model.

The Notation model determines which subset of the domain model is visible in the viewer and adds information on the concrete representation such as positions of diagram elements, font names, and other style data. Edit parts are responsible for all user interaction with the diagram and thus determine the behavior of the editor. Furthermore, they determine the graphical representation of the diagram elements by creating and combining figures, which are the building blocks of the view component.

³ <http://www.ibm.com/software/products/en/ratisoftarch/>

4. Integration in Eclipse

Table 4.1. Mapping of GMF edit part classes to the corresponding KGraph classes created for the layout graph.

Edit Part Class		KGraph Class
ShapeNodeEditPart	↦	KNode
ConnectionEditPart	↦	KEdge
AbstractBorderItemEditPart	↦	KPort
LabelEditPart	↦	KLabel

Each kind of GMF-based diagram editor or viewer must implement a dedicated set of edit parts. Since the implementation of edit parts requires a considerable amount of work, GMF offers a subproject named *GMF Tooling* for the automatic generation of Java code out of a set of specification models. This may potentially shorten development times by focusing work on abstract specifications instead of low-level code.

The connection of KIML to GMF is based on edit parts. Each element of the layout graph has one corresponding edit part from which all information that is necessary for the layout process can be extracted. `GmfDiagramLayoutManager`, which implements the `IDiagramLayoutManager` interface generically for GMF, realizes `buildLayoutGraph(...)` by analyzing the structure and the characteristics of the edit parts of a diagram viewer. The type of graph element can be derived from the superclass of an edit part class (see Table 4.1). The method `applyLayout(...)` is realized by writing the computed concrete layout into the Notation model. GMF then automatically updates the positions of the figures displayed in the diagram viewer.

In the following, I present several examples of GMF-based editors where automatic layout through KIML has been employed.

SyncCharts. In its first years (2009–2013), KIELER included a number of GMF-based editors for the evaluation of the general concepts developed in the project. Among these is an editor for *SyncCharts*, a statecharts dialect with synchronous semantics [And03]. The editor has been used for simulation and code generation [MFvH10, TAvH11] as well as for experimenting with focus & context implementations [FvH10b]. For all these activities,

4.3. Connection to Diagram Viewers

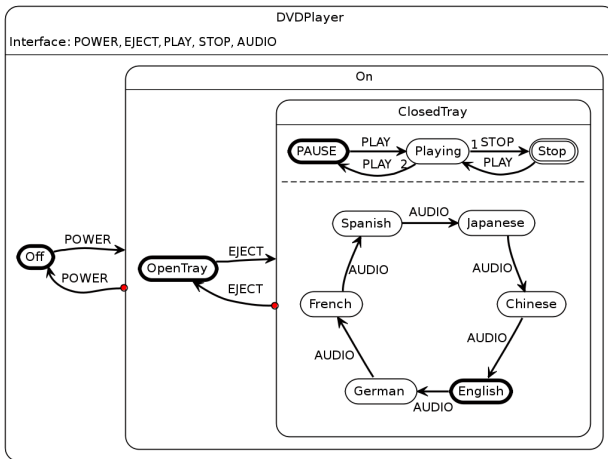


Figure 4.10. A diagram drawn by the KIELER SyncCharts editor with automatic layout provided by KIML.

automatic layout has been provided by KIML using the generic layout manager for GMF. An example is shown in Figure 4.10.

KAOM. KAOM is another KIELER editor that is based on GMF and can be processed with the generic layout manager. Its main purpose is to visualize data flow models from different languages, e.g. Ptolemy [EJL⁺03]. The KAOM meta model is inspired by the MoML format used by Ptolemy [BLL⁺08, Chapter 1]. The same meta model has also been used for representing ASCET models in an industrial application [FvHK⁺14]. Figure 4.11 shows a diagram imported from the Ptolemy demonstration models.

Ecore diagrams. Metamodels of the Eclipse Modeling Framework (EMF) are declared using a *meta-metamodel* named *Ecore*, which is itself defined with the Ecore format. Among other editing interfaces, EMF offers a GMF-based editor for Ecore models. For instance, the KGraph and KLayoutData metamodels depicted in Figure 3.2 (p. 107) have been drawn with that editor. Ecore diagrams can be processed with the generic layout manager,

4. Integration in Eclipse

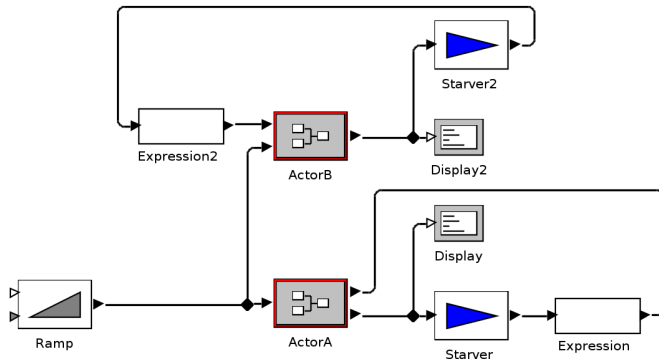


Figure 4.11. A data flow diagram drawn by the KAOM editor with automatic layout provided by KIML.

but required a special adaptation due to the concept of *opposite references*. If a class *A* references another class *B* and *B* references *A*, these two references can be declared as opposite, meaning that any change in the value of one reference is automatically transferred to the other one. In Ecore diagrams, opposite references are represented by two edges which are kept synchronized. The generic layout manager was adapted such that it creates only one edge in the layout graph for each pair of opposite references.

Papyrus. *Papyrus* is an Eclipse project that targets a UML modeling environment following the OMG specification [LTE⁺09]. Most of its graphical editors can be processed with the generic GMF layout manager, which has been used for implementing a *structure-based editing* approach [FSMvH10]. An activity diagram arranged with the Graphviz Dot algorithm can be seen in Figure 4.12. However, Papyrus has a concept of holding multiple editors in the same editor part and switching between them using tabs. At any time, a Papyrus editor part has exactly one *active* editor. This adaptation must be considered in a small extension of the generic layout manager by delegating to the active editor of a Papyrus editor part. Furthermore, the editor for sequence diagrams cannot be processed with graph layout algorithms, but it requires a dedicated layout algorithm as well as further

4.3. Connection to Diagram Viewers

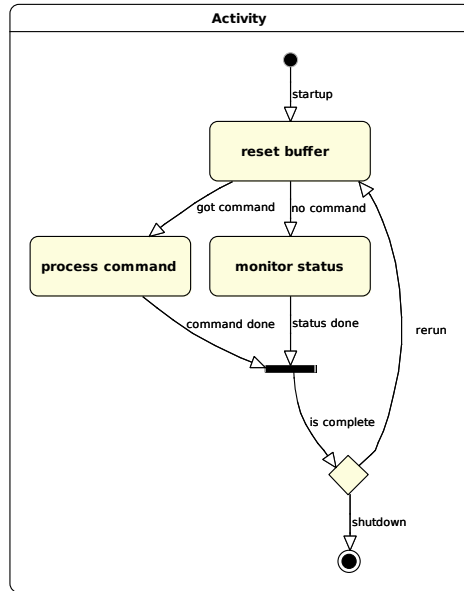


Figure 4.12. An activity diagram drawn by the Papyrus modeling environment with automatic layout provided by KIML [FSMvH10].

extensions of the layout manager. Hoops developed such an algorithm and layout manager extension in his diploma thesis [Hoo13].

Yakindu. *Yakindu Statechart Tools* (SCT) is a free tool for modeling statecharts.⁴ Figure 4.13 shows an example that is shipped together with the tool. The depicted diagram has been arranged using Graphviz Dot and the generic GMF layout manager. Yakindu SCT is adaptable to cover a wide range of syntax and semantics variants for statecharts. For instance, Haribi recently covered the adaptation to SyncCharts in his Master's thesis [Har13].

⁴ <http://www.statecharts.org/>

4. Integration in Eclipse

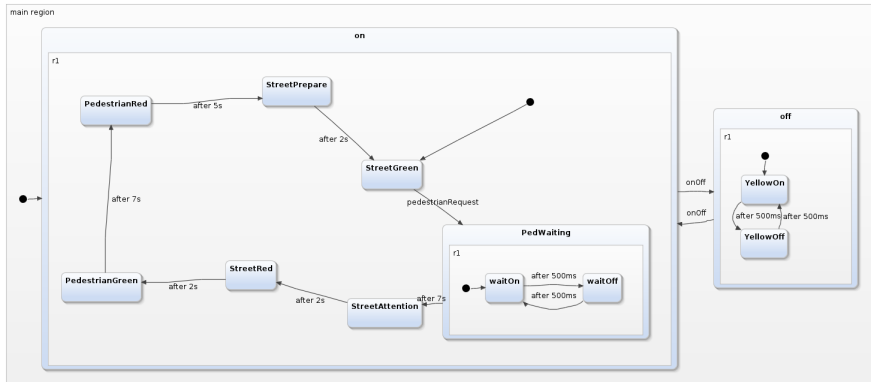


Figure 4.13. A statechart drawn by Yakindu SCT with automatic layout provided by KIML.

Model transformation. Model transformation is an integral concept of model-driven engineering. In order to view the result of a transformation using a graphical syntax, the generated diagram must be processed with automatic layout. Van Gorp and Rose have used GMF for specifying the graphical syntax of Petri nets and statecharts and used KIML to create readable drawings of transformation results [vGR13]. The result of one of their test cases is shown in Figure 4.14.

4.3.2 Graphiti Editors

The Graphiti framework uses GEF for user interaction and displaying diagrams, but unlike GMF, it does not require to implement any edit parts or figures for building a graphical editor. Graphiti offers a more abstract interface based on an EMF meta model named *Pictogram model* (see Figure 4.15). The Pictogram model contains *Pictogram elements*, specifying the structure of elements in the diagram, and *graphics algorithms*, specifying the concrete rendering of each element. With these two components, an instance of the Pictogram model contains complete information for drawing a diagram, but it says nothing about its behavior. Each kind of user interaction has to be

4.3. Connection to Diagram Viewers

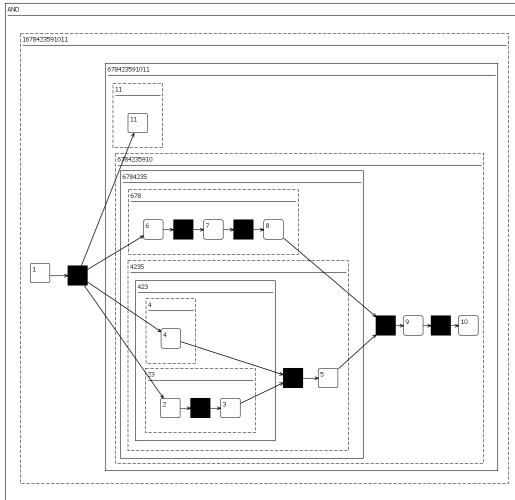


Figure 4.14. A statechart generated through model transformation from a Petri net [vGR13].

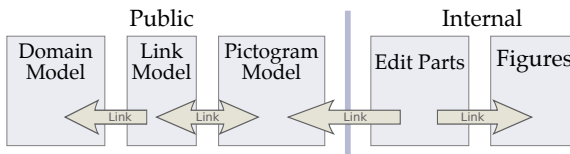


Figure 4.15. The public interface of a Graphiti diagram consists of the Pictogram model, which is linked to a domain model with the Link model. Edit parts and figures are implemented in the internal rendering engine.

implemented in Java with a so-called *feature*, e. g. creating a domain model element, adding a visual representation for the element to the Pictogram model, or arranging the content of a Pictogram element. The link between the Pictogram model and the domain model is realized with an additional meta model named *Link model*.

A generic diagram layout manager for Graphiti has been implemented in KIML with the class `GraphitiDiagramLayoutManager`. This implementation

4. Integration in Eclipse

Table 4.2. Mapping of Pictogram Model classes to the corresponding KGraph classes created for the layout graph.

Pictogram Model Class		KGraph Class
Shape	↦	KNode
Connection	↦	KEdge
Anchor	↦	KPort
ConnectionDecorator	↦	KLabel

uses the Pictogram model to extract a layout graph from a diagram. The mapping of Pictogram elements to KGraph elements is listed in Table 4.2. However, this mapping is not sufficient for extracting the layout graph correctly from all kinds of Graphiti-based diagrams, since the class Shape of the Pictogram model is not clear enough regarding the graph structure. Given an instance of Shape, it cannot be generically decided whether it represents a node, or merely a decorative element that should be ignored in the graph layout process. Therefore, the GraphitiDiagramLayoutManager is designed such that it can easily be extended to detect more precisely the graph structure of particular Graphiti-based diagrams.

eTrice. The Eclipse project *eTrice* implements the Real-Time Object-Oriented Modeling (ROOM) language [SGW94] using Xtext for textual modeling and Graphiti for graphical modeling. The tool comprises two graphical notations, namely structure diagrams and behavior diagrams. In 2012, a Google Summer of Code project was done with the goal of integrating KIML with eTrice.⁵ The integration was realized with a subclass of the generic layout manager for Graphiti, specializing the detection of the graph structure for the two types of diagrams. Figure 4.16 shows a behavior diagram arranged with KIML. The integration is now made part of the official release of eTrice.⁶

⁵http://wiki.eclipse.org/ETrice/GSoC/2012/DiagramLayout_Kieler

⁶<http://www.eclipse.org/etrice/>

4.3. Connection to Diagram Viewers

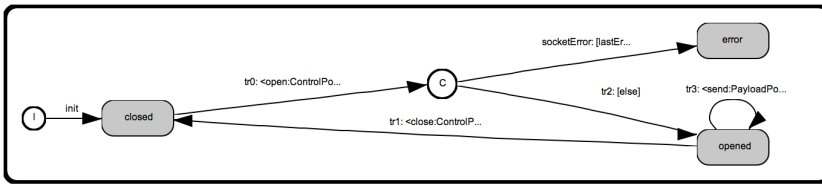


Figure 4.16. A behavior diagram drawn by eTrice with automatic layout provided by KIML.

4.3.3 Transient Views

GMF and Graphiti both target the *editing* of graphical models. However, graphical views are often generated from a non-graphical source such as a text editor. In these situations, editing of the graphical views is typically not necessary, but the speed and scalability of the view generation as well as the quality of graph layouts are crucial. Measurements of the execution time and memory consumption for generating views with GMF have shown that the framework does not satisfy the scalability demands, especially in industrial applications [FvHK⁺14]. Furthermore, years of experience working with GMF have led to the impression that, in spite of the model-based editor generation capabilities of GMF Tooling, the process of realizing precise requirements of the concrete syntax is time-consuming and error-prone. Graphiti improves on that deficiency by offering a meta model for graphical notations, but that meta model lacks in any means for specifying the arrangement of low-level drawing components such as lines and rectangles. Furthermore, as mentioned in Section 4.3.2, the Pictogram model does not specify precisely which elements to regard as nodes of the graph structure.

The KIELER Lightweight Diagrams (KLighD) project targets a scalable implementation of an approach named *transient views* [SSvH12a, SSvH12b, SSvH13], where graphical views are created full-automatically on demand and discarded after use. The implementation follows the idea of *model-driven visualization* as proposed by Bull et al. [BSLF06], which is an extension of the MDE approach to the creation of views. The central concept of KLighD is to use the same meta model for the view model and for the layout graph passed

4. Integration in Eclipse

to layout algorithms, namely the KGraph model (Section 3.1.1). This allows to perform automatic layout very efficiently [FvHK⁺14], since the extraction of the layout graph from the diagram is trivial. In fact, the diagram layout manager for KLighD merely creates a copy of the view model when a layout is requested. In order to ensure scalability, infrastructure for drag-and-drop editing is omitted, and the diagram rendering is delegated to the very fast graphics framework *Piccolo2D* [BGM04]. The specification of graphical elements of a KLighD diagram is done with an extension of the KGraph model named *KRendering* [SSvH12a].

Configuration with immediate feedback. In a direct comparison with GMF, an average speedup of 7.4 was achieved with KLighD for the application of computed graph layouts to the diagram [FvHK⁺14]. This improvement in performance paves the way for new interaction techniques with the diagram viewer. In particular, manipulation of the abstract layout can be done with immediate feedback. This is realized with a sidebar of KLighD views, shown in Figure 4.17, offering buttons and sliders for modifying values of layout options [SSM⁺13]. Whenever a button is pushed or a slider is dragged, the layout algorithm is immediately executed with the updated configuration and the result is visualized in the view. For diagrams that are not too large (up to several hundreds of nodes on normal computers) the computation of a new layout and its application to the view is fluid enough to respond seamlessly to the user's actions. In contrast to the Layout View (Section 4.2.1), the configuration in the sidebar is not persistent, but is discarded when the view is closed.

Ptolemy. All drawings of Ptolemy diagrams [EJL⁺03] shown in Section 2.2.4 have been created with KLighD. This is realized with a transformation of the MoML format used by Ptolemy into the KGraph format. The transformation is written in the Xtend language⁷, which allows a more compact representation compared to Java.

⁷ <http://www.eclipse.org/xtend/>

4.3. Connection to Diagram Viewers

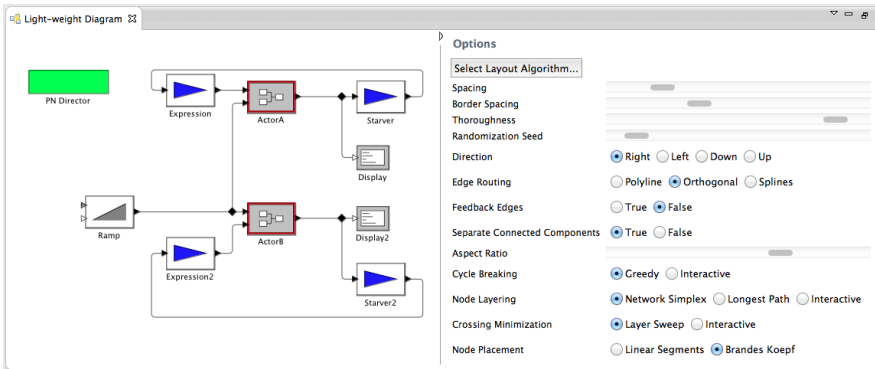


Figure 4.17. A KLighD view of a Ptolemy model (left) with a sidebar for modifying values of layout options (right).

ASCET. A similar visualization has been made for the ASCET language in the context of software development for electronic control units (ECUs) in the automotive domain [FvHK⁺14]. The usual procedure of creating ASCET models is to split them into hierarchically nested diagrams that can each be drawn on a single sheet of paper. Engineers who need to obtain an understanding of the data flow in a model sometimes have no other choice than printing the diagrams, glueing together the sheets, and drawing the missing connections between the diagrams by hand. This situation can be improved by automatically generating views where the different diagrams of a model are directly connected with each other, as seen in Figure 4.18. The views are generated using KLighD combined with the *KLay Layered* layout algorithm discussed in Section 4.4.1, which is optimized for the requirements of data flow diagrams.

Class diagrams. The class diagrams shown in Figures 4.2, 4.3, and 4.4 have been drawn with KLighD [SSSvH14]. The visualization is based on an Xtend transformation developed by Schwanke [Sch14]. The source model of that transformation is provided by the Java Development Tools (JDT), the standard Java IDE for Eclipse.

4. Integration in Eclipse

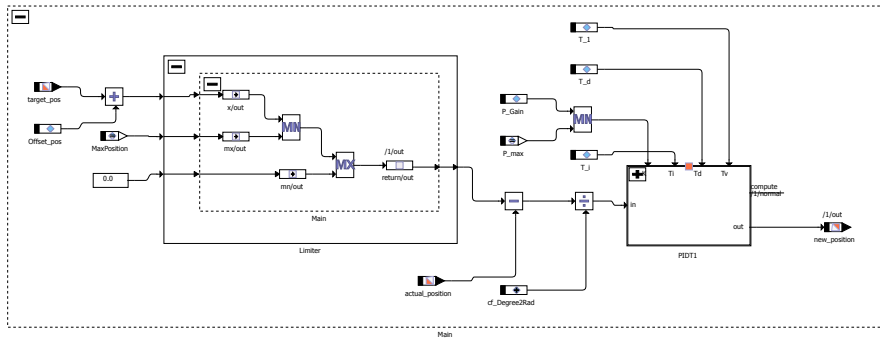


Figure 4.18. A focus & context view of an ASCET model drawn with KLightD.

SCCharts. The SyncCharts language mentioned in Section 4.3.1 provides a synchronous model of computation for statecharts [And03]. A problem with this approach is that it requires a unique state of each variable within one logical time unit, hence it rejects statements such as $\text{if } (x < 0) \ x = 0$, which are natural in sequential languages. The *sequentially constructive* model of computation overcomes this limitation by extending the synchronous semantics, which leads to the *SCCharts* language [vHDM⁺14]. Figure 4.19(a) shows an example that includes a hierarchical state (WaitAB) with two parallel regions. In order to compile such an SCChart, it is transformed into an *SC Graph* representation (SCG) that specifies the control flow, as seen in Figure 4.19(b). From this representation, further compilation steps can be applied to generate either hardware or software. Both the SCChart and the SCG representations are implemented in the KIELER project using KLightD.

4.4 Connection to Algorithms

Many graph layout algorithms are not able to handle hierarchical graphs, in which nodes may contain nested subgraphs. Such algorithms can be generically extended to process hierarchical graphs with the recursive scheme sketched in Algorithm 3.1 (p. 112). That scheme is implemented in

4.4. Connection to Algorithms

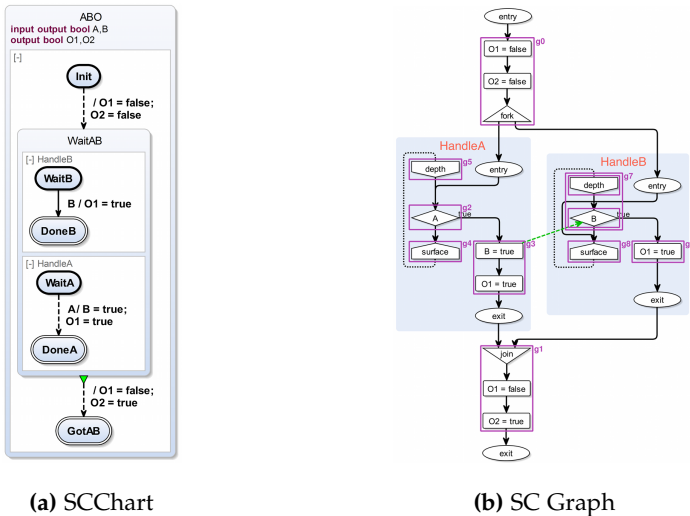


Figure 4.19. An SCChart example (statechart) with its automatically generated SC Graph (control flow graph), both drawn with KLightD [vHDM⁺14].

KIML with the class `RecursiveGraphLayoutEngine`. On each hierarchy level, it fetches an instance of the layout algorithm that was selected for that level and executes it. A layout algorithm is connected to KIML with a subclass of `AbstractLayoutProvider`, which declares a method

```
public abstract void doLayout(KNode parentNode,
    IKielerProgressMonitor progressMonitor)
```

When that method is invoked, the respective algorithm is expected to process the subgraph determined by the set of nodes accessed with `getChildren()` on the argument `parentNode`. The `progressMonitor` is used to track the progress of the algorithm during its execution, e. g. for displaying progress feedback to the user.

Implementations of `AbstractLayoutProvider` are registered in the extension point `de.cau.cs.kieler.kiml.layoutProviders` together with meta data on the algorithms (see Section 4.1.1). A subclass of `AbstractLayoutProvider` may provide multiple layout algorithms, of which one is selected through

4. Integration in Eclipse

a parameter given to the extension point. This is useful for connecting a whole library of layout algorithms with a single layout provider class.

In the following, we discuss some layout algorithms implemented in KIELER and the connection of two non-Java algorithm libraries.

4.4.1 The KIELER Layouters (KLayout)

KLayout is a Java library of layout algorithms. It has been developed with the same goals stated for KIML in Section 4.1, namely efficiency, simplicity, and flexibility. The algorithms are targeted towards graphs used in MDE, where in most cases less than a hundred nodes are visualized in one diagram, even if a model may contain thousands of nodes in total (cf. Section 2.1.2 in [Kla12]). The layout of huge graphs with many thousands of nodes is not targeted by KLayout. Therefore some compromises are possible with respect to efficiency: where design decisions had to be made either in favor of running time or in favor of simplicity and flexibility, the latter option was chosen. Measurements such as those done for the KLayout Layered algorithm, presented in Section 2.2.4, demonstrate that the running time is low enough for typical MDE graphs.

Each layout algorithm implementation defines its own data structure for representing graphs. A major contribution to both simplicity and flexibility is done by implementing the `IPropertyHolder` interface in all classes of graph elements in these data structures (see Section 4.1.1). This allows to attach arbitrary information to each graph element instance in a consistent and convenient manner. Most importantly, it enables the extension of layout algorithms without the need of modifying the data structures. For instance, if an algorithm is extended with a preprocessing step *A* and a postprocessing step *B*, and some information on certain graph elements needs to be passed from step *A* to step *B*, that information can simply be added to the graph elements with the method `setProperty(...)`. Later the information can be retrieved with `getProperty(...)`. The information is identified with an instance of `IProperty` declared as a constant, e. g.

```
public static final IProperty<Boolean> REVERSED
    = new Property<Boolean>("reversed", false);
```

4.4. Connection to Algorithms

The first argument to the Property constructor is its identifier, and the second argument is its default value. The particular property shown above is used in the layer-based algorithm to mark edges that have been reversed in the cycle elimination step.

Many approaches to graph layout are structured into multiple phases that are each dedicated to a specific optimization problem. The layer-based approach, for instance, uses five phases (see Section 2.1). For each of these phases there are multiple alternative algorithms, either using different heuristics for the optimization problems or arranging the graph with different styles (e. g. straight-line or orthogonal edge routing). Alternative algorithms for one phase can be exchanged according to the well-known *strategy* pattern. Which strategy to choose is decided based on layout options.

A problem of having multiple different implementations for the same problem is that whenever one implementation is extended to support further features such as ports or labels, the same kind of extension must be done for all other implementations. Furthermore, adding certain extensions can lead to lengthy code that is hard to maintain. For instance, the main edge direction would have to be considered in the node placement phase and the edge routing phase of the layer-based approach. The four options (up, down, left, right) would require special attention at all points in the code where concrete coordinates are processed. These problems can be avoided with the concept of *intermediate processors* [SSvH14], which was first proposed by Schulze [Sch11]. An intermediate processor for a layout algorithm with k phases can be assigned to one of $k + 1$ possible *slots*, as shown in Figure 4.20 for the layer-based algorithm. There is one slot before the first phase, one after the last phase, and one between each pair of consecutive phases. For each $i \in \{1, \dots, k\}$, the selected implementation of phase i is executed after the intermediate processors of slot i , but before those of slot $i + 1$. Which of the available intermediate processors are actually executed for an input graph depends on the properties of the graph and on its abstract layout. For instance, the layer-based algorithm is implemented such that the main edge direction is left-to-right. If the layout option for selecting the direction is set to anything different than this default, coordinate transformations are performed with intermediate processors in the first slot and in the last

4. Integration in Eclipse

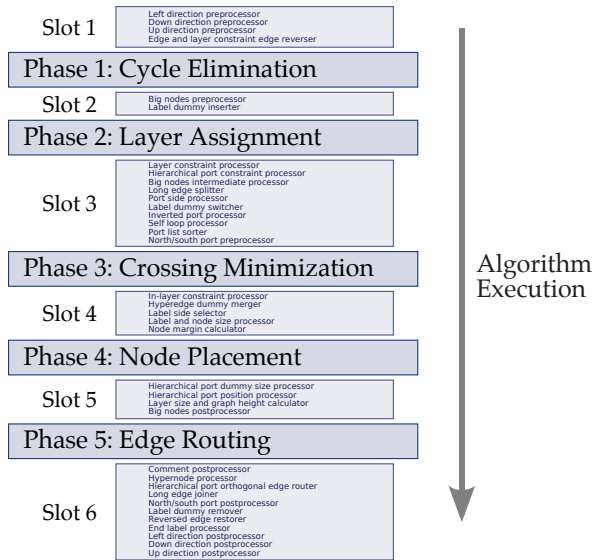


Figure 4.20. Main phases of the layer-based layout algorithm and slots for intermediate processors between these phases.

slot, e. g. transposing the x and y coordinates for the top-down direction. This preprocessing and postprocessing requires additional iterations over all graph elements, but allows simpler and more maintainable implementations of the main phases through a clear separation of concerns.

The approach of creating dummy nodes in order to extend algorithms towards new features, as done for ports with prescribed sides in Section 2.2.3, fits perfectly into the concept of intermediate processors: a preprocessor creates the dummy nodes, and a postprocessor removes them and adapts the final layout. This is contrary to the approach of Eiglsperger et al. [ESK05], who proposed to avoid the creation of dummy nodes for better execution time and memory performance. Years of experience with the implementation, maintenance and extension of KLayout Layered (see below) have shown that the flexibility and simplicity gained with the intermediate processors approach outweighs the performance overhead of creating dummy nodes

4.4. Connection to Algorithms

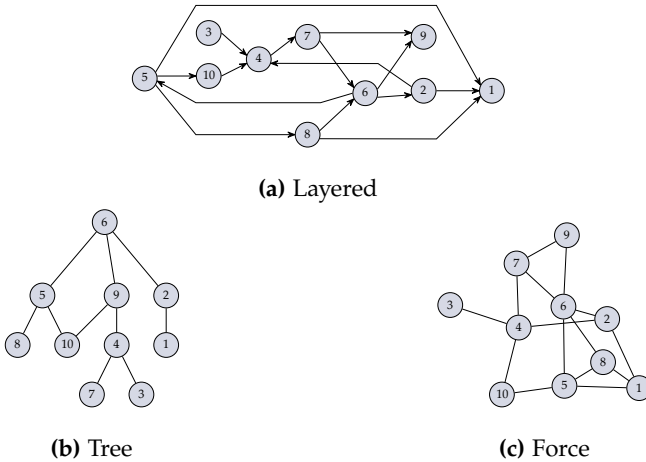


Figure 4.21. Graphs drawn with the KIELER Layouters (K Lay).

and adding more iterations over the graph.

The K Lay project currently contains four layout algorithms, which are briefly described in the following. Some examples are shown in Figure 4.21.

K Lay Layered – an implementation of the layer-based approach discussed in Chapter 2. This algorithm has enjoyed the highest attention and effort of all K Lay algorithms. It has been worked on by several students [Döh10, Sch11, Fuh12, Car12] and has been a basis for innovations [KSSvH12, SSvH14]. In particular, the concept of intermediate processors was first implemented in K Lay layered. The number of Java classes implementing intermediate processors varies over time due to continuous advancement of the algorithm; 33 classes were present at the time of writing.

K Lay Orthogonal – an implementation of the topology-shape-metrics approach [TDB88] realized in two student theses [Cla10, Klo12]. The algorithm has three phases and 15 intermediate processors.

K Lay Tree – a special algorithm for drawing trees realized during a practical course at CAU in summer semester 2013. The algorithm has four phases and six intermediate processors.

4. Integration in Eclipse

KLay Force – an implementation of the force-based approach [Ead84, FR91] realized during a practical course at CAU in summer semester 2010. Force-based algorithms cannot be divided into independent phases, hence the concept of intermediate processors could not be applied to this implementation.

These regular layout algorithms are complemented by three utility algorithms: *Box Layout* packs nodes into rows without considering edges, which is useful for parallel regions of composite states in statecharts. *Fixed Layout* keeps the layout as it is, allowing to disable automatic layout in selected parts of a diagram. *Randomizer* assigns random positions to graph elements, producing unreadable layouts that can be used as starting conditions for user studies.

4.4.2 Graphviz

The popular tool Graphviz can be used either as a C library or on the command line [GN00]. The main exchange format is DOT,⁸ a simple language for specifying graphs together with their appearance, layout options, and concrete layout. The connection to KIML was realized by executing Graphviz in a separate OS process and communicating with that process via standard input and output. The DOT format is implemented with an Xtext grammar, from which the Xtext code generator derives an EMF meta model for constructing instances of DOT graphs. The following steps are performed whenever a layout is requested.

1. Transform the input KGraph G_K to an instance G_I of the DOT meta model generated by Xtext.
2. Serialize the model G_I .
3. Send the resulting text to the Graphviz process.
4. Wait until the process delivers a response and read it.

⁸<http://www.graphviz.org/>

4.4. Connection to Algorithms

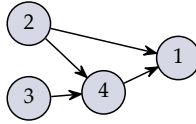


Figure 4.22. A graph drawn with the Dot algorithm provided by Graphviz.

Listing 4.4. The DOT graph generated with Xtext for the example shown in Figure 4.22. The edge comments are used as identifiers to correctly associate the edges found in the output graph (Listing 4.5).

```
digraph {  
  graph [nodesep="0.27778", ranksep="0.27778", rankdir=LR];  
  node [shape=box, fixedsize=true];  
  edge [dir=none];  
  node1 [width="0.41667", height="0.41667"];  
  node2 [width="0.41667", height="0.41667"];  
  node3 [width="0.41667", height="0.41667"];  
  node4 [width="0.41667", height="0.41667"];  
  node2 -> node4 [comment="edge1"];  
  node2 -> node1 [comment="edge2"];  
  node3 -> node4 [comment="edge3"];  
  node4 -> node1 [comment="edge4"];  
}
```

5. Parse the response, resulting in another instance G_O of the DOT meta model (corresponding to the parser's abstract syntax tree).
6. Transfer the concrete layout encoded in G_O to G_K .

Step 4 corresponds to a blocking read of the process output, which is represented with an `InputStream` instance in Java. That interface does not provide any mechanism to abort the blocking state after a timeout, hence step 4 can never terminate when the process gives no answer. We compensate for this flaw by running an additional Java thread that closes the process pipe when the read operation takes too long.

An example arranged with Graphviz is illustrated in Figure 4.22. The DOT graph G_I sent to the process is in Listing 4.4, and the corresponding response G_O is in Listing 4.5.

4. Integration in Eclipse

Listing 4.5. The DOT graph sent by Graphviz after receiving the text in Listing 4.4.

```
digraph {
  graph [bb="0,80,130,0", nodesep=0.27778, rankdir=LR, ranksep=0.27778];
  node [fixedsize=true, label="\N", shape=box];
  edge [dir=none];
  node1 [height=0.41667, pos="115,38", width=0.41667];
  node2 [height=0.41667, pos="15,15", width=0.41667];
  node2 -> node1 [comment=edge2,
    pos="30.041,18.295 48.671,22.668 81.368,30.342 99.985,34.711"];
  node4 [height=0.41667, pos="65,61", width=0.41667];
  node2 -> node4 [comment=edge1,
    pos="30.142,28.553 36.336,34.489 43.591,41.442 49.793,47.385"];
  node3 [height=0.41667, pos="15,65", width=0.41667];
  node3 -> node4 [comment=edge3,
    pos="30.142,63.822 36.336,63.305 43.591,62.701 49.793,62.184"];
  node4 -> node1 [comment=edge4,
    pos="80.142,54.224 86.336,51.255 93.591,47.779 99.793,44.807"];
}
```

Five layout algorithms are provided to KIML through the connection to Graphviz: layer-based [GKNV93], energy-based [GKN05], force-based [FR91], radial [Wil97], and circular [ST99]. The DOT format does not permit to select the layout algorithm through its key-value interface, but that selection must be done with a command line argument. Therefore one separate process must be created for each of the provided algorithms.

4.4.3 OGDF

The Open Graph Drawing Framework (OGDF) [CGJ⁺13] is implemented purely in C++ and provides its layout algorithms through subclasses of the abstract class `LayoutModule`.⁹ Furthermore, it offers functions for importing graphs from formats such as GML [Him97] or OGML, an XML-based format developed in a students project at TU Dortmund.¹⁰ The connection of OGDF to KIML was done similarly to the connection of Graphviz (Section 4.4.2). However, since OGDF does not provide a command line interface, the library was wrapped into an executable program that reads

⁹<http://www.ogdf.net/>

¹⁰<http://ls11-www.cs.tu-dortmund.de/mutzel/>

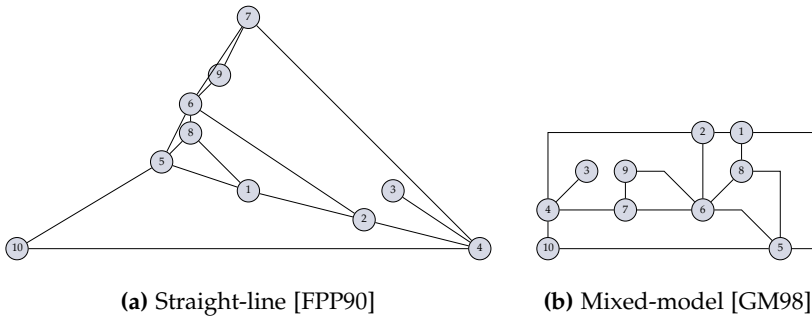


Figure 4.23. A graph drawn with two algorithms of the Open Graph Drawing Framework (OGDF).

graphs through its standard input, invokes layout algorithms, and writes the resulting coordinates to its standard output. The input to that program is transferred in the OGML format, while the output is a simple list of coordinates for nodes and edges.

With the current connection to KIML, 22 layout algorithms are provided. Some of these are quite special and are found rarely in other graph layout libraries, in particular the algorithms based on planarity. While some algorithms are regarded as experimental and are not of high practical value, such as the planar straight-line algorithm of De Fraysseix et al. [FPP90] of which a drawing can be seen in Figure 4.23(a), others yield very useful drawings. Figure 4.23(b) shows a drawing of the mixed-model algorithm by Gutwenger and Mutzel [GM98]. Other valuable implementations include orthogonal layout with extensions for UML class diagrams [GJK⁺03], upward planarization [CGMW11], and multilevel layouts [BGKM11]. With these capabilities, OGDF is a good supplement to the layouts provided by Graphviz and KIELER.

4.5 Tools for Algorithm Developers

The efficient development of graph layout algorithms requires not only a good IDE such as the Java Development Tools (JDT) shipped with Eclipse,

4. Integration in Eclipse

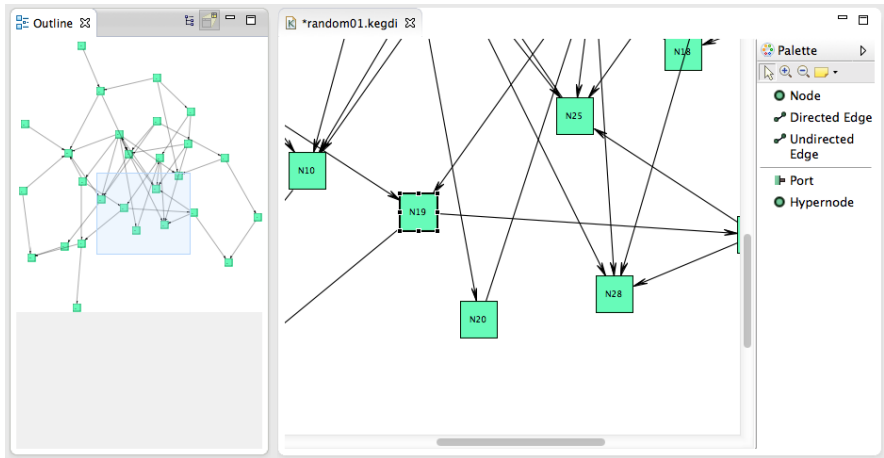


Figure 4.24. A KEG editor window (right) with an outline view of the whole graph (left). The palette on the right of the editor window can be used to create new graph elements.

but also convenient tools for the creation of test graphs and for the analysis of the algorithm output. Such tools are presented in the following. They have been used for the development of KLayout algorithms as well as the layout infrastructure of KIML.

4.5.1 Graph Editor

The KIELER Editor for Graphs (KEG) was initially developed by Rieß [Rie10]. It allows to create graphs with a drag-and-drop interface, shown in Figure 4.24. Graph elements can be selected in the *palette* and added to the graph by clicking onto the canvas. The concrete layout can be manipulated directly by dragging elements over the canvas. Since the editor is based on GMF, layout algorithms provided through KIML can be applied to it with the generic connection described in Section 4.3.1 and with no further adaptations.

The editor code has been generated using GMF Tooling, yielding the

4.5. Tools for Algorithm Developers

benefit of a relatively short development time. The initial idea was to use the KGraph meta model (Section 3.1.1) as domain model of the generated editor. However, limitations of GMF Tooling enforced the usage of another meta model, which was chosen to inherit from the classes of KGraph. The extended meta model includes undirected edges, which inherit from KEdge but are drawn without any arrowhead, and hypernodes, i. e. nodes used for representing hyperedges (see Section 2.3.2).

The main advantage of KEG is its intuitive user interface. Disadvantages are the limited capability to adapt the rendering of graph elements, and the waste of time caused by flaws of GMF. For instance, an edge can be connected to a port only if the user drags the mouse exactly over the graphical representation of the port, which means trying to hit an area of 8×8 pixels. Especially on high-resolution screens this can be quite frustrating, often forcing the user to change the zoom level just to connect an edge.

4.5.2 Textual Format

Textual modeling is sometimes more attractive compared to graphical modeling, since it allows more precise control over model elements: all details of an element can be specified textually in one place, while graphical editors often rely on the *Properties* view to edit the details, forcing users to switch frequently between two different views on the model. Other advantages of textual formats are the simpler merging of models with a version control system and the simpler realization of copy-and-paste operations. For these reasons, the KEG editor has been complemented with a textual graph editor based on the KGraph abstract syntax. As illustrated in Figure 4.25, the editor is linked with a KLighD view that automatically visualizes the graph modeled in the text. Any change in the text is immediately transferred to the KLighD view.

The text editor has been generated with the Xtext framework. Listing 4.6 shows an excerpt of the grammar for the KGraph format, i. e. its concrete syntax. An example graph in textual form is shown in Listing 4.7. The grammar follows a simple basic concept: graph elements are written in the form `<type> <identifier> { <content> }`, e.g. `knode node1 {...}`. All fur-

4. Integration in Eclipse

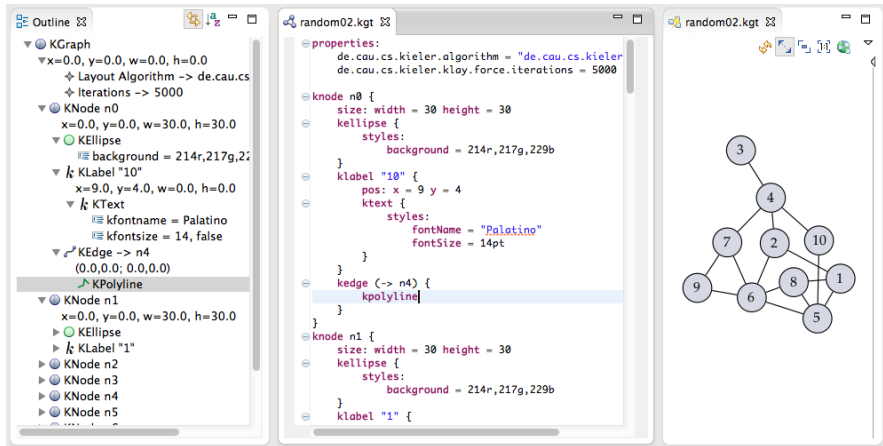


Figure 4.25. A text editor for the KGraph format (center) with a tree-based structural outline (left) and a synchronized KLighD view (right) visualizing the content of the editor.

ther details are written as key-value pairs in the form `<section>: (<key> = <value>)*`, e. g. `size: width = 40 height = 30`. The details also include layout options, which are given in a section named `properties`. Furthermore, the format supports the specification of the concrete rendering of graph elements according to the KRendering abstract syntax used in KLighD [SSvH12a]. The syntax of rendering elements follows the concept mentioned above for graph elements. Many of the images shown in this thesis have been created with this format, e. g. Figure 4.23.

Random graphs. Testing and evaluating layout algorithms is greatly simplified if graphs can be generated automatically. Therefore both the KEG editor and the textual KGraph editor offer a generator for random graphs. The generator can be controlled with numerous parameters such as the number of graphs, the number of nodes, and the number of edges. These parameters are presented in a *wizard*, i. e. a window with a series of dialogs for setting parameter values, of which one dialog is shown in Figure 4.26.

4.5. Tools for Algorithm Developers

Listing 4.6. An excerpt of the Xtext grammar for KGraph, showing syntax definitions for nodes and their layout data.

```
KNode returns KNode:
  {KNode}
  'knode' (data+=KIdentifier)? (('{'
    data+=KNodeLayout
    (labels+=KLabel | children+=KNode | ports+=KPort
     | outgoingEdges+=KEdge | data+=KRendering
     | data+=KRenderingLibrary)*
  '}') | data+=EmptyKNodeLayout);

KNodeLayout returns KShapeLayout:
  (('pos' ':' (('x' '=' xpos=Float)? & ('y' '=' ypos=Float)?))?
  & ('size' ':' (('width' '=' width=Float)?
   & ('height' '=' height=Float)?))?
  & ('properties' ':' (persistentEntries+=Property)*)?
  (('insets' ':' insets=KInsets) | insets=EmptyKInsets);

Property returns PersistentEntry:
  key=QualifiedID '=' value=PropertyValue;
```

Listing 4.7. A KGraph instance written in textual form according to the concrete syntax specified in Listing 4.6.

```
knode node1 {
  knode node2 {
    size:
      width = 40
      height = 30
    properties:
      portConstraints = FIXED_SIDE
      nodeLabelPlacement = "H_CENTER V_CENTER INSIDE"
    klabel "Node 2"
    kport p2 { properties: portSide = NORTH }
  }
  kport p1
  kedge (:p1 -> node2:p2)
}
```

4. Integration in Eclipse

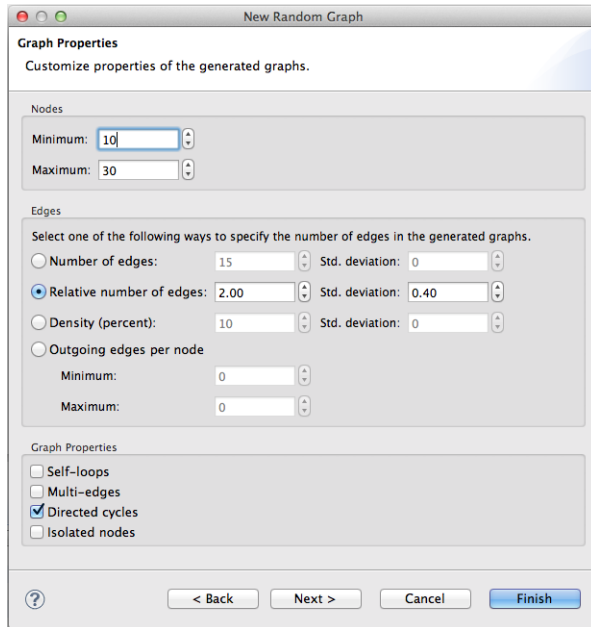


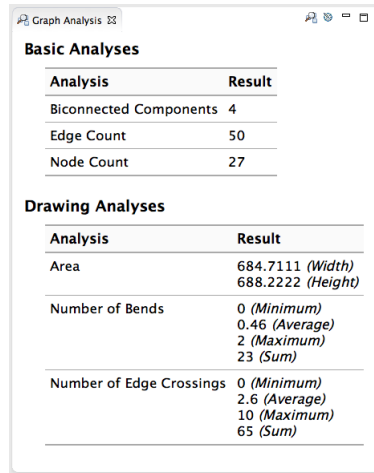
Figure 4.26. A dialog of the random graph generation wizard, offering to set the number of nodes, the number of edges, and other general properties.

The number of edges can be specified as an absolute value, as a factor multiplied by the number of nodes, as a density value depending on the squared number of nodes, or as a range of outgoing edges for each node.

4.5.3 Graph Analysis

The analysis of graph structures and graph drawings is fundamental for the evaluation of the quality of graph layout algorithms. Rieß developed an analysis tool that operates on the KGraph data structure and is thus fully compatible with KIML [Rie10]. That tool has been used for the evaluations presented in Chapter 2 and Chapter 3. Figure 4.27 shows an Eclipse view listing analysis results. In the context of this tool, an analysis is an algorithm

4.5. Tools for Algorithm Developers



The screenshot shows a window titled "Graph Analysis" with two tables. The first table, "Basic Analyses", has two columns: "Analysis" and "Result". It lists three items: "Biconnected Components" with a result of 4, "Edge Count" with a result of 50, and "Node Count" with a result of 27. The second table, "Drawing Analyses", also has two columns: "Analysis" and "Result". It lists three items: "Area" with results "684.7111 (Width)" and "688.2222 (Height)", "Number of Bends" with results "0 (Minimum)", "0.46 (Average)", "2 (Maximum)", and "23 (Sum)", and "Number of Edge Crossings" with results "0 (Minimum)", "2.6 (Average)", "10 (Maximum)", and "65 (Sum)".

Basic Analyses	
Analysis	Result
Biconnected Components	4
Edge Count	50
Node Count	27

Drawing Analyses	
Analysis	Result
Area	684.7111 (<i>Width</i>) 688.2222 (<i>Height</i>)
Number of Bends	0 (<i>Minimum</i>) 0.46 (<i>Average</i>) 2 (<i>Maximum</i>) 23 (<i>Sum</i>)
Number of Edge Crossings	0 (<i>Minimum</i>) 2.6 (<i>Average</i>) 10 (<i>Maximum</i>) 65 (<i>Sum</i>)

Figure 4.27. The Graph Analysis View displaying results of a selected subset of available analyses. The number of bends and the number of crossings are given with the minimum, average, and maximum value per edge, as well as the total sum.

that computes values for one particular aspect of a given graph. Which subset of the available analyses is to be displayed can be freely configured. As an alternative to this UI, which can display the results of only one graph, analysis results for a whole set of graphs can be written to a file in CSV (comma-separated values) format, enabling further processing with spreadsheet applications or similar tools.

Each analysis is assigned to one of the categories *basic* or *drawing*. Basic analyses focus on structural properties of graphs and are independent of their concrete layout. The available basic analyses address

- the number of graph elements of each kind (nodes, edges, labels, ports),
- the number of multi-edges and self-loops,
- the average, minimum, and maximum node degree,
- the number of connected components and biconnected components,
- an approximate number of directed and undirected cycles,

4. Integration in Eclipse

- the longest path for acyclic graphs,
- the number of compound nodes (containing nested subgraphs),
- the average number of nodes contained in a compound node, and
- the number of edges crossing hierarchy borders of compound graphs.

Drawing analyses, in contrast, focus on aesthetic criteria derived from the concrete layout. They address

- the area and aspect ratio of the drawing,
- the number of edge bends,
- the number of edge crossings,
- the number of edges pointing left, right, up, or down,
- the average, minimum, and maximum edge length,
- the number of layers for layer-based drawings,
- the number of edge overlapping nodes,
- the average, minimum, and maximum size of nodes, and
- the number of ports placed on the north, east, south, or west side of their containing nodes.

New analyses can easily be added by contributing to an extension point with the identifier `de.cau.cs.kieler.kiml.grana.analysisProviders`.

4.5.4 Graph Formats

When working on graph algorithms it is sometimes necessary to access libraries of example graphs or tools from other research groups. There are numerous formats for the representation of graphs, hence the translation between these formats is an important requirement. In the KIML API such translations are made available through the class `GraphFormatsService`, which is designed similarly to the other service classes of KIML. Graph formats are registered with the extension point `de.cau.cs.kieler.kiml.formats.graphFormats`. In addition to some meta data such as an identifier string, a contribution to this extension point must specify an implementation of the in-

4.5. Tools for Algorithm Developers

terface `IGraphFormatHandler`. Such a handler class is responsible for a specific data structure F representing instances of the supported format. `IGraphFormatHandler` includes the following operations:

- A. parsing serial representations of the format, e. g. as read from a file or a network message, to instances of the data structure F ,
- B. transforming instances of F to instances of the `KGraph` format,
- C. transforming `KGraph` instances to instances of F , and
- D. serializing instances of F .

These four operations can be invoked independently of each other, allowing them to be flexibly composed according to the application needs. For instance, the connection of the `Graphviz` library discussed in Section 4.4.2 is realized by transforming from `KGraph` to `DOT` (Step C), serializing the result (Step D) and passing it to the `Graphviz` process, parsing the reply of the process (Step A), and finally copying the layout information to the original `KGraph`. The `KGraph` format is not only used for compatibility with the other `KIML` services, but also as intermediary format for translation between any other formats. The translation from a format \mathcal{F}_1 to another format \mathcal{F}_2 requires the application of Steps A and B with the format handler implementation of \mathcal{F}_1 and the application of Steps C and D with the implementation of \mathcal{F}_2 .

The following formats have already been connected to `KIML`:

- ▷ `GML` [Him97]
- ▷ `GraphML` [BELP13] (<http://graphml.graphdrawing.org/>)
- ▷ `OGML` (<http://ls11-www.cs.tu-dortmund.de/>)
- ▷ `DOT` (<http://www.graphviz.org/>)
- ▷ `JSON` (<http://www.json.com/>)
- ▷ `MatrixMarket` (<http://math.nist.gov/MatrixMarket/>)
- ▷ `SVG` (<http://www.w3.org/Graphics/SVG/>)

4. Integration in Eclipse

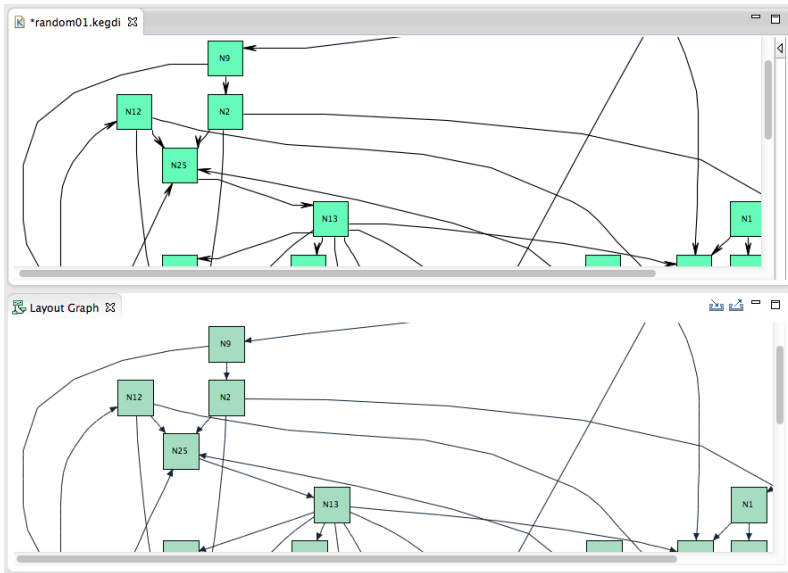


Figure 4.28. An instance of the KEG editor (above, with slight rendering errors) and the Layout Graph View (below, according to the computed layout) displaying the same graph after a layout has been computed.

4.5.5 Additional Tools

Layout Graph View. During the development of connections of diagram viewers to KIML (see Section 4.3), it is important to discern between faults caused by layout algorithms and faults caused by the connection to the viewer or the viewer itself. This is supported with the *Layout Graph View*, which draws the layout graph exactly as it is computed by layout algorithms. Figure 4.28 shows this view together with a KEG graph. In that example, subtle differences can be found between the layout computed by the layout algorithm and shown in the Layout Graph View and the layout applied to the GMF-based editor. Especially for edges incident to the nodes N9 and N13, the edge routing is corrupted by internal postprocessings done by GMF, leading to a disturbed appearance.

4.5. Tools for Algorithm Developers

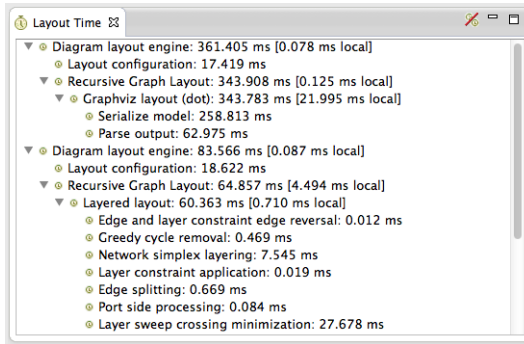


Figure 4.29. The Layout Time View displaying the runtime structure of executed modules of layout algorithms together with their execution times.

Layout Time View. The progress monitor passed to each layout algorithm (see Section 4.4) can be used to gather information on the runtime structure of algorithms and their execution time. This information is visualized in the *Layout Time* View as shown in Figure 4.29. For each layout algorithm that is invoked, the view displays a tree that corresponds to the execution of modules and submodules of the respective algorithm. The name of each module is printed, followed by its total execution time and, in brackets, the total time minus the sum of execution times of its submodules. For KLayout algorithms following the concept of intermediate processors (Section 4.4.1), the execution time of each processor is displayed, including the main phases of the algorithm.

Integration in Other Applications

The integration of a software library into applications that use different technological platforms is often an intricate task. In this chapter I present two approaches for integrating graph layout, one based on a Java class library and one based on a web service. Both use the KGraph format in order to connect the layout algorithms, hence they can reuse the interface described in Section 4.4. The interface presentations in this chapter are accompanied by examples where the graph layout infrastructure has helped to improve modeling applications.

5.1 Class Library

The most natural way to access a Java software library is through a JAR (Java Archive) file containing compiled class files. The KIELER project offers such a JAR with the layout algorithms developed in the KLayout subproject (see Section 4.4.1). This archive contains a selection of the KLayout algorithms, the basic classes of KIML defining the interface to layout algorithms, the KGraph data structure, and the basic classes of EMF required by that data structure. In order to integrate the library in a given modeling application, the graph format of that application must be mapped to the KGraph format, then a layout algorithm is invoked via its specific `AbstractLayoutProvider` implementation, and finally the computed layout is transferred to the application. This procedure is analogous to the connection of Eclipse-based diagram viewers through the `IDiagramLayoutManager` interface, described in Section 4.3.

The current version of the KLayout archive file is 2.2 MB large. 74% of that size is used by EMF code. In areas where program size is important, such

5. Integration in Other Applications

as online applications executed on the client side, it may be beneficial to omit the KGraph format and its EMF dependency and to use the internal data structure of the employed layout algorithm instead. This allows to include a much smaller archive at the cost of building on a more specialized graph structure, hence losing the compatibility to other layout algorithms. Which approach to take must be decided depending on the requirements of the particular application.

Two examples are presented in the following, a Java application using the full KLayout library with the KGraph format and a web application using only the KLayout layered algorithm adapted to JavaScript.

Ptolemy. The Ptolemy project¹ is mainly about modeling concurrency for real-time and embedded systems [EJL⁺03, Pto14]. The behavior of a system is modeled with *actors*, which communicate with their environment through ports. *Atomic* actors are predefined components from which larger models can be composed, while *composite* actors are defined by actor graphs, where the nodes are contained actors and the edges are connections between the ports of these actors. Ptolemy follows a *heterogeneous* approach to modeling the data flow of a system: each composite actor may define locally how to schedule the executions of its contained actors. The scheduling is determined by certain *models of computation*. Actor graphs are created with a Java application named *Vergil*, of which a screenshot is shown in Figure 5.1.

A first integration of a graph layout algorithm into Vergil using a class library was implemented early after the start of the KIELER project [SFvH09]. That integration has been improved in the following years [Fuh11, Chapter 5.4] and is now part of the Ptolemy software distribution. While the KLoDD layout algorithm [SFvHM10] was used in the beginning, it was later replaced by KLayout Layered [SSvH14]. Both were connected through the KGraph format, therefore changing the layout algorithm was mainly a matter of switching the invoked algorithm class, but the integration code could be kept as before. This flexibility outweighs the 1.6 MB overhead required for including the KGraph and EMF code in the integrated library.

Since most applications, including Ptolemy, do not support the extension

¹<http://ptolemy.eecs.berkeley.edu/>

5.1. Class Library

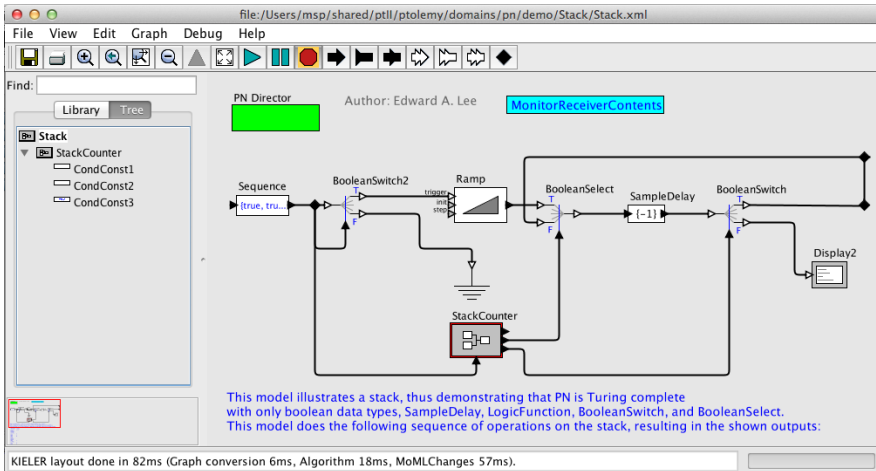


Figure 5.1. A screenshot of Vergil, a Java application for editing actor graphs and running simulations according to Ptolemy semantics. The layout of this actor graph has been computed with the integrated KLayout library.

points format of Eclipse, the KIML meta data discussed in Section 4.1.1 cannot be accessed. As a consequence, the more advanced layout configuration methods are not applicable. Instead of these, specific values of layout options must be set directly as properties of graph elements. While for some options fixed settings have been chosen, other options are offered to the users through a configuration dialog, shown in Figure 5.2. The settings of that dialog are stored with a typed attribute in the MoML file format used by Ptolemy, hence they are restored whenever the respective model is loaded.

ExplorViz. Monitoring is an important technique for the dynamic analysis of large applications, but the resulting data can be overwhelming, especially for *software landscapes* where multiple applications are connected. ExplorViz² aims at supporting the comprehension of software landscapes by visualizing

²<http://www.explorviz.net/>

5. Integration in Other Applications

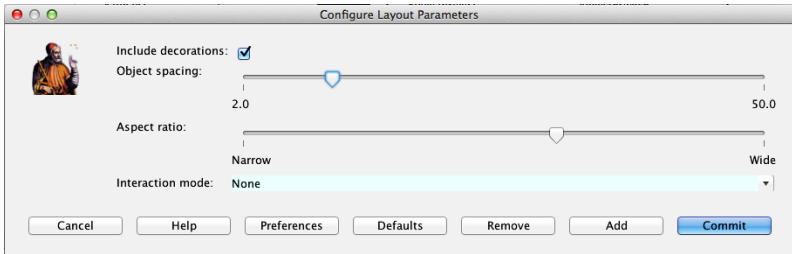


Figure 5.2. The layout configuration dialog integrated in Vergil controlling the behavior of the KLayout algorithm.

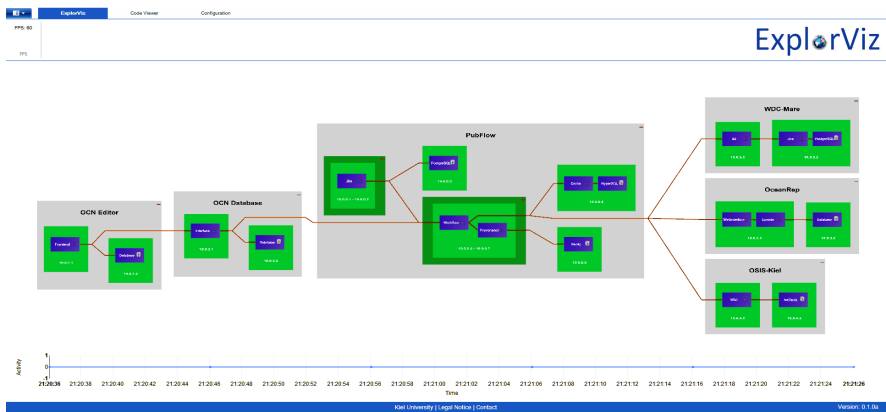


Figure 5.3. A software landscape visualization generated by ExplorViz using the KLayout layered layout algorithm.

monitoring traces [FWWH13, FvHH14]. The tool derives a landscape model from the trace data and then generates a visualization model based on the monitoring traces and the landscape model. The result is presented to the user through a web browser as shown in Figure 5.3.

The KLayout Layered layout algorithm has been integrated in ExplorViz by transforming its Java code to JavaScript with the GWT compiler.³ The

³<http://www.gwtproject.org/>

KGraph data structure has been omitted in order to minimize the program size, thus the visualization model of ExplorViz is mapped directly to the internal graph representation of KLayered.

5.2 Web Service

The idea of offering graph drawing algorithms as a web service has already been elaborated by Di Battista et al. [DLV95] and Bridgeman et al. [BGT99, BT04]. In his diploma thesis, Wersig has shown that the Eclipse-based graph layout infrastructure provided by KIML can be made available through a web service without any modification of its API [Wer11]. This means that the same layout algorithms, graph format translations, and meta data that have been connected to Eclipse-based applications are also available in the web. The main concept that allows reusing these features is the plug-in mechanism of Eclipse: the components defining the KIML infrastructure and the layout algorithms can be plugged into a server application. The server publishes the graph layout service with one or more web interfaces, e. g. a SOAP-based interface specified with a WSDL document [WCL⁺05]. Users of this service benefit from some very important advantages compared to other solutions such as a class library.

- Updates of the layout algorithms or the general infrastructure can be plugged into the server application without further adaptations. A running server can be updated with an automatic build process, hence the updated versions are available immediately to all clients.
- The web service interfaces are platform independent, allowing to integrate graph layout algorithms in applications written in C#, C++, or other languages that are not compatible to Java.
- The amount of code required on the client side is very small compared to the actual implementation of layout algorithms. This allows for lightweight integrations, e. g. in web pages using JavaScript.
- The integration of a web service access is generally less obtrusive than including a software library, and thus it may be more suitable for prototyping in situations where it is not clear whether the graph layout technology will actually be included in the final product.

5. Integration in Other Applications

The most obvious disadvantage is that a network connection to the server is required whenever a graph layout is requested. Furthermore, the client developers and users need to trust the server operator with respect to availability of the service and protection of data privacy.

Wersig implemented a SOAP-based web service for the graph layout server using JAX-WS⁴ and employed jETI [MNS05] for an alternative interface [Wer11]. Later Rüegg added an HTTP-based interface, which is easier to integrate in web pages. All these interfaces accept the following four arguments:

- a graph in serialized form,
- an identifier of the format of the input graph,
- an identifier specifying which format to use for the output, and
- a list of layout option assignments to be applied globally.

The reply message of the server contains the given graph enriched with layout information and serialized with the specified output format. The identifier strings for graph formats are specified through the graph formats service of KIML as explained in Section 4.5.4. The formats of the input graph and the output graph may be different, realizing the idea of Bridgeman et al. on a *graph drawing and translation service* [BGT99]. Layout options can be assigned globally with key-value pairs, where the keys are identifier strings of layout options (see Section 4.1.1) and the values are given as string representations. For instance, the pair (de.cau.cs.kieler.noLayout, true) indicates that no layout algorithm shall be executed, effectively reducing the web service call to a graph translation from the input format to the output format.

The development team of KIELER hosts an instance of the graph layout server that is updated regularly.⁵ In the following, examples of applications that make use of this service are presented.

Command line tool. A very small Java program (52 KB) for command line access to the graph layout service is available on the KIELER download

⁴<https://jax-ws.java.net>

⁵<http://layout.rtsys.informatik.uni-kiel.de:9444/>

page.⁶ The program can read a graph file, send it to the server, and write the result to another file. For instance, the command

```
java -jar kwebs.jar infile=graph0815.kgraph outfile=graph0815.svg
```

reads the file `graph0815.kgraph` in the KGraph format and requests a drawing in SVG format from the server, which is then written to the file `graph0815.svg`. The program also supports standard input and output, allowing it to be chained with other command line tools as in this example:

```
cat graph0815.graphml | java -jar kwebs.jar informat=graphml \
    outformat=dot | dot -T svg -o graph0815.svg
```

Here the file `graph0815.graphml` is first translated from GraphML to DOT using the web service, and the result is then processed by the Graphviz Dot program⁷ to generate an SVG file.

MDElite. Based on their observation that many computer science students find it difficult to understand the MDE tools of Eclipse, Batory et al. investigated the use of alternative tools for teaching MDE concepts [BLA13]. As a case study, they implemented transformations of class diagrams between three different UML tools. In this context it was necessary to generate positioning information of classes, which was done by transforming the models to DOT graphs (the file format of Graphviz), sending requests to the KIELER web service, and reading the positioning information from the returned graphs. The service requests were performed with the command line tool presented above. The tool set was successfully included in the assignments of an undergraduate course at the University of Texas at Austin [BLA13].

GraphArchive. During a discussion at a seminar in Dagstuhl, Germany, it became clear that the graph drawing research community lacked a system for consistently collecting graphs used in experiments [DKKM11]. That discussion led to a cooperation on the development of GraphArchive,⁸ a

⁶<http://www.informatik.uni-kiel.de/rtsys/kieler/>

⁷<http://www.graphviz.org>

⁸<http://www.graph-archive.org/>

5. Integration in Other Applications

database of graphs hosted at the University of Tübingen [BBE⁺11]. The database is accessed through a web interface where registered users can browse the existing graphs, download them, and upload new graphs. When a graph is accessed, users may choose one of several formats to download it, irrespectively of which format was originally used when the graph was uploaded to the database. This is realized through graph format translations, for which the KIELER web service is employed.

Survey

The concepts presented in this thesis have been evaluated with a web-based survey among users of the KIELER layout infrastructure. The survey was built with an online service¹ and was accessible through a web URL. It was open from November 2012 to July 2013. The two main goals of the survey were to gather feedback on the concepts presented in this thesis and to obtain insights on the requirements imposed by practical applications.

The full questionnaire is provided in Appendix C. It begins with a welcome page that motivates the survey and poses a few simple questions, as recommended by Dillman et al. [DTB98]. Some other principles stated by them are already fulfilled by the employed tool for survey construction, e. g. limiting the line length for better readability, or displaying the current progress status of a participant. In a study by Ganassali [Gan08] long questionnaires led to higher drop-out rates of participants compared to shorter versions, but surprisingly, they also led to longer responses to open-ended questions and to higher satisfaction of the participants. Therefore I chose a medium length for the questionnaire of this survey, with at most 32 questions (depending on how many questions are skipped because they are not applicable, see Appendix C), requiring about 20 minutes for participating.

Questions in a survey can be categorized into closed-ended questions, where participants can choose from a set of predefined answers, and open-ended questions, where they are free to write arbitrary text. I chose to include both types of questions, since each type has its specific advantages and disadvantages. As shown by Reja et al. [RLHV03], open-ended questions result in higher diversity of answers, but the answers are more

¹<http://www.surveygizmo.com/>

6. Survey

difficult to code, i. e. to extract relevant information out of them. The answers to closed-ended questions, in contrast, are restricted to what has been proposed by the survey designer, but they can be evaluated easily.

The detailed results of the survey are presented in Section 6.1, starting with general properties and requirements of modeling applications and then addressing the quality of the KIELER layout infrastructure. The results are discussed in Section 6.2. Question numbers are given as listed in Appendix C.

6.1 Detailed Results

78 persons participated in the survey, not counting those who dropped out on the first page (Questions 1, 2, and 3). Of these, 55 gave full responses and 23 gave partial responses. This corresponds to a drop-out rate of 29%, which is very similar to the rate of 27% measured in a study of Ganassali [Gan08]. Most participants (71%) were invited via email, while the others followed a link advertised on the KIELER homepage. This demonstrates that personal invitations can be much more effective than a general call for participation.

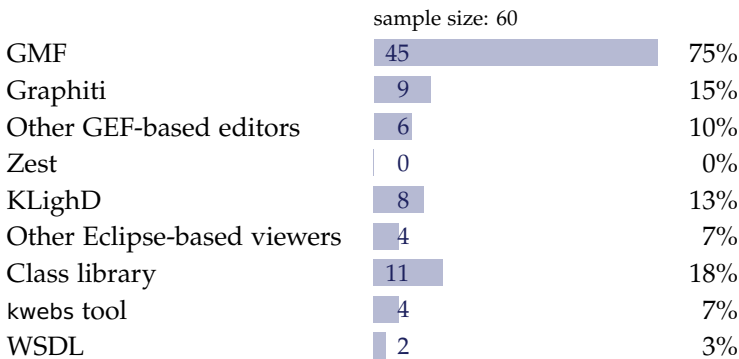
The employed survey tool was able to track the location of all participants except three. 56% of the tracked participants were from Germany, and 24% were from the rest of Europe.

General remarks. For some open-ended questions the answers were encoded such that similar answers could be grouped and counted, e. g. Question 22. Although in these cases the results are presented in the same format as for closed-ended questions, the relative frequencies of open-ended questions tend to be lower, hence they should be interpreted with care: the actual values of relative frequencies are not relevant, but rather their comparison. The relative frequencies are represented with horizontal bars followed by the corresponding percentage values, while absolute frequencies are written on these bars. For open-ended questions that have been encoded, groups with less than three responses were combined in a group named *other*. The *sample size* is the number of persons who responded to a particular question.

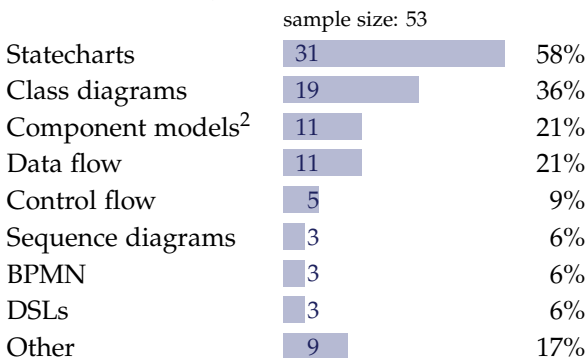
For other open-ended questions the responses are given directly. In order to ensure anonymity, the responses are given in random order, and names, projects, etc. of the respondents were removed. Furthermore, responses were filtered such that only statements relevant to the respective questions are shown here. The responses were corrected where necessary, and translated to English where given in German.

6.1.1 Application Requirements

Question 4: Modeling application. How have you been using KIELER layout?



Question 22: Model types. What kind of models do you work with?



²including architecture diagrams, structure diagrams, etc.

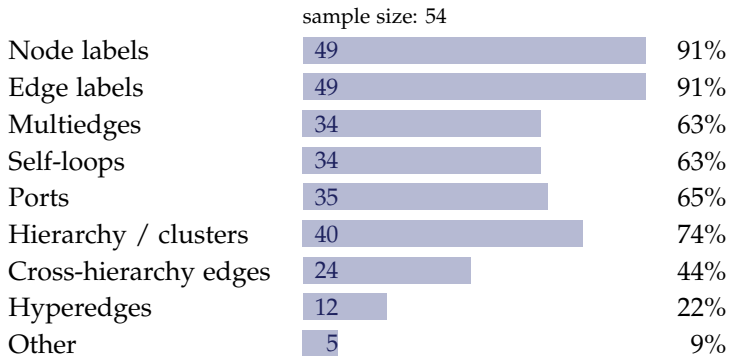
6. Survey

Question 23: Graph sizes. The following table shows statistical values on the upper and lower bound of typical graph sizes, according to the number of nodes, stated in responses to this question. Most participants responded in the form $x-y$, meaning that typical graphs of their applications have at least x and at most y nodes. Responses with only one value were included both in the lower bound and in the upper bound statistics. The column titled “Samples” refers to the sample size.

	Samples	Average	Std. dev.	Median	Min.	Max.
Lower bound	44	154	629	10	2	3 000
Upper bound	48	1 173	3 359	50	7	20 000

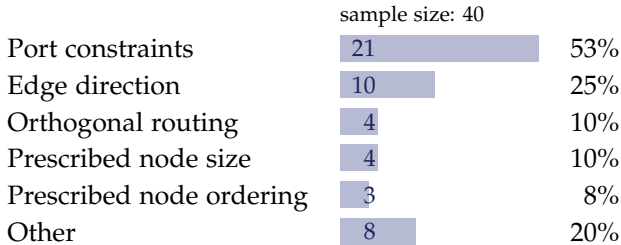
Question 24: Graph features. For this multiple-choice question responses to the *other* category could be written in a text box. However, no graph feature was mentioned more than once in that category.

Which of the following are typical features of your graphs?

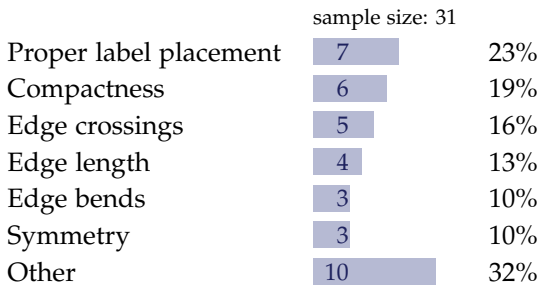


Question 25: Domain-specific constraints.

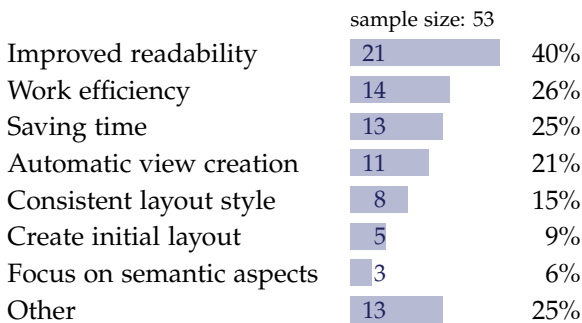
Which domain-specific constraints are required for the layout of your graphs?

**Question 26: Personal preference.**

Which aesthetic criteria would you add as your personal preference?

**Question 27: Benefits.**

Which benefits do you expect from automatic graph layout technology?



6. Survey

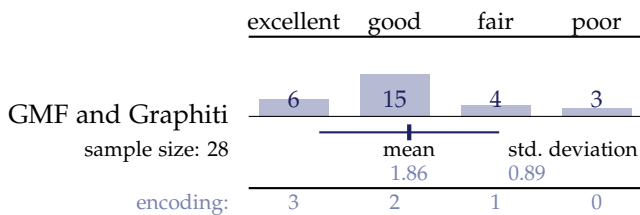
6.1.2 Quality Evaluation

81% stated that they had already used KIELER, while 19% stated that they had not. The questions on quality evaluation were shown only to the participants who have used KIELER.

Four answers were possible for questions on the rating of quality: *excellent*, *good*, *fair*, and *poor* (responses selecting *not used* were not counted in the evaluations). Here the relative frequencies of these answers are represented with vertical bars, and the absolute frequencies are written on these bars (see e. g. the evaluation of Question 6). Furthermore, the possible answers were encoded with numbers from 0 (poor) to 3 (excellent), allowing to compute mean values and standard deviations. As seen for Question 6, a mean value μ is represented with a short vertical line positioned on a scale between these two extremes. A value of $\mu = 1.86$, for instance, can be interpreted as “between *fair* and *good*, with a strong tendency towards *good*”, which is also expressed by the position of the vertical line. The standard deviation σ is shown with a horizontal line between the positions $\mu - \sigma$ and $\mu + \sigma$. This illustrates the amount by which the responses are scattered: a long horizontal line indicates high scatter, while a short line indicates low scatter.

Question 6: Generic integration.

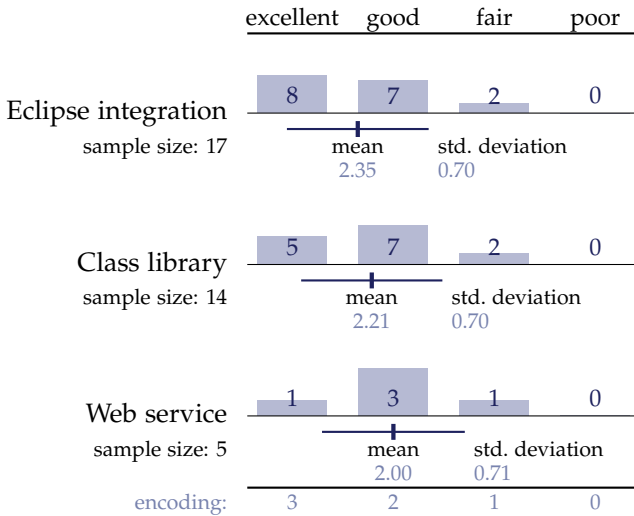
How well does the generic GMF / Graphiti layout integration work for your editor?



Question 8: Flexibility of integration interfaces.

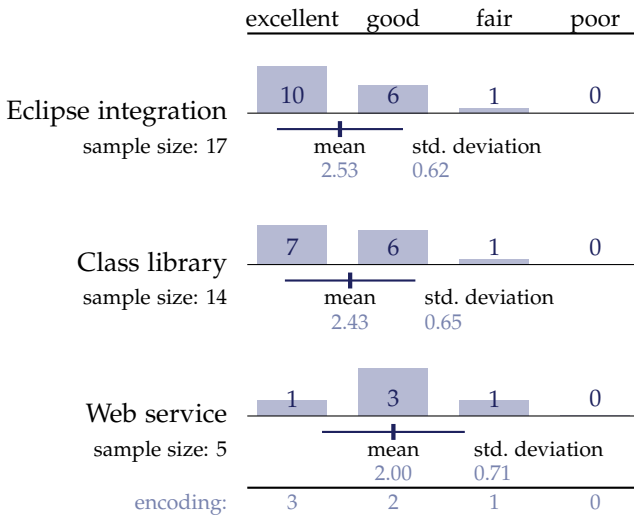
Please give us your opinion on the quality of KIELER Layout integration interfaces with respect to flexibility.

6.1. Detailed Results



Question 9: Efficiency of integration interfaces.

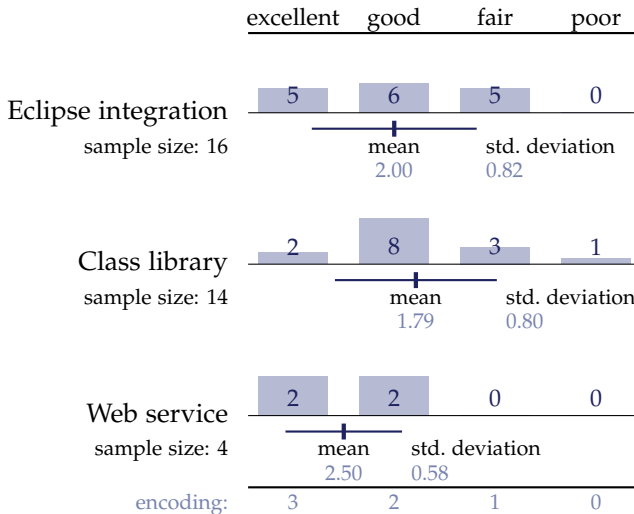
Please give us your opinion on layout integration interfaces w.r.t. efficiency.



6. Survey

Question 10: Simplicity of integration interfaces.

Please give us your opinion on layout integration interfaces w.r.t. simplicity.



Question 11: Experience with layout integration.

Please describe further aspects of your experience with KIELER Layout integration interfaces.

- ▷ Good options to configure the layout.
- ▷ KIELER seems to be a sound framework, easy to integrate though powerful. It can be customized and tweaked in many ways.
- ▷ The layouting feature which KIELER provides is excellent, integration can be done easily, good options to configure node and port positions.
- ▷ Documentation was somewhat limited such that I needed custom-made examples to see how to use the KGraph and its additional structures like KLayoutData. The patterns how to use layout options in code and how to find out which options are available were not trivial.
- ▷ The documentation on how to integrate KIELER plugins with different graphical editors is missing. The information on the site has nothing

related to integrating KIELER with different GMF based editors and how to customize the layout options and other things.

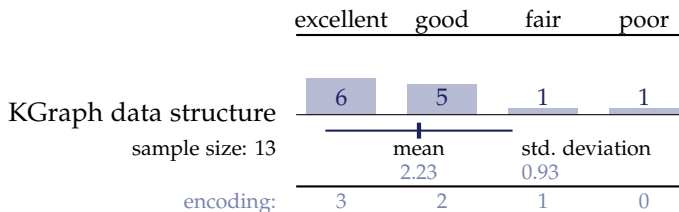
- ▷ Out of the box KIELER is relatively invasive regarding UI elements. This is not desired if a “silent” integration is required.
- ▷ Would love to see some more examples; specifically it is fairly hard to know which properties should be set where – it took a long time just to find a way to get the edge labels showing, for instance.
- ▷ I would love to see a Javadoc, accessible through the web. It took me a long time to gather enough information from the Confluence and Git examples to finally implement KIELER the way I needed.
- ▷ There are occasionally a few small UI problems, e. g. menu items being shown even though there is no GMF diagram open.
- ▷ There should be more documentation and bigger examples of how to apply the layout algorithms, because it is difficult to get started using KIELER, especially using the KIELER class library for layouting in Eclipse.
- ▷ Installation of the right plug-in version is a hassle, in particular when aiming to use KIELER with other plugins such as Epsilon.
- ▷ Documentation could be enhanced.
- ▷ By default, KIELER makes massive contributions to the Eclipse IDE, e. g. a very prominent KIELER menu, which is not acceptable for more subtle applications and RCPs.
- ▷ I did not understand the purpose of every method of the layout managers interface right away. KWebS needs a little work on the developers site to create a proper graph representation such as GraphML. But with this done, it is straightforward to use the web service. Improvements may include the documentation of the layout options. I know there is some, but it takes me a while to find the address in Confluence.
- ▷ Excellent flexibility. Very impressive results. May improve on the heuristic calculations for node placing.

6. Survey

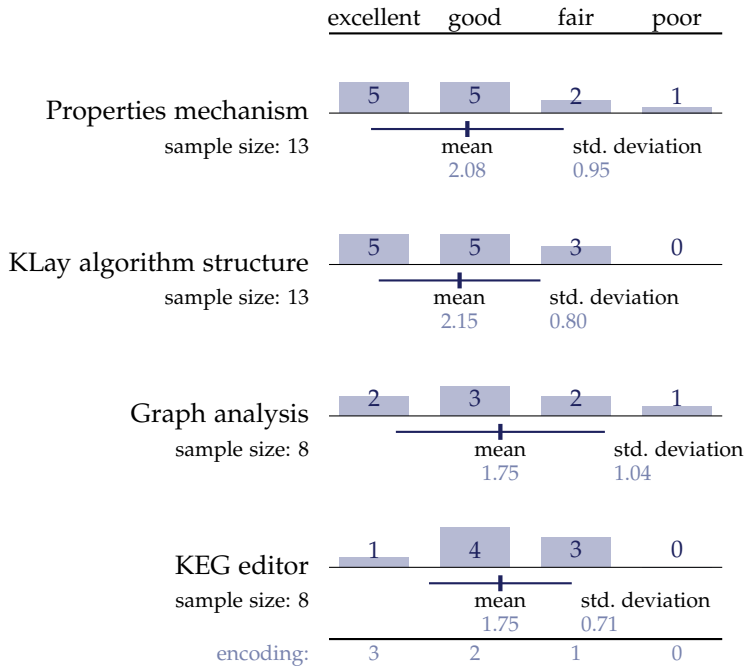
- ▷ The layout integration interface for Graphiti editors was very intuitive and easy to adapt for our editor. The meta models and concepts were very well laid out. The documentation was excellent and was very helpful for understanding the KIML framework as well as throughout the process of integration. Especially the Javadoc comments were really very good and useful. Throughout the implementation process, huge support was provided by the KIELER team.
- ▷ In some cases auto auto-laying was triggered where it was unwanted, even when turning every possible option off.
- ▷ Documentation could certainly be improved. A huge benefit is the integration of several different layout algorithms under the umbrella of a single layout architecture.
- ▷ I suggest to make a good documentation. KIELER is a great tool, but it needs a good documentation revealing its capabilities and functionalities.
- ▷ Easy integration through libraries like `DiagramLayoutService`, taking arbitrary set of layout options; documentation of the coordinate references on Wiki pages was extremely helpful.
- ▷ The automatic layout and customizing of layout is flexible enough to adapt. Initially it was a bit tricky, but once you get used to it, you will find it easy to integrate.

Question 13: Flexibility of algorithm development tools.

Please give us your opinion on the quality of the layout algorithm development interfaces and tools with respect to flexibility.

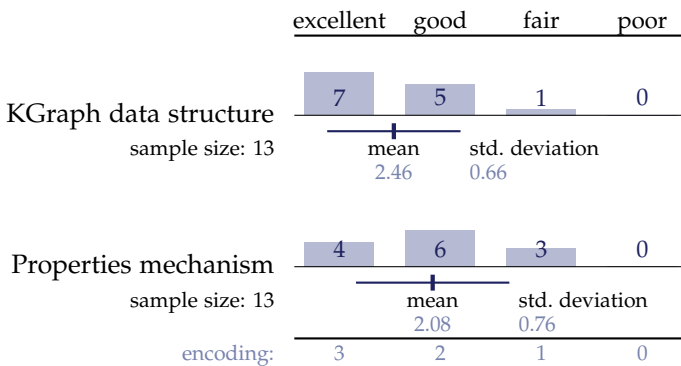


6.1. Detailed Results

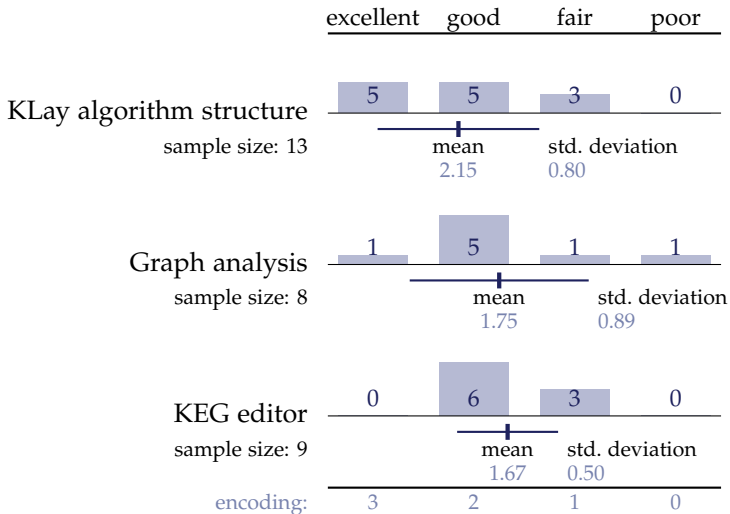


Question 14: Simplicity of algorithm development tools.

Please give us your opinion on algorithm development tools w.r.t. simplicity.



6. Survey



Question 15: Experience with algorithm development tools.

Please describe further aspects of your experience with our layout algorithm development interfaces.

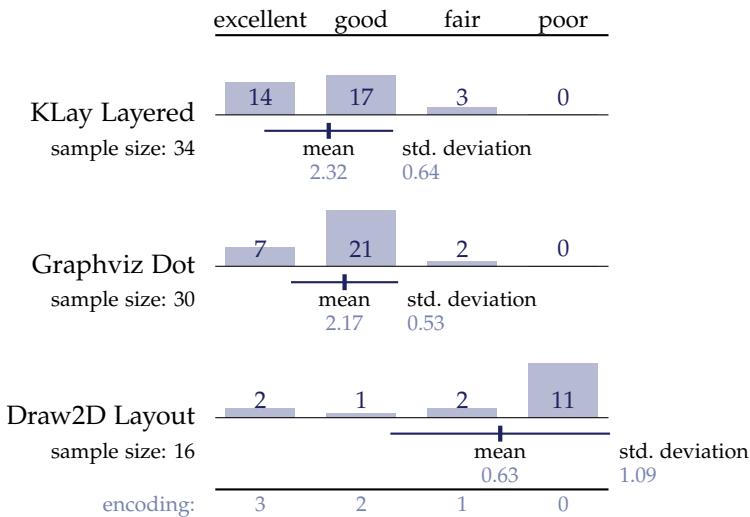
- ▷ Splendid support!
- ▷ Some debug and analysis tools may need some development and tweaking (e. g. a generalization of the performance measurement tool), but it is already nice that they are there in the first place.
- ▷ Again, the documentation is a downside. Probably, a hands-on tutorial on creating layout algorithms for KIML would be good. The extension point structure in particular is not straightforward and could use more documentation.
- ▷ More details in the documentation would be helpful.
- ▷ Needs more documentation and larger examples. It is not easy to find out what can be done with the layout algorithms.

- ▷ KGraph is simple to understand and in my opinion covers all basic concepts that are needed. The property mechanism itself seems to be very generic and usable in the same manner at very different spots. Hence, once familiar with it, it might be very flexible. But at the start it can feel a bit like looking for the right identifiers all over the place. On the other hand, it allows concise classes, as not every property needs its own getter and setter method.

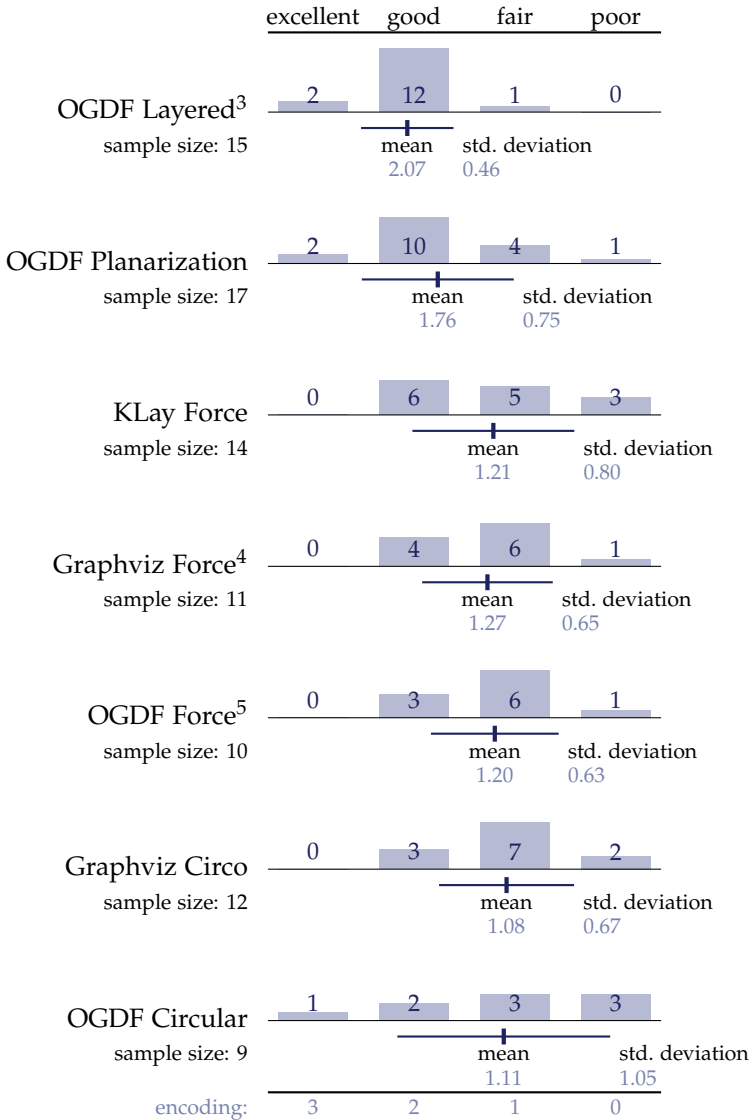
- ▷ I found both the algorithm and graph structure very nice and useful. Although I used a self-made graph structure, in retrospect it might have been better to use the KGraph structure directly. Personally, I did not like the property mechanism very much, but it is definitely useful and nice to have, and probably the best possible solution in Java.

Question 16: Quality of layout algorithms.

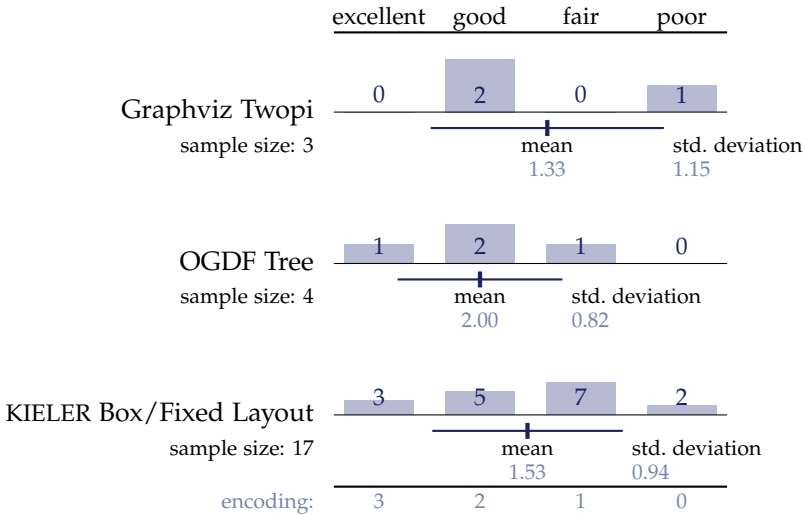
Please give us your opinion on the overall quality of layout algorithms.



6. Survey

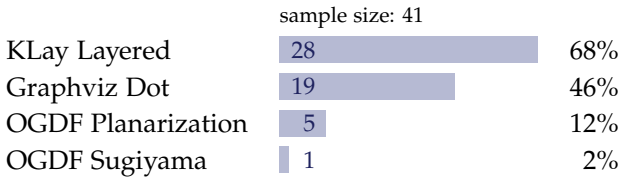


³i. e. Sugiyama and Upward Planarization



Question 17: Most used algorithms.

Which layout algorithms have you used most?



Question 18: Strengths and weaknesses of algorithms.

Please describe strengths and weaknesses of your most used algorithms.

Referring to KLayout Layered:

- ▷ KLayout Layered is very suitable for nearly any kind of data-flow like diagrams.
- ▷ The KLayout Layered algorithm doesn't move connection labels properly.

⁴i. e. Neato and FDP

⁵i. e. Davidson-Harel, Fruchterman-Reingold, Kamada-Kawai, FMMM, and GEM

6. Survey

- ▷ KLayout Layered sometimes generates superfluous bend points, and sometimes there might be more optimizations possible w.r.t. edge crossings minimization.
- ▷ The graph structure is quickly understood, adjacent nodes are easily identified. A weakness is the relatively monotonous representation, i. e. there is hardly any distinctive node constellation.
- ▷ KLayout Layered is very good but lacks good label placement.
- ▷ The layout algorithm fits exactly the needs of typical block diagrams. It supports ports and nesting, but there are often still too many unnecessary crossings in the layout.
- ▷ The built-in layout algorithms are great for my purposes. They very rarely are a little unpredictable, e. g. rearranging large parts of a diagram when I close or open a compartment.
- ▷ The layouts produced for block diagram models clearly make the models more understandable, especially when nodes are expanded. The left-to-right orientation according to data flow makes KLayout superior to other state-of-the-art layout algorithms. Some feedback by users is that the produced layouts could be improved by using less whitespace.
- ▷ KLayout Layered produces too much whitespace around the diagram.
- ▷ I have noticed some limitation in KLayout Layered when dealing with graphs with cycles.
- ▷ A general weakness is the label placement. Labels were partly positioned over other components of the graph which made some of them unreadable. Sometimes it is also hard to identify the component a label belongs to by its placement.
- ▷ Some seemingly unnecessary bends in edges.
- ▷ KLayout Layered has a large number of options for configuring layout and works quite well with hierarchical diagrams. However, it is unable to properly handle self-loops and edge labels, and edges originating from ports sometimes overlap with the nodes.

- ▷ It is not always understandable how the layout changes with using different layout options.
- ▷ KLayout Layered sometimes produces too much whitespace around the diagram.
- ▷ There are too many options for the user to change.
- ▷ Dot and KLayout Layered produce quite compact diagrams. KLayout reflects symmetries much better than dot.
- ▷ Very good and readable results in most cases, fast. Very well applicable to languages with a strong emphasis on data flow. Sometimes there are unnecessary bend points or edge crossings. The algorithm tends to generate very wide layouts as more layers are used. There is only basic support for interactive layouts and no concept of grouping nodes together other than through compound nodes.
- ▷ Support for labels on edges is missing. The shape of nodes is limited to boxes, so edge end points do not fit to other shapes such as circles.
- ▷ Strengths: readability, edge routing, node placement. Weaknesses: label placement, explanation of layout options.
- ▷ KLayout Layered is fast and offers very readable layouts that do not consume too much space. It is also very versatile and can be customized in many ways.
- ▷ The distance and order between nodes was quite satisfying, and the adaptability of ports and node dimensions was exactly what I needed. However, some nodes were placed diagonally of the connected node instead of being on the same level.

Referring to Graphviz Dot:

- ▷ Dot sometimes reorders nodes such that the mental map is lost.
- ▷ Excellent generic layout algorithm, supports clusters and nested nodes. However, it does not support manhattan layout and has limited support for nested structures and ports.

6. Survey

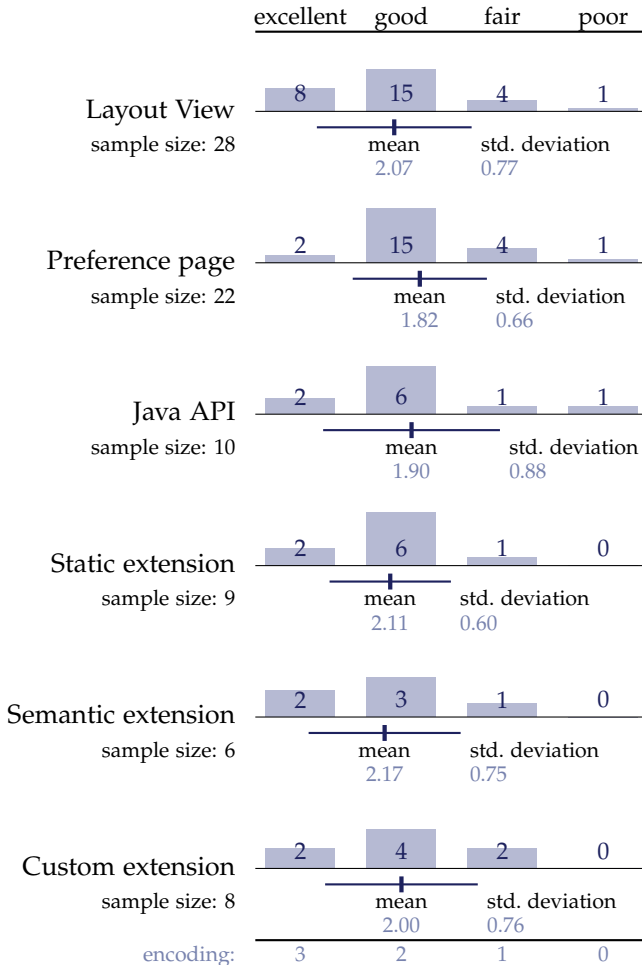
- ▷ Dot sometimes creates weird edges.
- ▷ Very good for most graphs, but problematic for circular control flows.
- ▷ Dot works great with state machines and gives very appealing layout. However, I have found a few instances when two different ports on a node overlapped resulting in an unacceptable layout.
- ▷ Dot shows very good results in most cases and is superior to the Sugiyama algorithm of OGDF.
- ▷ Dot has nice spline arcs, but sometimes they are too close to each other. Also, Dot sometimes fails to show inherent symmetry even within simple models. The same applies to other algorithms such as Planarization (OGDF).
- ▷ Overall Dot gives a nice layout. However, the placement of tail labels, as used in SyncCharts for transition priorities, is poor. I also sometimes miss the opportunity to manually enforce some ordering. For instance, when creating two somewhat similar diagrams, such as before and after transforming some part of the diagram, it is often desirable to have these diagrams look as similar to each other as possible.
- ▷ The algorithm works with all kinds of graphs – the results look good. Dot makes a serious approach on edge routing with spline curves.

Referring to OGDF Planarization:

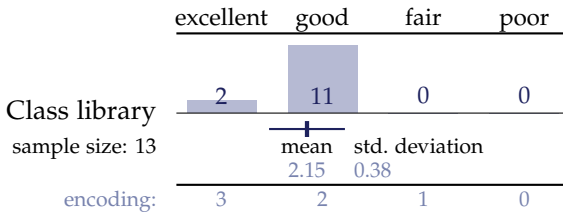
- ▷ Planarization is the only algorithm suitable for class diagrams, but does not give good results. Missing label placement is the biggest issue. Very long edges in favor of crossing minimization lead to complex long wires for larger diagrams.
- ▷ Very clearly arranged layout.
- ▷ Planarization-based diagrams tend use a lot of screen space.
- ▷ OGDF has a GPL license, thus it is not usable in commercial or differently licensed use cases.

Question 19: Flexibility of layout configuration interfaces. For this and the following question, the two groups on layout configuration via the web service received less than three responses, hence they are not shown here.

Please give us your opinion on the quality of KIELER layout configuration interfaces with respect to flexibility.

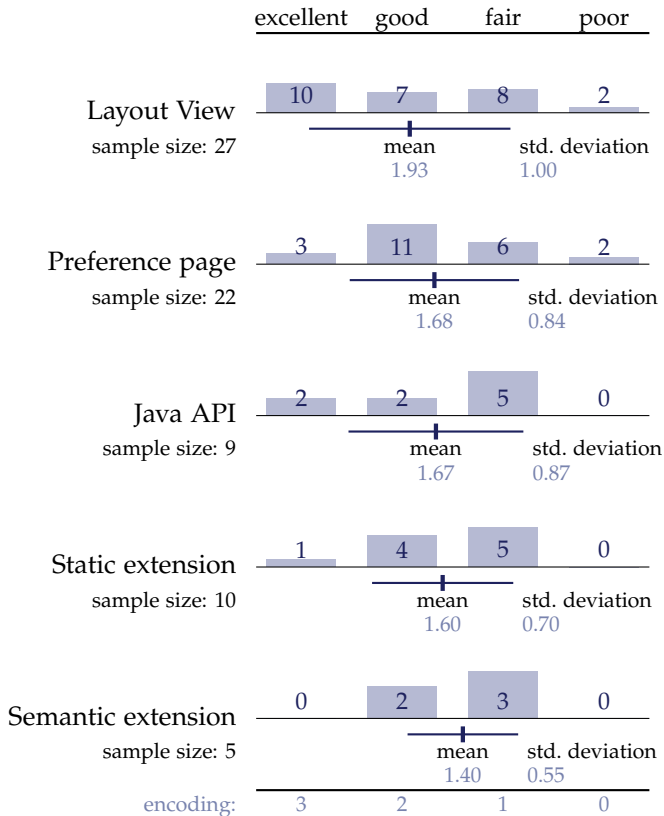


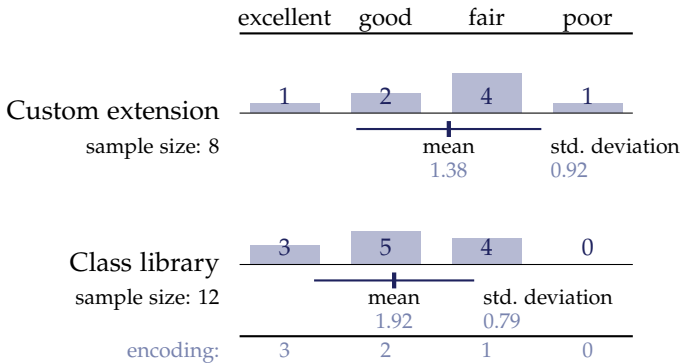
6. Survey



Question 20: Simplicity of layout configuration interfaces.

Please give us your opinion on layout configuration w.r.t. simplicity.





Question 21: Experience with layout configuration interfaces.

Please describe further aspects of your experience with KIELER layout configuration interfaces.

- ▷ The Layout View provides a good overview of possible configurations while preserving clarity. The web service offers an excellent way to decouple layout options from each individual graph representation, and allows great flexibility. Good documentation would be very helpful at this point.
- ▷ The programmer interfaces definitely need more documentation and examples; the preference page is rather intuitive.
- ▷ The KIELER Wiki provides good documentation.
- ▷ My major problem is documentation, which should contain more code examples.
- ▷ Sometimes there is a mixture between high level configuration and configuration that expects detailed understanding of the layout algorithms.
- ▷ The UI is intrusive: KIELER items show up in menus everywhere in Eclipse, even when not applicable. I cannot get a good layout of an Ecore diagram in Ecore tools, only a half decent one. The produced layout is difficult to modify by hand due to the added routing points on edges.

6. Survey

- ▷ The preference page needs more documentation on how the options affect the graphs.
- ▷ I find it tedious to tweak the layout via the Layout View.
- ▷ The standard layout control mechanisms requires several mouse clicks to see the effect of a certain choice in the Layout View.
- ▷ One thing that always bugged me a little about layout options was that they have to be defined in the plugin.xml as well as in the Java code.
- ▷ For some graphs, certain algorithms are not applicable. I would recommend to display only applicable algorithms.
- ▷ It would be nice to have more common configuration options, like minimal node distances, making it easier to compare the algorithms directly.
- ▷ Layout options often lack proper and detailed documentation.
- ▷ Layout preference page changes should be applied immediately, at least as an option.
- ▷ In our product, we do not offer the layout options directly to users, as this would be too complex. Rather, we define a configuration that works best and fine-tune it based on user feedback.
- ▷ The configuration interfaces are good, intuitive, and easy to use.

Question 28: Other used layout tools.

Which graph layout tools (other than KIELER) have you already used?

sample size: 47

Graphviz	19	40%
yEd / yFiles ⁶	10	21%
GMF / Draw2D	4	9%
Enterprise Architect ⁷	3	6%
Other	10	21%

⁶<http://www.yworks.com/>

⁷<http://www.sparxsystems.com/>

Question 29: Advantages of other layout tools.

Please state good features and advantages of other layout tools.

Referring to Graphviz:

- ▷ Lightweight, fewer bugs, easier to use.
- ▷ Graphviz is very general and has a very simple file format, but also very strong features and layout algorithms.
- ▷ Graphviz can be used via command line and can produce output files in different formats.
- ▷ Graphviz is quicker to use: write the graph in the DOT language and you are done. KIELER has more features and dynamical interaction – it is more than just the layout engine.
- ▷ Support for command line; very compact DOT format.
- ▷ Usually fast, good layout, multi-platform, simple input format.
- ▷ Simple, text-based input syntax.
- ▷ Generally good layouts.
- ▷ Dot creates nice graphs from a very simple textual interface, which is however not well suited for our case.
- ▷ Graphviz is easy to use as a command-line based tool, provides a simple textual language for specifying graphs and styling, and produces images. However, it is not interactive at all.
- ▷ Fast, well-documented, stable.

Referring to yEd / yFiles:

- ▷ It is really difficult to tell the features of KIELER from the web page without having tried, so a comparison is simple: yWorks comes with a better functionality overview and a better set of examples, so the barrier is lower.

6. Survey

- ▷ yFiles is much better and also offers partial relayouting on changes.
- ▷ yFiles offers professional support and is quite mature.
- ▷ Easy to learn and to use.
- ▷ yWorks provide tooling (a demo application) that is easier to use compared to KIELER.
- ▷ Integration into existing GMF / GEF based editor is very simple, results are quite good.
- ▷ KIELER layout is usually applied on the whole diagram after a model change has taken place. yEd supports the user in placing diagram objects while the model modification is still in progress, that is, while the user is still dragging new nodes into place. This keeps the user's mental model valid, while applying the current generation of KIELER layout technologies may rearrange the whole diagram when applied.

Question 30: Disadvantages of other layout tools.

Please state problems and deficiencies of other layout tools.

General comments (referring to multiple tools):

- ▷ Some tools can only be applied to the whole diagram and not to a selected subset. Once you have made some manual adjustments, you won't use them anymore or you'll lose these adjustments. They often ignore generally accepted guidelines (e. g. generalizations upwards, associations sideways) and require a lot of post-treatment. They don't care about semantics, e. g. which classes are most important or more strongly related than others. A human would consider this in his layout.
- ▷ The major problem is the lacking support for clustered graphs and inter-hierarchical edges.
- ▷ Hard to integrate, lack of configuration parameters, slow, interaction can be impacted adversely.

6.1. Detailed Results

- ▷ The layout algorithms of GraphViz, TomSawyer, and yWorks do not generate layouts that fit to the problem domain of block diagram models with ports on entities.
- ▷ I haven't found any other layout tool that is cost free, does orthogonal layout, and is not bound to a specific widget toolset such as SWT, AWT, Swing, etc.
- ▷ Not very flexible, often limited to single aspects of layout.
- ▷ Other tools are more difficult to use and feel static, as if very simple layouters were used.
- ▷ Few layout algorithms available, no flexibility, often poor results.
- ▷ Only semi-automatic layout, or hard manual configuration needed.

Referring to Graphviz:

- ▷ No orthogonal layout.
- ▷ With dot, sometimes the edge routing seems a bit weird. Also, while there seem to exist parameters to enforce a particular layering and ordering, I did not really get these to work yet.
- ▷ Graphviz documentation is partly hard to read. Meaning and interaction of layout options is sometimes unclear.
- ▷ Not as easy extendable.
- ▷ Lack of rectilinear layout with constrained positions of nodes and edges.
- ▷ Not easy to influence the outcome – no interactive mode.
- ▷ Lacking port support.

Referring to yEd / yFiles:

- ▷ Not useful for advanced layouting.
- ▷ yFiles costs a substantial amount of money. It offers only restricted location constraints and layouting of ports.

6. Survey

- ▷ yFiles is expensive.
- ▷ yFiles is expensive and closed source.
- ▷ The price.

Referring to other tools:

- ▷ The Ptolemy II layout algorithm produced very ugly layouts.
- ▷ Zest is only applicable for special kinds of graphs.
- ▷ Enterprise Architect costs money and layouts are poor, i. e. not usable.

Question 31: Final comments.

- ▷ Location constraints are really important in our context.
- ▷ KIELER is a very useful toolkit for automatic layout. If the algorithms can scale up in performance for huge data sets, they can become a standard.
- ▷ I love the web site where I can simply send files and receive files as output. I hate trying to figure out how to call library packages; I don't want to know the details of graph structures or methods.
- ▷ KIELER is a very promising project.
- ▷ It would be really nice to see the Dot layout algorithm implemented in plain java, so that there is no need to install Graphviz anymore. Keep up the good work!
- ▷ I think KIELER is already a very good opportunity to layout statecharts, but it still needs some further development to support all features of statecharts. It would be easier to use with a more detailed documentation, especially for the programming interface.
- ▷ Great work!
- ▷ A clearly stated release plan would be fine, such that I know when I have to resynchronize my code with KIELER. Also some API policy would be nice: what is fixed, what is provisional or internal.

- ▷ Great choice of state-of-the-art layouts, free and open-source, very helpful developers. A better documentation would be nice, perhaps a tutorial. Currently the entrance level is high, at least for people that have never worked with this kind of programs.
- ▷ Please, make a documentation.
- ▷ Please ensure development is sustained; without updates the tool will die for sure.
- ▷ Could KIELER support custom requirements by the user, e. g. just lay-out a special part of a graph? The label placement could use some optimization.
- ▷ Overall it is a good effort. No such brilliant open source tool as KIELER available, I think.
- ▷ I am looking forward to trying it!
- ▷ I am looking forward to continue working together with the KIELER framework and the people behind that!
- ▷ There is plenty of further potential in automatic layout, and many fields of application that can be involved in the future.
- ▷ It has been a pleasure to work with the members of the KIELER group. The KIELER layout mechanism is very high performance.

6.2 Discussion

The survey has addressed many different aspects of the KIELER layout infrastructure and algorithms. The overall feedback is very positive and confirms the usefulness of the concepts presented in this thesis. Quite naturally, a number of problems have also been identified, some of which can be attributed to the fact that KIELER is an academic project and therefore has a different focus compared to usual open source projects or commercial products. The most prominent problem, mentioned many times in responses to

6. Survey

open-ended questions, is the lack of documentation. The scientific articles about KIELER⁸ have a rather abstract view that is not always helpful for users who seek to understand certain technical details. The technical documentation is published in a Wiki,⁹ but in the past, the complexity of KIELER software has grown a lot faster than its Wiki pages. One reason for this is that the software is subject to frequent changes due to the prevalent focus on innovation and research, hence there are concerns that documentation may quickly be outdated. While a definite solution on how to manage the technical documentation of a constantly evolving academic software has not been found yet, the documentation of the layout infrastructure has been considerably extended in reaction to this result of the survey.

More results are discussed in the following, starting with application requirements and then covering the evaluation of quality.

Application requirements. From the responses to Question 4 it can be seen that GMF is by far the most used modeling tool in which layout algorithms provided by KIELER have been applied. The most commonly used model types are Statecharts, class diagrams, component models, and data flow models (Question 22). As seen with Question 24, labels are found in almost all graph applications, but hierarchy and ports are also very commonly found features. Many participants expect an improved readability of diagrams from the use of graph layout technology (Question 27). Further aspects that were mentioned often are work efficiency, i. e. reducing the amount of work caused by manual interaction with diagrams, and saving time by automating the creation of layouts.

The lower and upper bounds of typical graph sizes, given in response to Question 23, are extremely scattered, ranging from a few nodes to tens of thousands of nodes, hence the average values are not very meaningful in this case. The median values, in contrast, are less biased by these extremes; according to these, typical graphs have between 10 and 50 nodes. This result supports the design decisions made for the KIELER layout algorithms (Section 4.4.1), where code maintainability has been favored over execution time performance. The execution time measurements in Section 2.2.4

⁸<http://www.informatik.uni-kiel.de/en/rtsys/publications/>

⁹<http://rtsys.informatik.uni-kiel.de/confluence/display/KIELER/>

showed that graphs smaller than 100 nodes and 400 edges are processed in less than 0.1 seconds by K Lay Layered.

It is interesting to see that in response to Question 25 port constraints were the most mentioned type of domain-specific constraint. A possible explanation for this is that K Lay Layered, the most advanced layout algorithm of KIELER, has always had a strong focus on data flow diagrams [SFvHM10, KSSvH12, SSvH14], hence many survey participants probably got in contact with KIELER through their need for automatic layout of this kind of diagrams.

Quality evaluation. The generic layout integration has been rated very differently for GMF and Graphiti (Question 6): the average rating was 2.08 for GMF users, which corresponds to the rating *good*, and 0.5 for Graphiti users, i. e. between *poor* and *fair*. The reason for this difference in quality is explained in Section 4.3. While for GMF the graph structure can be reliably detected using classes of edit parts, the Graphiti integration is focused on the Pictogram model, often requiring adaptations in order to derive the graph structure for a particular Graphiti-based editor.

According to the responses to Questions 8, 9, and 10, addressing the quality of layout integration interfaces, the Eclipse-based integration and the class library are very flexible and efficient, with average ratings consistently above 2.2 (between *good* and *excellent*). The simplicity, in contrast, was given moderate ratings, which is another indicator for the need for better documentation. Interestingly, responses for the layout web service resulted in an inverse situation, where simplicity has a higher rating than flexibility and efficiency. This might be a hint that the web-based layout approach is especially suited for rapid prototyping, but a more direct integration should be used in later stages of development. A further observation is that the Eclipse integration was rated consistently better than the class library, which is not surprising given that most of the work on KIELER was done in the context of the Eclipse platform. The free text responses given for Question 11 contain very positive feedback for the integration interfaces. A common criticism addressed the too prominent user interface contributions of Eclipse plug-ins shipped with KIELER. In reaction to this, the user interface was completely redesigned with the goal of a more subtle

6. Survey

integration, allowing to more flexibly combine KIELER with other projects to create rich client applications (RCAs).

The layout configuration interfaces mostly received ratings around *good* (Questions 19 and 20) with respect to their flexibility. However, similarly to the integration interfaces, the simplicity was rated consistently lower than the flexibility. Obviously, more effort to help users to understand the layout configuration concepts would be appropriate. This consequence is also backed by the responses to the open-ended Question 21, where many participants expressed their need for more or better documentation.

The interfaces and tools for layout algorithm development have been given diverse ratings (Questions 13 and 14). The KGraph data structure (Section 3.1.1), the properties mechanism (Section 4.1.1), and the general structure of KLayout algorithms (Section 4.4.1) have very good overall results, while the KIELER Editor for Graphs (KEG) and the graph analysis framework (Section 4.5) have mediocre results. An explanation for this is that the first three interfaces are part of the public API of KIELER, and as such they have been repeatedly improved and extended. The graph editor and graph analysis tools, in contrast, are merely used internally to support the development of the actual algorithms, hence these tools have only been improved when it seemed necessary to one of the developers in the KIELER team.

Of all layout algorithms that are currently available through KIELER, including Graphviz and OGDF algorithms, KLayout Layered has the best average rating of 2.3 (Question 16). This is probably due to the special focus on port constraints and hyperedges (see Chapter 2), which are important requirements for the layout of data flow diagrams. The other layer-based algorithms also have good average ratings, with the exception of the Draw2D Layout, for which 69% of the persons chose the rating *poor*, leading to the worst result of all considered algorithms. It is worth to note that this is the default algorithm integrated in GMF, hence for many users of Eclipse-based modeling tools it is the only available option for automatic layout. The other layout types received only mediocre ratings; in particular, the force-based algorithms (average 1.2) and the circular algorithms (average 1.1) are mostly graded close to *fair*. The majority of participants seems to prefer algorithms that emphasize the direction of edges (e. g. layer-based)

Table 6.1. Comparison of four layout types according to the average rating of the respective layout algorithms in response to Question 16 of the survey and the frequencies with which they have been selected by participants of the evolutionary meta layout experiment (Section 3.2.3). The ratings are on a scale from 0 (worst) to 3 (best). The selection frequencies are given separately for the evolutionary and the manual configuration approaches. The same ranking of layout types can be derived from all three results. The average rating of layer-based algorithms is 2.22 if the Draw2D layout algorithm is excluded.

Layout Type	Rating	Evol. Selection	Manual Selection
Layer-based	1.95	45%	63%
Planarization-based	1.76	39%	25%
Force-based	1.23	12%	5%
Circular	1.10	2%	5%

over algorithms for undirected graphs (e. g. force-based or circular). This tendency is also reflected by the responses to Question 17, where KLayout Layered and Dot have been reported as the most used layout algorithms. Furthermore, the ranking of layout types that results from their average rating is exactly the same as the ranking that results from the user study reported in Section 3.2.3 regarding how frequently algorithms of these layout types have been selected by the participants of the study. These two results are compared in Table 6.1.

The free text feedback on KLayout Layered reported in Question 18 is largely very positive, but there is some criticism regarding the handling of labels, unnecessary bend points and crossings of edges, unnecessary whitespace, and the documentation of layout options. The handling of labels and the documentation have been improved since then, but the other criteria remain open topics for research.

When asked about advantages of other graph layout tools compared to KIELER, participants have often named simplicity and good documentation (Question 29). These have already been identified as weak spots of KIELER considering responses to several other questions. The command line usage of Graphviz has been named as another advantage, allowing very simple interfacing and the integration with command line scripts. This may be a

6. Survey

motivation to promote the web-service-based command line tool mentioned in Section 5.2, which is available for download on the KIELER web page, but is probably still unknown to most users. Commonly named disadvantages (Question 30) were the poor extensibility, lacking features such as port constraints, and the high costs, which of course applies only to commercial products such as yFiles.

Conclusion

The contributions of this thesis are on three separate levels: layout algorithms, abstract layout interfaces supporting automatic configuration, and integration in modeling applications. The results are summarized in the following.

7.1 Summary of Results

Layout algorithms. There are several approaches for the automatic layout of graphs, but here we have focused on the layer-based approach because it emphasizes the directions of edges and has proven very suitable for domain-specific extensions. The main extension we have considered is towards port constraints, which need to be handled mainly during the crossing minimization phase. We have considered two approaches for ranking ports, *layer-total* and *node-relative*, both of which can be employed in the barycenter heuristic for reducing the number of crossings in a layered graph. Furthermore, dummy nodes are used for the routing of edges connected to north/south-side ports and *inverted* ports, e. g. input ports positioned on the east side of a node. This *global* routing approach has been shown to produce significantly fewer edge crossings and bends compared to an earlier local approach. Experiments have been performed using a set of data flow models from the Ptolemy project [EJL⁺03].

A further case that requires adaptations is the presence of hyperedges. While the mapping of hyperedges to sets of normal edges is relatively simple, here it has been shown that the methods for counting crossings yield values that differ substantially from the actual crossings numbers when applied to such hypergraphs. Two specialized counting methods have

7. Conclusion

been proposed, and both perform much better than the standard method in the experiments.

Finally, the aspect of user interaction with the layout algorithm has been discussed, and a *sketch-driven* approach has been presented. This method cannot handle the case of dynamic graphs where new graph elements may be added, expecting the layout algorithms to position the new elements while keeping the remaining elements at their previous positions. The sketch-driven approach has two other use cases instead: reacting to manual changes to the layout of a graph, and computing a layer-based drawing from another kind of layout, e. g. force-based or planarization-based layout. The latter can be a useful alternative to the usual layer assignment and node ordering methods.

Layout configuration. We differentiate between *concrete layout*, that is the positioning information of graph elements computed as output of layout algorithms, and *abstract layout*, that is the selection and configuration of layout algorithms. *Meta layout* denotes a process of generating an abstract layout, and an implementation of such a process is called a *layout configurator*. The KGraph meta model allows to capture the structure of graphs as well as their concrete layout and abstract layout, hence it completely specifies both the input and the output of layout algorithms. Layout configurators generate layout option mappings for the elements of a graph, and multiple configurators can be combined based on priorities. The generic handling of layout algorithms and their parameters, also called *layout options*, requires a representation of their meta data, e. g. the name of a layout algorithm or the data type of an option.

Based on this foundation of a generic layout interface, a genetic representation of abstract layouts has been proposed, allowing the application of meta heuristics for optimizing layouts on the configuration level. The general scheme is to create several abstract layouts using a meta heuristic and then to evaluate them by executing the encoded layout algorithms and applying metrics to the resulting concrete layouts. The metrics must be able to give meaningful hints on the quality of the layouts; five proposals for measuring different aesthetic criteria are presented here. Furthermore, an evolutionary algorithm is described as an example of a meta heuristic for

optimizing layout configurations. The search for suitable individuals in the solution space can be guided by users by modifying the relative weights of the layout metrics either directly or indirectly through selection of preferred layouts. The effectiveness of this approach has been evaluated with a user study by comparing the new approach with manual configuration through a table of layout options. Though most objective results of this comparison are not statistically significant, the participants reported a high satisfaction with the evolutionary configuration method, especially regarding the indirect adjustment of metric weights.

Integration. Eclipse has been chosen as the implementation platform for the KIELER project because it offers numerous tools supporting model-driven engineering with applications both in academics and in the industry. The layout algorithms and layout configuration concepts described in this thesis have been implemented in Java and integrated into Eclipse as part of KIELER. The proposed API allows to specify the meta data of layout algorithms and layout options through Eclipse *extension points*. Layout configurations can be specified on different levels: default parameter values of layout algorithms, application-specific configurations provided through an extension point, dynamic configurations evaluated at run-time, and settings made by the user. The integration supports multiple diagram viewers (*front-ends*) and layout algorithms (*back-ends*). Generic integrations have been implemented for GMF and Graphiti, two very frequently used diagramming frameworks, as well as for KLighD, a tool for generating transient views that is part of KIELER. Several example applications employing these integrations have been presented. On the back-end side, the graph layout libraries Graphviz and OGDF have been connected, providing a total of 27 layout algorithms. These are complemented by a set of Java-based implementations of layout algorithms, which include the extensions of the layer-based approach discussed in the first part of this thesis.

Two options have been presented for the integration of graph layout into applications that are not based on Eclipse. For applications using Java the layout algorithms can be accessed as class libraries. The graphical editor of the Ptolemy project, for instance, has been extended so it offers a menu entry for automatic layout of data flow diagrams. If a Java library is not

7. Conclusion

applicable, a further alternative is to employ a web service by sending and receiving graph instances in serialized form. This allows for platform independent and light-weight integrations, which is particularly suited for prototype development.

Evaluation. A web-based survey has been conducted in order to evaluate the KIELER layout infrastructure. The survey addressed several topics, including application requirements, quality of layout algorithms, and quality of interfaces for users and application programmers. The overall result is quite positive: the average of all given quality ratings (excluding ratings of Graphviz and OGDF layout algorithms) is 1.99 on a scale from 0 (worst) to 3 (best), which corresponds almost exactly to the label “good”. Many of the free text responses given by participants confirm the usefulness of KIELER. The most frequent criticism was about missing software documentation, a task that is often treated with lower priority in academic software projects.

It is difficult to measure the success regarding the general goal of this thesis to support the use of automatic graph layout in the context of model-driven engineering. One reason is that the proposed solutions involve different subdisciplines of computer science, namely algorithm engineering and software engineering, with quite different perspectives. Another reason is that the success is ultimately determined by the actual increase of productivity among MDE practitioners, which can hardly be captured directly. However, some hints on the degree of success can be found, e. g. the high average ratings and positive feedback collected in the survey. The number of users of the KIELER layout infrastructure can be another hint. Although this number is unknown, a lower bound can be estimated: excluding students and members of the development team of KIELER, 37 survey participants stated that they had already used KIELER. Many of these users had been invited to the survey after they contacted me asking for support on the integration of automatic layout in their applications. Some of these applications are mentioned in Section 4.3 and in Chapter 5.

The application EHANDBOOK is developed by ETAS¹ and targets the interactive documentation of large data flow models from the automotive

¹<http://www.etas.com/>

industry [FvHK⁺14]. This application relies on a large portion of the results of this thesis and uses them as essential features. The layer-based layout algorithm with the extensions for ports and hyperedges discussed in Chapter 2 is used to generate views on the data flow models. The meta model and configuration concepts discussed in Section 3.1 are used to control the behavior of the layout algorithm, e. g. by dynamically setting different levels of port constraints in order to adapt the view to specific situations. The Eclipse integration presented in Chapter 4 connects the graph layout functionality to the concrete diagram viewer, first using GMF and later KLighD as diagramming frameworks. EHANDBOOK is now offered as a product for calibration processes of electronic control units.

7.2 Lessons Learned

A number of observations could be made during the work on the layout algorithms and the layout infrastructure.

Details matter. Most of the research on graph layout is focused on the optimization of criteria such as the number of edge crossings, the number of edge bends, the total area, etc., which are very important aspects. However, many layout algorithms exhibit flaws that are not covered with these optimization goals, but lead to layouts with obvious room for improvement. For instance, some of the edges in Figure 7.1, which has been done with a layer-based algorithm, are much longer than necessary. Many users would be tempted to manually fine-tune this layout by moving `SampleDelay3` and `Display3` to the left and shortening the outgoing edge of `UpdateC1`. By knowing the background of the employed layout algorithm, one can easily see that the reason for these long edges is that `UpdateC1`, `SampleDelay3`, `Display4`, and the large composite node `UpdateC2` have been assigned to layer 4, while `Display3` is in layer 5 because two adjacent nodes cannot be in the same layer. The width of layer 4 is dominated by `UpdateC2`. This can be explained to users, but still the result is not satisfying. Possible solutions would be to apply a global layer assignment for compound graphs [San96b, For02] or orthogonal compaction [KKM01].

a member of the KIELER development team is that it is hard to find a development process that always fits the needs of an academic software project, since both the focus and the members of the project are likely to change continuously. On the one hand, guidelines and tools for software development are very beneficial, e. g. source code management, code reviews, issue tracking, Wiki-based documentation, and automated builds. When used appropriately, they can substantially support the goal of high-quality software. On the other hand, the involved persons often have duties and interests beyond software development, e. g. writing papers or theses, teaching, and overseeing cooperations and funded projects. Typically these kinds of activities have higher priority than a software project. Therefore it is not possible to apply a rigid development process, but the weekly time spent for the software should depend on the individual situation of each involved person. Having said that, I would highly recommend any computer science student to engage in an academic software project, since the experience gained in this way is definitely valuable for the upcoming career. Furthermore, an open-source software project is an optimal medium for knowledge transfer into industrial practice.

Habits change slowly. Whenever a manual engineering activity is replaced with an automatism, skepticism is a natural reaction. For instance, the advent of compilers generating machine code has initially been resisted by some practitioners in a similar way as the high-level code generation brought by MDE [Sel03]. A skeptical mindset is even reinforced when the first available implementations of automatism provide results of much lower quality than the handmade variants. I believe that automatic graph layout is undergoing the same process of slowly increasing acceptance. One key to accelerating this process is to provide high-quality layouts that are adaptable to the needs of particular applications, a goal which I hope has come more into reach with the contributions of this thesis. Another key is to integrate such a graph layout infrastructure in the modeling environments used in practice and to communicate the capabilities and advantages of automatic layout to the users of these environments. This is largely the responsibility of the tool providers.

The integration of the KIELER layer-based layout algorithm into Ptolemy

7. Conclusion

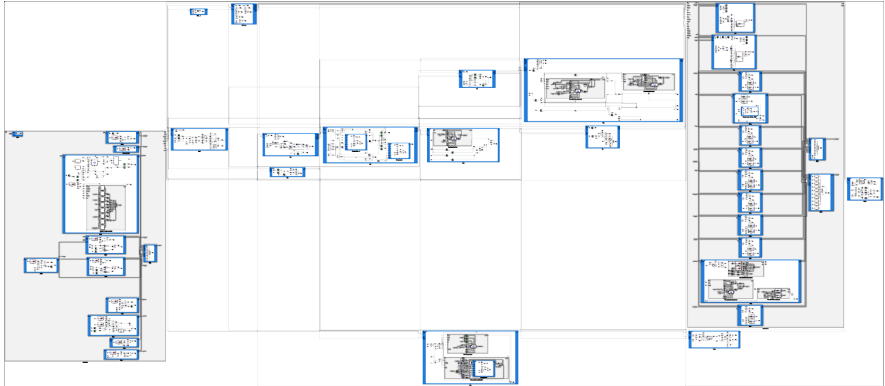


Figure 7.2. A drawing of a large ASCET model where more than 90% of the area is unused.

(see Section 5.1) has already been a success. This integration has been developed in direct cooperation with Edward A. Lee, leader of the Ptolemy project at UC Berkeley. In October 2011, he wrote in an email

“I have to say we have crossed a tipping point. I am now using automatic layout every time I use Vergil. I don’t see how we ever did without it.”

This was twelve years after the first release of Ptolemy II.

7.3 Future Work

Compactness. One of the most severe problems of KLayout Layered, the layer-based algorithm with the extensions for ports and hyperedges discussed in Chapter 2, is *compactness*. This becomes manifest in large graphs, especially where very large nodes are involved, e. g. composite nodes in compound graphs. Figure 7.2 shows a drawing of a data flow model from an industrial application [FvHK⁺14]. When displayed in original scale, less than 1% of the drawing would be visible on a single computer screen. Considering that a huge portion of the drawing remains white, it is evident that more

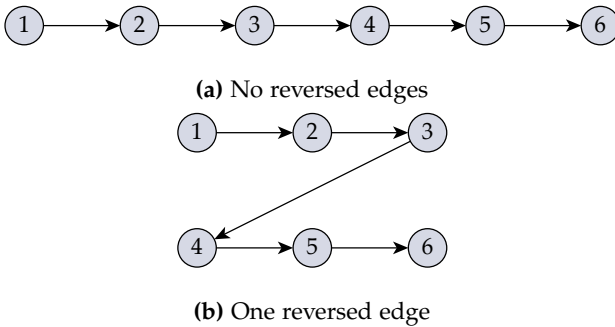


Figure 7.3. Reversing the edge (3,4) leads to a more compact layout in terms of aspect ratio.

compact drawings of this model are possible, but the current standard methods do not provide sufficient means for optimizing the area and the aspect ratio of drawings.

Future research on graph layout methods should have a stronger focus on compactness [GvHM⁺14]. For instance, the established goal of the first phase of the layer-based approach, to find a minimal feedback arc set, is questionable in some situations. The layout in Figure 7.3(a) has no feedback edges, but a very bad aspect ratio of 12.84 (width/height ratio). In contrast, by reversing a single edge the same graph can be drawn as shown in Figure 7.3(b), which results in an aspect ratio of 1.70, a value near to that of wide screens.

The minimal number of layers for an acyclic graph is determined by its longest path, since all nodes of a path have to be assigned to different layers. Hence the longest path limits the freedom of layer assignment algorithms regarding their goal of finding a layering that allows a compact drawing. This fact has been neglected in previous approaches to cycle elimination, which only address the number of reversed edges. A possible approach for further research in this area is to generalize the edge reversal phase such that it targets two optimization goals: minimize the number of reversed edges and minimize the length of the longest path.

Another idea that could be followed is to remove a subset of the edges

7. Conclusion

such that the remaining graph can be drawn more compactly, and to reinsert these edges using a routing algorithm such as the orthogonal routing method of Wybrow et al. [WMS10]. This can be applied to any graph layout approach. For the layer-based approach, edges spanning a large number of layers could be removed in order to reduce the number of dummy nodes, possibly improving the total size. Alternatively, edges on a longest path could be removed in order to reduce the minimal number of layers, possibly improving the aspect ratio. For planarization-based methods, edges violating planarity are removed and then reinserted by replacing edge crossings with dummy nodes. Here a new approach could be to insert such edges at the very end of the algorithm, again reducing the number of dummy nodes and thus allowing more compact drawings.

Port constraints. In Section 2.2 it has been shown how to extend the layer-based approach to consider port constraints. Such an extension has already been proposed for planarization by Gutwenger et al. [GKM07]. The topology-shape-metrics approach, which produces orthogonal drawings of planarized graphs, has been considered for including port constraints by Eiglsperger et al. [EFK00] and Siebenhaller [Sie09]. However, the application of planarization-based methods to data flow diagrams has not been evaluated yet, hence it is unknown how well these methods would work in practice for this important application domain.

Previous work on adding constraints to energy-based layout methods included clustering, alignment, symmetric shape [DFM93], absolute and relative positions [KKR96], and general inequalities on node coordinates [HM98]. Dwyer et al. add either order-preserving constraints or separation constraints for drawing directed graphs [DK05, DKM09]. These results have been applied to data flow diagrams in a recent work [RKD⁺14]. It would be interesting to investigate this new approach in the context of industrial applications and to compare the results with the layer-based approach.

Structure-driven layout configuration. In Section 3.2 an automatic layout configuration approach has been discussed which is based on analysis of graph drawings. A problem with this approach is that it requires to execute

layout algorithms in order to analyze their results. An interesting question is how to determine a suitable configuration just by analyzing the graph structure. Archambault et al. have done this for a limited set of layout algorithms in the context of graph decomposition [AMA07], but important layout algorithms such as layer-based and planarization-based methods have not been considered yet.

Many graph layout approaches exploit specific structural properties of a graph to compute a layout. Graphs that do not meet these requirements have to be transformed, thus corrupting the aesthetic properties of the resulting layout. For instance, given an acyclic graph the layer-based approach arranges it such that all edges point at a specific direction. In contrast, for a cyclic graph at least one edge has to be reversed in order to break the directed cycles, which implies that the chosen edge points at the opposite direction in the final layout. A basic idea that could be realized is to analyze the same structural properties that are regarded in a layout algorithm, thus building on the intrinsic correlation of structural properties and aesthetic properties. For each layout algorithm, a *suitability* value between 0 and 1 could be computed in a similar way as the layout metrics proposed in Section 3.2.1, but referring only to properties of the graph, and not its drawing. By comparing the suitability values of different algorithms, the most suitable algorithm can be determined. For layer-based algorithms the suitability could depend on the number of edges that need to be reversed, while for planarization-based algorithms the number of edges to remove for obtaining a planar subgraph could be considered. The main challenge is that these criteria both depend on NP-hard problems, hence the actual number of edges to reverse or remove depends on the employed heuristics, which may even contain randomized decisions. Another open problem is to find meaningful formulae for the suitability values.

Appendix

A Sketch-Driven Layout Experiment

The experiment for evaluating the responsiveness of sketch-driven layout algorithms, described in Section 2.4.2, consists of 18 tasks that refer to three graphs, which are shown in Figure A.1. The tasks were performed independently of each other in the given order. The goal of the experiment was to evaluate the responsiveness of the sketch-driven approach, i. e. how well the layout matches the user's expectation when it is modified manually and then processed with the sketch-driven algorithm.

Graph 1 (Figure A.1(a))

1. N4 shall be in the last layer.
2. N8 shall be between N2 and N9.
3. N10 shall be above N18.
4. (N18, N3) shall be routed below N9.
5. (N1, N7) shall be routed above (N14, N7).
6. (N5, N12) shall be routed above N7.

Graph 2 (Figure A.1(b))

1. (N14, N21) shall point from left to right.
2. N3 shall be in the same layer as N19.
3. N4 and N9 shall be in separate layers, each with no other node in it.
4. (N12, N3) shall be routed below the other outgoing edges of N12.
5. (N11, N10) shall be reversed.
6. N17 shall be between (N12, N6) and (N12, N19).

A. Sketch-Driven Layout Experiment

Graph 3 (Figure A.1(c))

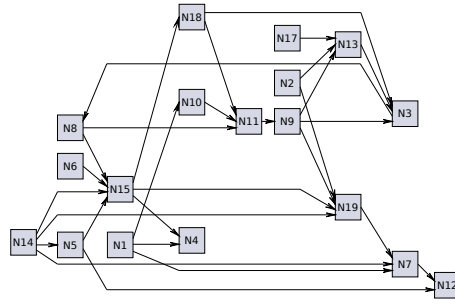
1. (N26, N23) and (N1, N6) shall not cross.
2. (N10, N15) shall have no crossings.
3. (N20, N13) shall be as short as possible.
4. (N3, N24) shall be as long as possible.
5. All outgoing edges of N7 shall point from right to left.
6. The incoming edges of N16 shall have no crossings.

The average results for each task are shown in Table A.1.

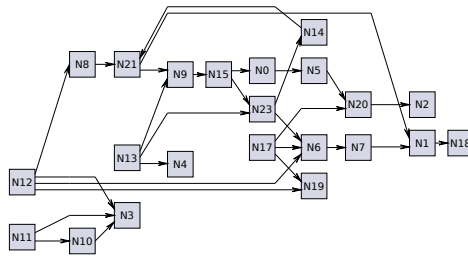
Table A.1. Detailed results for each task: average number of drag-and-drop operations, layout invocations, and layouts with unexpected results.

Task		Drag-and-drop op.	Layout inv.	Unexpected
Graph 1	1	1.00	1.00	0.00
	2	1.50	1.00	0.00
	3	1.13	1.00	0.00
	4	1.38	1.00	0.00
	5	1.75	1.38	0.25
	6	1.25	1.13	0.13
Graph 2	1	1.00	1.00	0.00
	2	1.00	1.00	0.00
	3	2.25	1.63	0.25
	4	4.25	2.00	0.75
	5	1.00	1.00	0.00
	6	1.75	1.25	0.13
Graph 3	1	1.50	1.13	0.00
	2	1.13	1.00	0.00
	3	1.38	1.38	0.25
	4	1.50	1.25	0.00
	5	1.00	1.00	0.00
	6	9.50	3.00	0.13

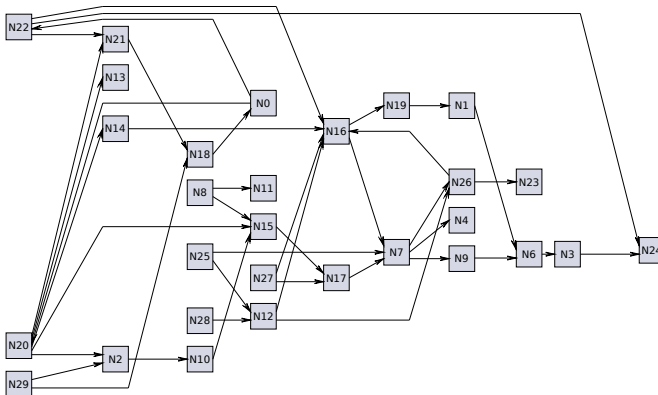
Appendix



(a)



(b)



(c)

Figure A.1. Graphs used for the experiment on the responsiveness of sketch-driven layout.

B Evolutionary Meta Layout Experiment

The experiment for the evaluation of the evolutionary meta layout approach discussed in Section 3.2.3 involved 24 tasks to be performed with eight graphs, which are shown in Figure B.1. Each participant processed four randomly chosen graphs with the evolutionary configuration tool and the remaining four graph with a manual approach. The goal was to capture the error rate and solution time for the tasks and compare these results for the two configuration methods. The tasks are listed in the following.

1. Afcon:
 - ▷ How long is the shortest path from SG to CG?
 - ▷ How long is the shortest path from GH to KE?
 - ▷ Which node has the highest degree?
2. Climate system:
 - ▷ How long is the shortest path from incoming_solar_radiation to melting_permafrost?
 - ▷ How long is the shortest path from saturation of carbon sinks to energy retained?
 - ▷ How many nodes are not reachable from albedo?
3. MySql-history:
 - ▷ Is there a path from MariaDB 5.1 to MySql 5.2?
 - ▷ Is there a path from MySqlCluster 6.2 to MySqlCluster 7.3?
 - ▷ Which node has the highest degree?
4. NZ-Threat:
 - ▷ How long is the shortest path from Native to Gradual decline?
 - ▷ How long is the longest path between any nodes?
 - ▷ How many nodes do not have any successor?

Appendix

5. Presocratic:

- ▷ How many nodes do not have any predecessors?
- ▷ How many nodes do not have any successors?
- ▷ How long is the longest path between any nodes?

6. Salamander-foodweb:

- ▷ Which node does not have any predecessors?
- ▷ Which node has the highest number of predecessors?
- ▷ How long is the shortest path from Photosynthesis plants etc. to Salamander Larval Stage?

7. Sanskrit lexer:

- ▷ How long is the shortest path from absya to licv?
- ▷ How many nodes are not reachable from Accept?
- ▷ How many nodes have exactly two successors?

8. TIME:

- ▷ Which node has the highest degree?
- ▷ How long is the shortest path from Sleep to Birthweight?
- ▷ How many nodes do not have any successors?

The participants were requested to select layouts for the presented graphs using the EVOL or the MANUAL method (see Section 3.2.3). After they selected a layout for a graph, they worked on the corresponding three tasks. Table B.1 shows the average results for each of the eight graphs. The values in the row labeled “Average” differ from the total averages given in Section 3.2.3 because for each graph the ratio of the number of participants who worked with MANUAL and those who worked with EVOL was different. This is due to the random assignment of configuration methods to the graphs.

After all tasks were done, the participants were asked for their opinion on the quality of the evolutionary meta layout method. For each of the

B. Evolutionary Meta Layout Experiment

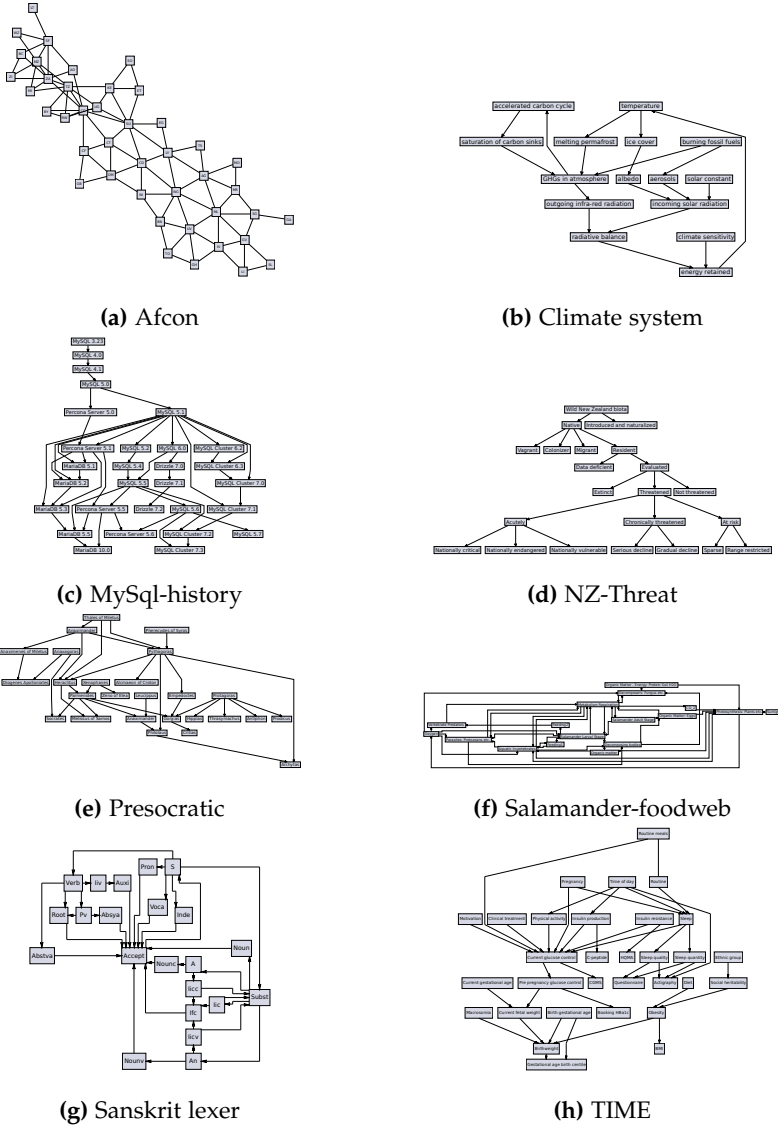


Figure B.1. Graphs used for the experiment on evolutionary meta layout.

Appendix

Table B.1. Rate of correct answers and average time in seconds for working on the tasks for the 21 non-expert participants of the experiment. The variables r_M and t_M represent results with MANUAL, while r_E and t_E represent results with EVOL. None of the differences of mean values are statistically significant.

Graph	Rate of correct answers			Average working time		
	r_M	r_E	$r_E - r_M$	t_M	t_E	$t_E - t_M$
Afcon	61.1%	92.6%	31.5%	226.6	144.2	-82.4
Climate sys.	87.5%	89.7%	2.2%	117.6	129.8	12.2
MySQL-His.	100.0%	100.0%	0.0%	85.0	80.4	-4.6
NZ-Threat	93.3%	87.9%	-5.4%	63.1	71.9	8.8
Presocratic	50.0%	35.6%	-14.4%	161.2	184.2	23.0
Salamander	85.7%	95.2%	9.5%	126.9	116.7	-10.1
Sanskrit lex.	59.0%	66.7%	7.7%	188.8	184.9	-4.0
TIME	82.0%	87.5%	5.5%	131.9	139.6	7.7
<i>Average</i>	77.3%	81.9%	4.6%	137.6	131.5	-6.2

following statements they could choose between the answers *strongly agree*, *agree*, *neutral*, *disagree*, and *strongly disagree*. The results are shown in Figure B.2.

1. The Evolutionary Layout window fails to suggest high quality layout proposals. I would like to see better layout proposals.
2. The Evolutionary Layout window quickly gave me a layout similar to what I expected.
3. I can influence the generated layout proposals effectively by adjusting the criteria sliders.
4. I can influence the generated layout proposals effectively by picking favored layout proposals.
5. The Evolutionary Layout window presents too little variety of interesting layout proposals. I would like to see more different layout proposals.

B. Evolutionary Meta Layout Experiment

6. The Evolutionary Layout window gave me surprising and new ideas about how to arrange diagrams.

The questionnaire included two open-ended questions for obtaining more details on the participants' subjective impression of the evolutionary meta layout approach. When asked what they liked specifically, the most frequent answers were the visual and intuitive approach, the great variety of offered layouts, the immediate previews of the current population, the ability to get to good results quickly, and the feature of selecting favored layouts to indirectly adjust the weights of layout metrics. These answers confirm the usefulness of the multidrawing approach [BMRW98] where users are guided visually through the process of finding good layouts.

The most frequent answers to the question what the participants did not like were the high similarity of some offered layouts, the high number of useless layouts, and the impression that directly setting the weights of layout metrics would be ineffective or yield unexpected results. The issue of too high similarity can be attributed to non-optimal parameters of the evolutionary algorithm, where a higher mutation rate could increase the number of different solutions, and to the distance function presented in Section 3.2.1, which is used to discard layouts that are too similar. That function compares the abstract layouts instead of the concrete layouts, which is much more efficient in terms of computation time, but neglects the fact that similarity does not correlate well between these levels of abstraction. The high number of useless layouts is due to the quality of the layout algorithm libraries included in the experiment. These libraries contain some algorithms that are still in experimental state and fail to produce readable layouts for certain graphs. The perceived lack of effectiveness of the direct manipulation of metric weights can be explained with the relatively high number of displayed metrics. Modifying the weight of one metric only does not affect the overall fitness by much. On the other hand, the automatic adjustment of weights was perceived as much more efficient, although in theory the same effect could be achieved with direct adjustments.

Appendix

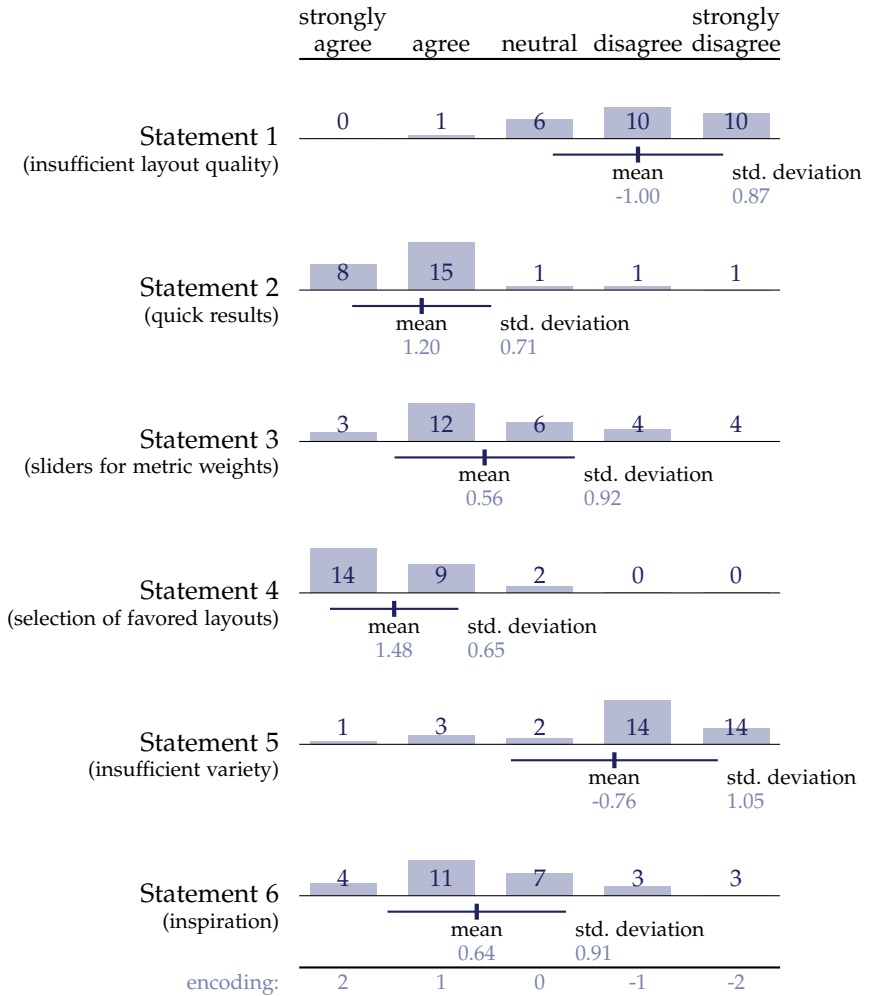


Figure B.2. Frequency of answers for the subjective evaluation of evolutionary meta layout. The colored boxes illustrate the relative frequencies, while the numbers on these boxes give the absolute frequencies. The lines below each row mark the mean value plus/minus the standard deviation. The sample size was 25 for the evaluation of all six statements.

C Survey Questionnaire

The survey consisted of 32 questions, some of which were open-ended, i. e. participants could write their answers in a text box, and others were closed-ended, i. e. participants could choose from a set of proposed answers. The full questionnaire is shown below. The answers proposed for closed-ended questions where multiple answers could be selected are preceded by squares, while circles are used when only one of the proposed answers could be selected. Where no proposed answers are given, the questions are open-ended. For some questions a table of possible answers was offered (see e. g. Table C.1). In these cases one of the five offered ratings could be selected for each row of the table. The gray text below some of the questions was given as explanation in order to help participants to better understand those questions.

The questionnaire is non-linear, i. e. the answers given to some questions determine whether other questions are made visible or not. The rules for these nonlinearities are given in the following

- Question 3 is visible only if *No* is selected in Question 2.
- If *No* is selected in Question 3, the survey is aborted and the participant is marked as *disqualified*.
- If *No* is selected in Question 2 and *Yes* is selected in Question 3, the survey jumps to Question 22 (i. e. all questions between 3 and 22 are skipped).
- Question 6 is visible only if *Yes* is selected in Question 5.
- Questions 8, 9, and 10 are visible only if *Yes* is selected in Question 7.
- Question 11 is visible only if *Yes* is selected in Question 5 or in Question 7.
- Questions 13, 14, and 15 are visible only if *Yes* is selected in Question 12.

Welcome to the KIELER Layout survey! The survey is conducted by the Real-Time and Embedded Systems research group, University of Kiel, Germany. Our goal is to get an idea of your experience with the automatic graph layout technology of the KIELER project, or if you haven't used it yet, your experience with other graph layout tools. We will use this information in our research and publications, of course keeping your identity anonymous. You can learn more about the KIELER project on our website.

1. How did you become aware of the KIELER project?
2. Have you already used any layout technology from the KIELER project?
Also select *yes* if you have programmed layout technology for the KIELER project or integrated KIELER layout in another application.
 Yes No
3. Have you used or would you like to use graph layout technology in general?
 Yes No

Modeling Application

4. How have you been using KIELER layout?
Select all environments in which you have used, integrated, or developed KIELER layout technology. The KEG, KAOM, SyncCharts, Papyrus, and Yakindu editors are GMF-based.
 In a GMF-based editor
 In a Graphiti-based editor
 In a GEF-based editor (without GMF or Graphiti)
 In a Zest-based view
 In a KLighD view
 In another Eclipse-based editor or view (without GEF, Zest, or KLighD)
 In an Eclipse-independent application using a class library
 In an Eclipse-independent application using the provided `kwebs` command line tool for web service access

C. Survey Questionnaire

- In an Eclipse-independent application using the WSDL interface (*Web Services Description Language*)

Integration in Applications

5. Have you used the generic KIELER Layout integrations for GMF or Graphiti?

Select *yes* if you have installed and used the *KIELER Layout for GMF* or the *KIELER Layout for Graphiti* feature from our Eclipse update site or from our Git repository. However, if you have used KIELER Layout only with the editors and views provided by the KIELER project (SyncCharts, KAOM, KEG, KLightD), select *no*.

Yes No

6. How well does the generic GMF / Graphiti layout integration work for your editor?

This refers to the *KIELER Layout for GMF / Graphiti* Eclipse features and the (non-KIELER) diagram editor or viewer for which you have installed the layout feature. If you have used KIELER Layout with multiple editors, give an overall rating and explain it in the text box of Question 11.

excellent good fair poor did not work

7. Have you been involved in integrating KIELER Layout into an application?

Select *yes* only if you have worked on the layout integration for an Eclipse editor or view (e. g. with the `IDiagramLayoutManager` interface), or using the class library for Eclipse-independent applications, or using the KIELER Layout web service (KWebS).

Yes No

8. Please give us your opinion on the quality of KIELER Layout integration interfaces with respect to *flexibility*.

Was the interface suitable for your application? Could the graphs and required layout options be mapped properly?

See Table C.1

9. Please give us your opinion on the quality of KIELER Layout integration interfaces with respect to *efficiency*.

Is the running time of graph transformation and layout computation acceptable?

See Table C.1

Appendix

Table C.1. Options for answering Questions 8, 9, and 10.

	excellent	good	fair	poor	not used
Eclipse integration	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Class library	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Web service	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

10. Please give us your opinion on the quality of KIELER Layout integration interfaces with respect to *simplicity*.

Are the concepts of the interface easy to understand? Could the integration be done without complex adaptations?

See Table C.1

11. Please describe further aspects of your experience with KIELER Layout integration interfaces.

E.g. problems, limitations, benefits, documentation, support, etc.

Implementing Layout Algorithms

12. Have you been involved in developing layout algorithms following the KIELER interface for layout algorithms, or adapting other libraries to that interface?

Yes No

13. Please give us your opinion on the quality of the layout algorithm development interfaces and tools with respect to *flexibility*.

Interfaces: first three rows. Was the interface suitable for your algorithms? Could the graphs and required layout options be mapped properly?

Tools: last two rows. Did the tool offer all features you required during algorithm development?

See Table C.2

C. Survey Questionnaire

Table C.2. Options for answering Questions 13 and 14.

	excellent	good	fair	poor	not used
KGraph structure	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Property mechanism	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
KLay algorithm structure	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Graph analysis	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Graph editor (KEG)	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

14. Please give us your opinion on the quality of the layout algorithm development interfaces and tools with respect to *simplicity*.

Interfaces: first three rows. Are the concepts of the interface easy to understand? Could the algorithm implementation be done without complex adaptations?

Tools: last two rows. Is the tool intuitive and easy to use?

See Table C.2

15. Please describe further aspects of your experience with our layout algorithm development interfaces.

E. g. problems, limitations, benefits, documentation, support, etc.

Quality of Layout Algorithms

16. Please give us your opinion on the overall quality of layouts created by the algorithms you have used so far.

In rows where multiple algorithms are mentioned, rate the algorithm you have used most.

See Table C.3.

17. Which of the layout algorithms shown above have you used most?
18. Please describe with more detail strengths and weaknesses of your most used algorithms.

Appendix

Table C.3. Options for answering Question 16.

	excellent	good	fair	poor	not used
KLay Layered / KLoDD (KIELER)	○	○	○	○	○
Dot (Graphviz)	○	○	○	○	○
Draw2D Layout	○	○	○	○	○
Sugiyama / Upward Planarization (OGDF)	○	○	○	○	○
Planarization (OGDF)	○	○	○	○	○
KLay Force (KIELER)	○	○	○	○	○
Neato / FDP (Graphviz)	○	○	○	○	○
Davidson-Harel / Fruchterman-Reingold / Kamada-Kawai / FMMM / GEM (OGDF)	○	○	○	○	○
Circo (Graphviz)	○	○	○	○	○
Circular (OGDF)	○	○	○	○	○
Twopi (Graphviz)	○	○	○	○	○
Tree / Radial Tree (OGDF)	○	○	○	○	○
Box Layout / Fixed Layout (KIELER)	○	○	○	○	○

Layout Configuration

19. Please give us your opinion on the quality of KIELER layout configuration interfaces with respect to *flexibility*.

Is the interface effective in configuring the automatic layout in the way you need it? The first two rows are user interfaces, the remaining rows are programmer interfaces.

See Table C.4

20. Please give us your opinion on the quality of KIELER layout configuration interfaces with respect to *simplicity*.

Are the concepts of the interface easy to understand? The first two rows are user interfaces, the remaining rows are programmer interfaces.

See Table C.4

C. Survey Questionnaire

Table C.4. Options for answering Questions 19 and 20. The extension point elements have been renamed after the survey was conducted; here the new names are given (cf. Section 4.1.2).

	excellent	good	fair	poor	not used
Layout view	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Preference page	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
View management / Java interface	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
staticConfig element of layoutConfigs extension point	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
semanticConfig element of layoutConfigs extension point	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
customConfig element of layoutConfigs extension point	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Class library with KGraph data structure	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Web service with global option declaration	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>
Web service with annotation of graph elements	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>	<input type="radio"/>

21. Please describe further aspects of your experience with KIELER layout configuration interfaces.

E. g. problems, limitations, benefits, documentation, support, etc.

Graph Types and Properties

22. What kind of graph-based models do you work with?

E. g. Statecharts, class diagrams, metabolic networks, etc.

23. What are typical sizes for your graphs?

E. g. 20–50 nodes

Appendix

24. Which of the following are typical features of your graphs?

Select all that apply.

- Node labels
- Edge labels
- Multiedges (multiple edges between two nodes)
- Self-loops (edges with the same node as source and target)
- Ports (specific points of a node that edges are attached to)
- Hierarchy / clusters (compound nodes containing a subgraph)
- Edges that connect nodes from different hierarchy levels / clusters
- Hyperedges (connection of multiple nodes)
- Other

25. Which domain-specific constraints are required for the layout of your graphs?

E. g. constraints on the positioning of ports, orientation of edges, etc.

26. Which requirements and aesthetic criteria would you add as your personal preference?

Modeling With Automatic Layout

27. Which benefits do you expect from automatic graph layout technology?

28. Which graph layout tools (other than KIELER) have you already used?

29. If applicable, please state good features and advantages of these other layout tools.

30. If applicable, please state problems and deficiencies of these other layout tools.

Final Comments

31. Any further comments you wish to make on KIELER, automatic layout in general, or this survey?
32. *Optional:* Give your contact details for further scientific cooperation.

Thank you for taking our survey. Your contribution is a great help for our research.

Bibliography

- [AAB⁺12] David Auber, Daniel Archambault, R. Bourqui, A. Lambert, M. Mathiaut, P. Mary, M. Delest, J. Dubois, and G. Melançon. The Tulip 3 Framework: A scalable software library for information visualization applications based on relational data. Technical Report 7860, Project-Team GRAVITE, Inria, January 2012.
- [AHN07] Radoslav Andreev, Patrick Healy, and Nikola S. Nikolov. Applying ant colony optimization metaheuristic to the DAG layering problem. In *Proceedings of the Parallel and Distributed Processing Symposium (IPDPS'07)*, pages 1–9, 2007. doi:10.1109/IPDPS.2007.370426.
- [AMA07] Daniel Archambault, Tamara Munzner, and David Auber. Topolayout: Multilevel graph layout by topological features. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):305–317, 2007.
- [And03] Charles André. Semantics of SyncCharts. Technical Report ISRN I3S/RR–2003–24–FR, I3S Laboratory, Sophia-Antipolis, France, April 2003.
- [Bab02] Danil E. Baburin. Using graph based representations in reengineering. *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering*, pages 203–206, 2002. doi:10.1109/CSMR.2002.995805.
- [BB00] André M. S. Barreto and Helio J. C. Barbosa. Graph layout using a genetic algorithm. In *Proceedings of the Sixth Brazilian Symposium on Neural Networks*, pages 179–184, 2000. doi:10.1109/SBRN.2000.889735.

Bibliography

- [BB01] Helio J. C. Barbosa and André M. S. Barreto. An interactive genetic algorithm with co-evolution of weights for multiobjective problems. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO'01)*, pages 203–210, 2001.
- [BB05] Michael Baur and Ulrik Brandes. Crossing reduction in circular layouts. In *Revised Papers of the 30th International Workshop on Graph-Theoretic Concepts in Computer Science*, volume 3353 of *LNCS*, pages 332–343. Springer, 2005. doi:10.1007/978-3-540-30559-0.
- [BBE⁺11] Christian Bachmaier, Franz J. Brandenburg, Philip Effinger, Carsten Gutwenger, Jyrki Katajainen, Karsten Klein, Miro Spönemann, Matthias Stegmaier, and Michael Wybrow. The Open Graph Archive: A community-driven effort. Poster at the 19th International Symposium on Graph Drawing, Technische Universiteit Eindhoven, Netherlands, September 2011.
- [BBS96] Jürgen Branke, Frank Bucher, and Hartmut Schmeck. Using genetic algorithms for drawing undirected graphs. In *Proceedings of the Third Nordic Workshop on Genetic Algorithms and their Applications*, pages 193–206, 1996.
- [BBS04] Robert Ian Bull, Casey Best, and Margaret-Anne Storey. Advanced widgets for Eclipse. In *Proceedings of the 2004 OOPSLA Workshop on Eclipse Technology Exchange*, pages 6–11. ACM, 2004. doi:10.1145/1066129.1066131.
- [BDD⁺00] Stina S. Bridgeman, Giuseppe Di Battista, Walter Didimo, Giuseppe Liotta, Roberto Tamassia, and Luca Vismara. Turn-regularity and optimal area drawings of orthogonal representations. *Computational Geometry*, 16(1):53–93, 2000.
- [BDL95] Paola Bertolazzi, Giuseppe Di Battista, and Giuseppe Liotta. Parametric graph drawing. *IEEE Transactions on Software Engineering*, 21(8):662–673, August 1995. doi:10.1109/32.403790.
- [BEKW02] Ulrik Brandes, Markus Eiglsperger, Michael Kaufmann, and Dorothea Wagner. Sketch-driven orthogonal graph drawing.

- In *Proceedings of the 10th International Symposium on Graph Drawing (GD'02)*, volume 2528 of LNCS, pages 1–11. Springer, 2002. doi:10.1007/3-540-36151-0.
- [BELP13] Ulrik Brandes, Markus Eiglsperger, Jürgen Lerner, and Christian Pich. Graph Markup Language (GraphML). In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, pages 517–541. CRC Press, 2013.
- [BF05] Robert Ian Bull and Jean-Marie Favre. Visualization in the context of model driven engineering. In *Proceedings of the International Workshop on Model Driven Development of Advanced User Interfaces (MDDAUI'05)*, volume 159. CEUR Workshop Proceedings, 2005.
- [BGKM11] Gereon Bartel, Carsten Gutwenger, Karsten Klein, and Petra Mutzel. An experimental evaluation of multilevel layout methods. In *Proceedings of the 18th International Symposium on Graph Drawing (GD'10)*, volume 6502 of LNCS, pages 80–91. Springer, 2011. doi:10.1007/978-3-642-18469-7.
- [BGM04] Benjamin B. Bederson, Jesse Grosjean, and Jon Meyer. Toolkit design for interactive structured graphics. *IEEE Transactions on Software Engineering*, 30(8):535–546, August 2004.
- [BGT99] Stina Bridgeman, Ashim Garg, and Roberto Tamassia. A graph drawing and translation service on the World Wide Web. *International Journal of Computational Geometry & Applications*, 9(04n05):419–446, 1999. doi:10.1142/S021819599900025X.
- [BH11] Franz J. Brandenburg and Kathrin Hanauer. Sorting heuristics for the feedback arc set problem. Technical Report MIP-1104, Department of Informatics and Mathematics, University of Passau, February 2011.
- [BJL01] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. A fast layout algorithm for k -level graphs. In *Proceedings of the 8th International Symposium on Graph Drawing (GD'00)*,

Bibliography

- volume 1984 of *LNCS*, pages 229–240. Springer, 2001. doi:
10.1007/3-540-44541-2.
- [BK02] Ulrik Brandes and Boris Köpf. Fast and simple horizontal coordinate assignment. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, volume 2265 of *LNCS*, pages 33–36. Springer, 2002. doi:10.1007/3-540-45848-4.
- [BLA13] Don Batory, Eric Latimer, and Maider Azanza. Teaching model driven engineering from a relational database perspective. In *Proceedings of the 16th International Conference on Model-Driven Engineering Languages and Systems (MODELS'13)*, volume 8107 of *LNCS*, pages 121–137. Springer, 2013. doi:10.1007/978-3-642-41533-3.
- [BLL⁺08] Christopher Brooks, Edward A. Lee, Xiaojun Liu, Stephen Neuendorffer, Yang Zhao, and Haiyang Zheng. Heterogeneous concurrent modeling and design in Java, volume 2: Ptolemy II software architecture. Technical Report UCB/EECS-2008-29, EECS Department, University of California, Berkeley, April 2008. URL: <http://www.eecs.berkeley.edu/Pubs/TechRpts/2008/EECS-2008-29.html>.
- [BM01] Oliver Bastert and Christian Matuszewski. Layered drawings of digraphs. In Michael Kaufmann and Dorothea Wagner, editors, *Drawing Graphs: Methods and Models*, volume 2025 of *LNCS*, pages 87–120. Springer, 2001. doi:10.1007/3-540-44969-8.
- [BM04] John Boyer and Wendy Myrvold. On the cutting edge: Simplified $\mathcal{O}(n)$ planarity by edge addition. *Journal of Graph Algorithms and Applications*, 8(3):241–273, 2004.
- [BMJ04] Wilhelm Barth, Petra Mutzel, and Michael Jünger. Simple and efficient bilayer cross counting. *Journal of Graph Algorithms and Applications*, 8(2):179–194, 2004.
- [BMRW98] Therese Biedl, Joe Marks, Kathy Ryall, and Sue Whitesides. Graph multidrawing: Finding nice drawings without defin-

- ing nice. In *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*, volume 1547 of *LNCS*, pages 347–355. Springer, 1998. doi:10.1007/3-540-37623-2.
- [Bol03] Azad Bolour. Notes on the Eclipse plug-in architecture. *Eclipse Corner Articles*, July 2003. www.eclipse.org/articles.
- [BP90] Karl-Friedrich Böhringer and Frances Newbery Paulisch. Using constraints to achieve stability in automatic graph layout algorithms. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 43–51, New York, 1990. ACM. doi:10.1145/97243.97250.
- [Bra01] Jürgen Branke. Dynamic graph drawing. In Michael Kaufmann and Dorothea Wagner, editors, *Drawing Graphs: Methods and Models*, volume 2025 of *LNCS*. Springer, 2001.
- [BRSG07] Chris Bennett, Jody Ryall, Leo Spalteholz, and Amy Gooch. The aesthetics of graph visualization. In *Proceedings of the International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging (CAe'07)*, pages 57–64. Eurographics Association, 2007.
- [BS90] Bonnie Berger and Peter W. Shor. Approximation algorithms for the maximum acyclic subgraph problem. In *Proceedings of the first annual ACM-SIAM symposium on Discrete algorithms (SODA'90)*, pages 236–243. SIAM, 1990.
- [BSLF06] Robert Ian Bull, Margaret-Anne Storey, Marin Litoiu, and Jean-Marie Favre. An architecture to support model driven software visualization. In *Proceedings of the 14th IEEE International Conference on Program Comprehension (ICPC'06)*, pages 100–106. IEEE, 2006.
- [BT00] Stina Bridgeman and Roberto Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. *Journal of Graph Algorithms and Applications*, 4(3):47–74, 2000.

Bibliography

- [BT04] Stina Bridgeman and Roberto Tamassia. GDS – a graph drawing server on the internet. In Michael Jünger and Petra Mutzel, editors, *Graph Drawing Software*, pages 193–213. Springer, 2004. doi:10.1007/978-3-642-18638-7.
- [Car12] John Julian Carstens. Node and label placement in a layered layout algorithm. Master’s thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2012.
- [CG07] Markus Chimani and Carsten Gutwenger. Algorithms for the hypergraph and the minor crossing number problems. In *18th International Symposium on Algorithms and Computation (ISAAC’07)*, volume 4835 of LNCS, pages 184–195. Springer, 2007.
- [CGJ⁺13] Markus Chimani, Carsten Gutwenger, Michael Jünger, Gunnar W. Klau, Karsten Klein, and Petra Mutzel. The Open Graph Drawing Framework (OGDF). In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, pages 543–569. CRC Press, 2013.
- [CGM⁺11] Markus Chimani, Carsten Gutwenger, Petra Mutzel, Miro Spönemann, and Hoi-Ming Wong. Crossing minimization and layouts of directed hypergraphs with port constraints. In *Proceedings of the 18th International Symposium on Graph Drawing (GD’10)*, volume 6502 of LNCS, pages 141–152. Springer, 2011. doi:10.1007/978-3-642-18469-7.
- [CGMW10] Markus Chimani, Carsten Gutwenger, Petra Mutzel, and Hoi-Ming Wong. Layer-free upward crossing minimization. *Journal of Experimental Algorithmics*, 15(2.2):1–27, March 2010. doi:10.1145/1671973.1671975.
- [CGMW11] Markus Chimani, Carsten Gutwenger, Petra Mutzel, and Hoi-Ming Wong. Upward planarization layout. *Journal of Graph Algorithms and Applications*, 15(1):127–155, February 2011. doi:10.7155/jgaa.00220.

- [CHJM12] Markus Chimani, Philipp Hungerländer, Michael Jünger, and Petra Mutzel. An SDP approach to multi-level crossing minimization. *Journal of Experimental Algorithmics*, 17(3.3):1–26, September 2012. doi:10.1145/2133803.2330084.
- [Cla10] Ole Claußen. Implementing an algorithm for orthogonal graph layout. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2010.
- [CMT02] Rodolfo Castelló, Rym Mili, and Ioannis G. Tollis. A framework for the static and interactive visualization of statecharts. *Journal of Graph Algorithms and Applications*, 6(3):313–351, 2002.
- [CMT04] Rodolfo Castelló, Rym Mili, and Ioannis G. Tollis. ViSta – visualizing statecharts. In Michael Jünger and Petra Mutzel, editors, *Graph Drawing Software*, pages 299–320. Springer, 2004. doi:10.1007/978-3-642-18638-7.
- [CON85] Norishige Chiba, Kazunori Onoguchi, and Takao Nishizeki. Drawing plane graphs nicely. *Acta Informatica*, 22(2):187–201, 1985. doi:10.1007/BF00264230.
- [DETT99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.
- [DF03] Camil Demetrescu and Irene Finocchi. Combinatorial algorithms for feedback problems in directed graphs. *Information Processing Letters*, 86(3):129–136, 2003. doi:10.1016/S0020-0190(02)00491-X.
- [DFM93] Ed Dengler, Mark Friedell, and Joe Marks. Constraint-driven diagram layout. In *Proceedings of the IEEE Symposium on Visual Languages*, pages 330–335, 1993. doi:10.1109/VL.1993.269619.
- [DGC⁺05] Ugur Dogrusoz, Erhan Giral, Ahmet Cetintas, Ali Civril, and Emek Demir. A compound graph layout algorithm for biological pathways. In *Revised Selected Papers of the 12th International*

Bibliography

- Symposium on Graph Drawing (GD'04)*, volume 3383 of LNCS, pages 442–447. Springer, 2005. doi:10.1007/978-3-540-31843-9.
- [DGL⁺97] Giuseppe Di Battista, Ashim Garg, Giuseppe Liotta, Armando Parise, Roberto Tamassia, Emanuele Tassinari, Francesco Vargiu, and Luca Vismara. Drawing directed acyclic graphs: An experimental study. In *Proceedings of the Symposium on Graph Drawing (GD'96)*, volume 1190 of LNCS, pages 76–91. Springer, 1997.
- [DK05] Tim Dwyer and Yehuda Koren. Dig-CoLa: directed graph layout through constrained energy minimization. In *Proceedings of the IEEE Symposium on Information Visualization (INFOVIS'05)*, pages 65–72, October 2005. doi:10.1109/INFVIS.2005.1532130.
- [DKKM11] Camil Demetrescu, Michael Kaufmann, Stephen Kobourov, and Petra Mutzel. Graph Drawing with Algorithm Engineering Methods (Dagstuhl Seminar 11191). *Dagstuhl Reports*, 1(5):47–60, 2011. doi:http://dx.doi.org/10.4230/DagRep.1.5.47.
- [DKM09] Tim Dwyer, Yehuda Koren, and Kim Marriott. Constrained graph layout by stress majorization and gradient projection. *Discrete Mathematics*, 309(7):1895–1908, 2009. doi:10.1016/j.disc.2007.12.103.
- [DLV95] Giuseppe Di Battista, Giuseppe Liotta, and Francesco Vargiu. Diagram server. *Journal of Visual Languages & Computing*, 6(3):275–298, 1995. doi:10.1006/jvlc.1995.1016.
- [dLV02] Juan de Lara and Hans Vangheluwe. AToM3: A tool for multi-formalism and meta-modelling. In *Proceedings of the 5th International Conference on Fundamental Approaches to Software Engineering (FASE'02)*, volume 2306 of LNCS, pages 174–188. Springer, 2002. doi:10.1007/3-540-45923-5.
- [dMNE93] C. F. Xavier de Mendonça Neto and Peter D. Eades. Learning aesthetics for visualization. In *Anais do XX Seminário Integrado de Software e Hardware*, pages 76–88, 1993.

- [DMW09] Tim Dwyer, Kim Marriott, and Michael Wybrow. Dunnart: A constraint-based network diagram authoring tool. In *Revised Papers of the 16th International Symposium on Graph Drawing (GD'08)*, volume 5417 of *LNCS*, pages 420–431. Springer, 2009. doi:10.1007/978-3-642-00219-9.
- [dNE02] Hugo A. D. do Nascimento and Peter D. Eades. User hints for directed graph drawing. In *Revised Papers of the 9th International Symposium on Graph Drawing*, volume 2265 of *LNCS*, pages 205–219. Springer, 2002. doi:10.1007/3-540-45848-4.
- [Döh10] Philipp Döhning. Algorithmen zur layerzuweisung. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2010.
- [dRB06] Jim des Rivieres and Wayne Beaton. Eclipse platform technical overview. *Eclipse Corner Articles*, April 2006. www.eclipse.org/articles.
- [DS09] Cody Dunne and Ben Shneiderman. Improving graph drawing readability by incorporating readability metrics: A software tool for network analysts. Technical Report HCIL-2009-13, University of Maryland, 2009.
- [DTB98] Don A. Dillman, Robert D. Tortora, and Dennis Bowker. Principles for constructing web surveys. Technical report, Social & Economic Sciences Research Center, Washington State University, 1998.
- [Dub06] Denis Dubé. Graph layout for domain-specific modeling. Master's thesis, School of Computer Science, McGill University, Montréal, 2006.
- [Dud11] Björn Duderstadt. Evolutionary meta layout for KIELER. Student research project, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, May 2011.
- [Ead84] Peter Eades. A heuristic for graph drawing. *Congressus Numerantium*, 42:149–160, 1984.

Bibliography

- [EEHT05] Karsten Ehrig, Claudia Ermel, Stefan Hänsgen, and Gabriele Taentzer. Generation of visual editors as eclipse plug-ins. In *Proceedings of the 20th IEEE/ACM International Conference on Automated Software Engineering (ASE'05)*, pages 134–143. ACM, 2005. doi:10.1145/1101908.1101930.
- [EFK00] Markus Eiglsperger, Ulrich Fößmeier, and Michael Kaufmann. Orthogonal graph drawing with constraints. In *Proceedings of the Eleventh Annual ACM-SIAM Symposium on Discrete Algorithms (SODA'00)*, pages 3–11. SIAM, 2000.
- [EGB03] Thomas Eschbach, Wolfgang Guenther, and Bernd Becker. Crossing reduction for orthogonal circuit visualization. In *Proceedings of the 2003 International Conference on VLSI*, pages 107–113. CSREA Press, 2003.
- [EGB06] Thomas Eschbach, Wolfgang Guenther, and Bernd Becker. Orthogonal hypergraph drawing for improved visibility. *Journal of Graph Algorithms and Applications*, 10(2):141–157, 2006.
- [EGK⁺04a] Markus Eiglsperger, Carsten Gutwenger, Michael Kaufmann, Joachim Kupke, Michael Jünger, Sebastian Leipert, Karsten Klein, Petra Mutzel, and Martin Siebenhaller. Automatic layout of UML class diagrams in orthogonal style. *Information Visualization*, 3(3):189–208, 2004. doi:10.1057/palgrave.ivs.9500078.
- [EGK⁺04b] John Ellson, Emden R. Gansner, Eleftherios Koutsofios, Steven C. North, and Gordon Woodhull. Graphviz and Dynagraph – static and dynamic graph drawing tools. In Michael Jünger and Petra Mutzel, editors, *Graph Drawing Software*, pages 127–148. Springer, 2004. doi:10.1007/978-3-642-18638-7.
- [EJL⁺03] Johan Eker, Jörn W. Janneck, Edward A. Lee, Jie Liu, Xiaojun Liu, Jozsef Ludvig, Stephen Neuendorffer, Sonia Sachs, and Yuhong Xiong. Taming heterogeneity—the Ptolemy approach. *Proceedings of the IEEE*, 91(1):127–144, Jan 2003. doi:10.1109/JPROC.2002.805829.

- [EK02] Markus Eiglsperger and Michael Kaufmann. Fast compaction for orthogonal drawings with vertices of prescribed size. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, volume 2265 of *LNCS*, pages 124–138. Springer, 2002.
- [ELMS91] Peter Eades, Wei Lai, Kazuo Misue, and Kozo Sugiyama. Preserving the mental map of a diagram. In *Proceedings of the First International Conference on Computational Graphics and Visualization Techniques*, pages 34–43, 1991.
- [ELS93] Peter Eades, Xuemin Lin, and W. F. Smyth. A fast and effective heuristic for the feedback arc set problem. *Information Processing Letters*, 47(6):319–323, 1993. doi:10.1016/0020-0190(93)90079-0.
- [EM01] Timo Eloranta and Erkki Mäkinen. TimGA: A genetic algorithm for drawing undirected graphs. *Divulgaciones Matemáticas*, 9(2):155–170, 2001.
- [ENSS98] Guy Even, Joseph Naor, Baruch Schieber, and Madhu Sudan. Approximating minimum feedback sets and multicuts in directed graphs. *Algorithmica*, 20(2):151–174, 1998.
- [ESK04] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of Sugiyama’s algorithm for layered graph drawing. In *Proceedings of the 12th International Symposium on Graph Drawing (GD'04)*, volume 3383 of *LNCS*, pages 155–166. Springer, 2004.
- [ESK05] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of Sugiyama’s algorithm for layered graph drawing. *Journal of Graph Algorithms and Applications*, 9(3):305–325, 2005.
- [EW86] Peter Eades and Nicholas C. Wormald. The median heuristic for drawing 2-layered networks. Technical Report 69, University of Queensland, Department of Computer Science, 1986.

Bibliography

- [FEK08] Peter Friese, Sven Efftinge, and Jan Köhnlein. Build your own textual DSL with tools from the Eclipse Modeling Project. *Eclipse Corner Articles*, April 2008. www.eclipse.org/articles.
- [Fis25] Ronald Aylmer Fisher. Applications of “Student’s” distribution. *Metron*, 5:90–104, 1925.
- [FK96] Ulrich Fößmeier and Michael Kaufmann. Drawing high degree graphs with low bend numbers. In *Proceedings of the Symposium on Graph Drawing (GD’95)*, volume 1027 of LNCS, pages 254–266. Springer, 1996.
- [For02] Michael Forster. Applying crossing reduction strategies to layered compound graphs. In *Proceedings of the 10th International Symposium on Graph Drawing (GD’02)*, volume 2528 of LNCS, pages 115–132. Springer, 2002. doi:10.1007/3-540-36151-0.
- [For05] Michael Forster. A fast and simple heuristic for constrained two-level crossing reduction. In *Proceedings of the 12th International Symposium on Graph Drawing (GD’04)*, volume 3383 of LNCS, pages 206–216. Springer, 2005.
- [FPP90] H. De Fraysseix, J. Pach, and R. Pollack. How to draw a planar graph on a grid. *Combinatorica*, 10(1):41–51, 1990. doi:10.1007/BF02122694.
- [FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software—Practice & Experience*, 21(11):1129–1164, 1991. doi:http://dx.doi.org/10.1002/spe.4380211102.
- [FR07] Robert France and Bernhard Rumpe. Model-driven development of complex software: A research roadmap. In *Proceedings of Future of Software Engineering (FOSE’07)*, pages 37–54. IEEE, 2007. doi:10.1109/FOSE.2007.14.
- [FSMvH10] Hauke Fuhrmann, Miro Spönemann, Michael Matzen, and Reinhard von Hanxleden. Automatic layout and structure-

- based editing of UML diagrams. In *Proceedings of the 1st Workshop on Model Based Engineering for Embedded Systems Design (M-BED'10)*, Dresden, March 2010.
- [Fuh11] Hauke Fuhrmann. *On the Pragmatics of Graphical Modeling*. Dissertation, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel, 2011.
- [Fuh12] Insa Fuhrmann. Layout of compound graphs. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, February 2012.
- [FvH10a] Hauke Fuhrmann and Reinhard von Hanxleden. On the pragmatics of model-based design. In *Foundations of Computer Software. Future Trends and Techniques for Development—15th Monterey Workshop 2008, Budapest, Hungary, September 24–26, 2008, Revised Selected Papers*, volume 6028 of LNCS, pages 116–140, 2010. doi:10.1007/978-3-642-12566-9.
- [FvH10b] Hauke Fuhrmann and Reinhard von Hanxleden. Taming graphical modeling. In *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS'10)*, volume 6394 of LNCS, pages 196–210. Springer, October 2010. doi:10.1007/978-3-642-16145-2.
- [FvHH14] Florian Fittkau, André van Hoorn, and Wilhelm Hasselbring. Towards a dependability control center for large software landscapes. In *Proceedings of the Tenth European Dependable Computing Conference (EDCC'14)*, pages 58–61. IEEE, May 2014.
- [FvHK⁺14] Patrick Frey, Reinhard von Hanxleden, Christoph Krüger, Ulf Rüegg, Christian Schneider, and Miro Spönemann. Efficient exploration of complex data flow models. In *Proceedings of Modellierung 2014*, Vienna, Austria, March 2014.
- [FWWH13] Florian Fittkau, Jan Waller, Christian Wulf, and Wilhelm Hasselbring. Live trace visualization for comprehending large

Bibliography

- software landscapes: The ExplorViz approach. In *Proceedings of the 1st IEEE International Working Conference on Software Visualization (VISSOFT'13)*, pages 1–4, September 2013. doi:10.1109/VISSOFT.2013.6650536.
- [Gan08] Stéphane Ganassali. The influence of the design of web survey questionnaires on the quality of responses. *Survey Research Methods*, 2(1):21–32, 2008.
- [Gan11] Emden R. Gansner. Notes on practical graph drawing. Invited talk at the Dagstuhl Seminar 11191, May 2011. <http://www.dagstuhl.de/mat/Files/11/11191/11191.GansnerEmden.Slides.pdf>.
- [GB04] Erich Gamma and Kent Beck. *Contributing to Eclipse – Principles, Patterns, and Plug-Ins*. Eclipse Series. Addison-Wesley, 2004.
- [GHHL08] John C. Grundy, John Hosking, Jun Huh, and Karen Na-Liu Li. Marama: An Eclipse meta-toolset for generating multi-view environments. In *Proceedings of the 30th International Conference on Software Engineering (ICSE'08)*, pages 819–822. ACM, 2008. doi:10.1145/1368088.1368210.
- [GHM⁺05] Olivier Gruber, B. J. Hargrave, Jeff McAffer, Pascal Rapicault, and Thomas Watson. The Eclipse 3.0 platform: Adopting OSGi technology. *IBM Systems Journal*, 44(2):289–299, 2005. doi:10.1147/sj.442.0289.
- [GJ79] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co, New York, 1979.
- [GJ83] Michael R. Garey and David S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983. doi:10.1137/0604033.
- [GJK⁺03] Carsten Gutwenger, Michael Jünger, Karsten Klein, Joachim Kupke, Sebastian Leipert, and Petra Mutzel. A new approach

- for visualizing UML class diagrams. In *Proceedings of the ACM Symposium on Software Visualization (SoftVis'03)*, pages 179–188. ACM, 2003. doi:10.1145/774833.774859.
- [GKM07] Carsten Gutwenger, Karsten Klein, and Petra Mutzel. Planarity testing and optimal edge insertion with embedding constraints. In Michael Kaufmann and Dorothea Wagner, editors, *Proceedings of the 14th International Symposium on Graph Drawing (GD'06)*, volume 4372 of LNCS, pages 126–137. Springer-Verlag, 2007.
- [GKM08] Carsten Gutwenger, Karsten Klein, and Petra Mutzel. Planarity testing and optimal edge insertion with embedding constraints. *Journal of Graph Algorithms and Applications*, 12(1):73–95, 2008.
- [GKN05] Emden R. Gansner, Yehuda Koren, and Stephen C. North. Graph drawing by stress majorization. In *Proceedings of the 12th International Symposium on Graph Drawing (GD'04)*, volume 3383 of LNCS, pages 239–250. Springer, 2005. doi:10.1007/b105810.
- [GKNV93] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.
- [GM98] Carsten Gutwenger and Petra Mutzel. Planar polyline drawings with good angular resolution. In *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*, volume 1547 of LNCS, pages 167–182. Springer-Verlag, 1998.
- [GM04] Carsten Gutwenger and Petra Mutzel. An experimental study of crossing minimization heuristics. In *Proceedings of the 11th International Symposium on Graph Drawing (GD'03)*, volume 2912 of LNCS, pages 13–24. Springer, 2004. doi:10.1007/978-3-540-24595-7.
- [GMEJ90] Lindsay J. Groves, Zbigniew Michalewicz, Paul V. Elia, and Cezary Z. Janikow. Genetic algorithms for drawing directed

Bibliography

- graphs. In *Proceedings of the Fifth International Symposium on Methodologies for Intelligent Systems*, pages 268–276, 1990.
- [GMW05] Carsten Gutwenger, Petra Mutzel, and René Weiskircher. Inserting an edge into a planar graph. *Algorithmica*, 41(4):289–308, 2005. doi:10.1007/s00453-004-1128-8.
- [GN00] Emden R. Gansner and Stephen C. North. An open graph visualization system and its applications to software engineering. *Software—Practice and Experience*, 30(11):1203–1234, 2000.
- [Gre12] Tim Grebien. Managing academic Eclipse-based projects. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, August 2012.
- [GSM11] Graeme Gange, Peter Stuckey, and Kim Marriott. Optimal k -level planarization and crossing minimization. In *Proceedings of the 18th International Symposium on Graph Drawing (GD'10)*, volume 6502 of *LNCS*, pages 238–249. Springer, 2011. doi:10.1007/978-3-642-18469-7.
- [GvHM⁺14] Carsten Gutwenger, Reinhard von Hanxleden, Petra Mutzel, Ulf Rüegg, and Miro Spönemann. Examining the compactness of automatic layout algorithms for practical diagrams. In *Proceedings of the Workshop on Graph Visualization in Practice (GraphViP'14)*, Melbourne, Australia, July 2014.
- [Har87] David Harel. Statecharts: A visual formalism for complex systems. *Science of Computer Programming*, 8(3):231–274, June 1987.
- [Har13] Wahbi Haribi. A SyncCharts editor based on YAKINDU SCT. Master's thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2013.
- [HH91] Tyson R. Henry and Scott E. Hudson. Interactive graph layout. In *Proceedings of the 4th Annual ACM Symposium on User Interface Software and Technology (UIST'91)*, pages 55–64. ACM, 1991. doi:10.1145/120782.120788.

- [HH08] Martin Harrigan and Patrick Healy. Practical level planarity testing and layout with embedding constraints. In *Proceedings of the 15th International Symposium on Graph Drawing (GD'07)*, volume 4875 of *LNCS*, pages 62–68. Springer, 2008.
- [Him95] Michael Himsolt. GraphEd: A graphical platform for the implementation of graph algorithms. In *Proceedings of the DIMACS International Workshop on Graph Drawing (GD'94)*, volume 894 of *LNCS*, pages 182–193. Springer, 1995. doi:10.1007/3-540-58950-3_370.
- [Him97] Michael Himsolt. GML: A portable graph file format. Technical report, Universität Passau, Department of Informatics and Mathematics, 1997.
- [HJ05] Stefan Hachul and Michael Jünger. Drawing large graphs with a potential-field-based multilevel algorithm. In *Revised Selected Papers of the 12th International Symposium on Graph Drawing (GD'04)*, volume 3383 of *LNCS*, pages 285–295. Springer, 2005. doi:10.1007/978-3-540-31843-9.
- [HLH12] Weidong Huang, Chun-Cheng Lin, and Mao Lin Huang. An aggregation-based approach to quality evaluation of graph drawings. In *Proceedings of the 5th International Symposium on Visual Information Communication and Interaction (VINCI'12)*, pages 110–113. ACM, 2012. doi:10.1145/2397696.2397712.
- [HM98] Weiqing He and Kim Marriott. Constrained graph layout. *Constraints*, 3(4):289–314, 1998. doi:10.1023/A:1009771921595.
- [HN02] Patrick Healy and Nikola S. Nikolov. How to layer a directed acyclic graph. In *Proceedings of the 9th International Symposium on Graph Drawing (GD'01)*, volume 2265 of *LNCS*, pages 563–566. Springer, 2002.
- [Hoo13] Gregor Hoops. Automatic layout of UML sequence diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, April 2013.

Bibliography

- [Ism12] Alaa Aly Khalaf Ismaeel. *Dynamic Hierarchical Graph Drawing*. PhD thesis, Department of Economics and Business Engineering, Karlsruhe Institute of Technology, 2012.
- [JLM98] Michael Jünger, Sebastian Leipert, and Petra Mutzel. Level planarity testing in linear time. In *Proceedings of the 6th International Symposium on Graph Drawing (GD'98)*, volume 1547 of LNCS, pages 224–237. Springer, 1998.
- [JLMO97] Michael Jünger, Eva K. Lee, Petra Mutzel, and Thomas Odenthal. A polyhedral approach to the multi-layer crossing minimization problem. In *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of LNCS, pages 13–24. Springer, 1997. doi:10.1007/3-540-63938-1.
- [JM97] Michael Jünger and Petra Mutzel. 2-layer straightline crossing minimization: Performance of exact and heuristic algorithms. *Journal of Graph Algorithms and Applications*, 1:1–25, 1997.
- [JM04] Michael Jünger and Petra Mutzel, editors. *Graph Drawing Software*. Springer, 2004.
- [Kan96] Goos Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16(1):4–32, 1996.
- [KD10] Lars Kristian Klauske and Christian Dziobek. Improving modeling usability: Automated layout generation for Simulink. In *Proceedings of the MathWorks Automotive Conference (MAC'10)*, 2010.
- [KD11] Lars Kristian Klauske and Christian Dziobek. Effizientes Erstellen von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus. In *Tagungsband Dagstuhl-Workshop MBEES: Modellbasierte Entwicklung eingebetteter Systeme VII*, pages 115–126, 2011.
- [KK89] Tomihisa Kamada and Satoru Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, 1989.

- [KKM01] Gunnar W. Klau, Karsten Klein, and Petra Mutzel. An experimental comparison of orthogonal compaction algorithms. In *Proceedings of the 8th International Symposium on Graph Drawing (GD'00)*, volume 1984 of LNCS, pages 37–51. Springer, 2001. doi:10.1007/3-540-44541-2.
- [KKR96] Thomas Kamps, Joerg Klein, and John Read. Constraint-based spring-model algorithm for graph layout. In *Proceedings of the Symposium on Graph Drawing (GD'95)*, volume 1027 of LNCS, pages 349–360. Springer, 1996. doi:10.1007/BFb0021818.
- [Kla12] Lars Kristian Klauske. *Effizientes Bearbeiten von Simulink Modellen mit Hilfe eines spezifisch angepassten Layoutalgorithmus*. PhD thesis, Technische Universität Berlin, 2012.
- [Klo12] Paul Klose. A generic framework for the topology-shape-metrics based layout. Master's thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, October 2012.
- [KLSW94] Peter D. Karp, John D. Lowrance, Thomas M. Strat, and David E. Wilkins. The Grasper-CL graph management system. *LISP and Symbolic Computation*, 7:251–290, 1994.
- [KM98] Gunnar W. Klau and Petra Mutzel. Quasi-orthogonal drawing of planar graphs. Technical Report MPI-I-98-1-013, Max-Planck-Institut für Informatik, Saarbrücken, 1998.
- [KM99] Gunnar W. Klau and Petra Mutzel. Optimal compaction of orthogonal grid drawings. In *Proceedings of the 7th International IPCO Conference on Integer Programming and Combinatorial Optimization*, volume 1610 of LNCS, pages 304–319. Springer-Verlag, 1999.
- [KSSvH12] Lars Kristian Klauske, Christoph Daniel Schulze, Miro Spöemann, and Reinhard von Hanxleden. Improved layout for data flow diagrams with port constraints. In *Proceedings of the 7th International Conference on the Theory and Application of*

Bibliography

Diagrams (DIAGRAMS'12), volume 7352 of *LNAI*, pages 65–79. Springer, 2012. doi:10.1007/978-3-642-31223-6.

- [KW01] Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs: Methods and Models*. Number 2025 in LNCS. Springer-Verlag, Berlin, Germany, 2001.
- [LTE⁺09] Agnes Lanusse, Yann Tanguy, Huascar Espinoza, Chokri Mraidha, Sebastien Gerard, Patrick Tessier, Remi Schnekenburger, Hubert Dubois, and François Terrier. Papyrus UML: an open source toolset for MDA. In *Fifth European Conference on Model-Driven Architecture Foundations and Applications (ECMDA-FA'09)*, volume WP09-12 of *CTIT Proceedings Series*, pages 1–4, 2009.
- [Mas94] Toshiyuki Masui. Evolutionary learning of graph layout constraints from examples. In *Proceedings of the 7th Annual ACM Symposium on User Interface Software and Technology (UIST'94)*, pages 103–108. ACM, 1994. doi:10.1145/192426.192468.
- [MFvH10] Christian Motika, Hauke Fuhrmann, and Reinhard von Hanxleden. Semantics and execution of domain specific models. In *2nd Workshop Methodische Entwicklung von Modellierungswerkzeugen (MEMWe 2010) INFORMATIK 2010, GI-Edition – Lecture Notes in Informatics (LNI)*, pages 891–896, Leipzig, Germany, September 2010. Bonner Köllen Verlag.
- [Min06] Mark Minas. Generating meta-model-based freehand editors. In *Proceedings of the 3rd International Workshop on Graph Based Tools (GraBaTs'06)*, volume 1 of *Electronic Communications of the EASST*, Berlin, Germany, 2006.
- [MM07] Sonja Maier and Mark Minas. A pattern-based layout algorithm for diagram editors. In *Proceedings of the First International Workshop on Layout of (Software) Engineering Diagrams (LED'07)*, volume 7 of *Electronic Communications of the EASST*, Berlin, Germany, 2007.

- [MM08] Sonja Maier and Mark Minas. A generic layout algorithm for meta-model based editors. In *Third International Symposium on Applications of Graph Transformations with Industrial Relevance*, volume 5088 of LNCS, pages 66–81. Springer, 2008. doi:10.1007/978-3-540-89020-1.
- [MM10a] Sonja Maier and Mark Minas. Combination of different layout approaches. In *Proceedings of the Second International Workshop on Visual Formalisms for Patterns (VFfP'10)*, volume 31 of *Electronic Communications of the EASST*, Berlin, Germany, 2010.
- [MM10b] Sonja Maier and Mark Minas. Pattern-based layout specifications for visual language editors. In *Proceedings of the Workshop on Visual Formalisms for Patterns (VFfP'09)*, volume 25 of *Electronic Communications of the EASST*, Berlin, Germany, 2010.
- [MNS05] Tiziana Margaria, Ralf Nagel, and Bernhard Steffen. jETI: A tool for remote tool integration. In *Proceedings of the 11th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'05)*, volume 3440 of LNCS, pages 557–562. Springer, 2005. doi:10.1007/978-3-540-31980-1.
- [Mut97] Petra Mutzel. An alternative method to crossing minimization on hierarchical graphs. In *Proceedings of the Symposium on Graph Drawing (GD'96)*, volume 1190 of LNCS, pages 318–333. Springer, 1997. doi:10.1007/3-540-62495-3.
- [NAGa⁺12] Bernadete Maria de Mendonça Neta, Gustavo Henrique Diniz Araujo, Frederico Gadelha Guimarães, Renato Cardoso Mesquita, and Petr Ya. Ekel. A fuzzy genetic algorithm for automatic orthogonal graph drawing. *Applied Soft Computing*, 12(4):1379–1389, 2012. doi:10.1016/j.asoc.2011.11.023.
- [Nor96] Stephen C. North. Incremental layout in DynaDAG. In *Proceedings of the Symposium on Graph Drawing*, volume 1027 of LNCS, pages 409–418. Springer, 1996. doi:10.1007/BFb0021824.

Bibliography

- [NS00] Oliver Niggemann and Benno Stein. A meta heuristic for graph drawing: learning the optimal graph-drawing method for clustered graphs. In *Proceedings of the Working Conference on Advanced Visual Interfaces (AVI'00)*, pages 286–289, New York, NY, USA, 2000. ACM. doi:10.1145/345513.345354.
- [NTB05] Nikola S. Nikolov, Alexandre Tarassov, and Jürgen Branke. In search for efficient heuristics for minimum-width graph layering with consideration of dummy nodes. *Journal of Experimental Algorithmics*, 10, 2005. doi:10.1145/1064546.1180618.
- [NW02] Stephen C. North and Gordon Woodhull. Online hierarchical graph drawing. In *Revised Papers of the 9th International Symposium on Graph Drawing*, volume 2265 of LNCS, pages 232–246. Springer, 2002. doi:10.1007/3-540-45848-4.
- [OSG03] The OSGi Alliance. *OSGi Service Platform, Release 3*. IOS Press, 2003.
- [Pau93] Frances Newbery Paulisch. *The Design of an Extendible Graph Editor*. Number 704 in LNCS. Springer, 1993.
- [Pet95] Marian Petre. Why looking isn't always seeing: Readership skills and graphical programming. *Communications of the ACM*, 38(6):33–44, June 1995.
- [PHG06] Helen C. Purchase, Eve E. Hoggan, and Carsten Görg. How important is the “mental map”? – an empirical investigation of a dynamic graph layout algorithm. In *Proceedings of the 14th International Symposium on Graph Drawing (GD'06)*, volume 4372 of LNCS, pages 184–195. Springer, 2006. doi:10.1007/978-3-540-70904-6.
- [Pri06] Wolfgang Prinz. *The Graph Visualization System (GVS)*. Master's thesis, Institute for Information Systems and Computer Media (IICM), Graz University of Technology, March 2006.
- [Pto14] Claudius Ptolemaeus, editor. *System Design, Modeling, and Simulation using Ptolemy II*. Ptolemy.org, 2014. URL: <http://ptolemy.org/books/Systems>.

- [Pur97] Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of LNCS, pages 248–261. Springer, 1997.
- [Pur02] Helen C. Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages and Computing*, 13(5):501–516, 2002.
- [PvH06] Steffen Prochnow and Reinhard von Hanxleden. Comfortable modeling of complex reactive systems. In *Proceedings of Design, Automation and Test in Europe Conference (DATE'06)*, Munich, Germany, March 2006.
- [PvH07] Steffen Prochnow and Reinhard von Hanxleden. State-chart development beyond WYSIWYG. In *Proceedings of the ACM/IEEE 10th International Conference on Model Driven Engineering Languages and Systems (MoDELS'07)*, volume 4735 of LNCS, pages 635–649, Nashville, TN, USA, October 2007. IEEE. doi:10.1007/978-3-540-75209-7.
- [Rie10] Martin Rieß. A graph editor for algorithm engineering. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, September 2010.
- [RKD⁺14] Ulf Rüegg, Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. Stress-minimizing orthogonal layout of data flow diagrams with ports. In *Proceedings of the 22nd International Symposium on Graph Drawing (GD'14)*, 2014.
- [RLHV03] Urša Reja, Katja Lozar Manfreda, Valentina Hlebec, and Vasja Vehovar. Open-ended vs. close-ended questions in web questionnaires. *Metodološki zvezki*, 19:159–177, 2003.
- [RNHL99] John Reekie, Stephen Neuendorffer, Christopher Hylands, and Edward A. Lee. Software practice in the Ptolemy project. Technical report, Gigascale Semiconductor Research Center, April 1999. <http://ptolemy.eecs.berkeley.edu/publications/papers/99/softwareprac/>.

Bibliography

- [RSOR98] A. Rosete-Suarez and A. Ochoa-Rodriguez. Genetic graph drawing. In P. Nolan, R. A. Adey, and G. Rzevski, editors, *Applications of Artificial Intelligence in Engineering XIII*, volume 1 of *Software Studies*. WIT Press / Computational Mechanics, 1998. doi:10.2495/AI980091.
- [Rus13] Adrian Rusu. Tree drawing algorithms. In Roberto Tamassia, editor, *Handbook of Graph Drawing and Visualization*, pages 155–192. CRC Press, 2013.
- [RWC11] Dan Rubel, Jaime Wren, and Eric Clayberg. *The Eclipse Graphical Editing Framework (GEF)*. Eclipse Series. Addison-Wesley, 2011.
- [San94] Georg Sander. Graph layout through the VCG tool. Technical Report A03/94, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, October 1994.
- [San96a] Georg Sander. A fast heuristic for hierarchical Manhattan layout. In *Proceedings of the Symposium on Graph Drawing (GD'95)*, volume 1027 of *LNCS*, pages 447–458. Springer, 1996.
- [San96b] Georg Sander. Layout of compound directed graphs. Technical Report A/03/96, Universität des Saarlandes, FB 14 Informatik, 66041 Saarbrücken, June 1996.
- [San04] Georg Sander. Layout of directed hypergraphs with orthogonal hyperedges. In *Proceedings of the 11th International Symposium on Graph Drawing (GD'03)*, volume 2912 of *LNCS*, pages 381–386. Springer, 2004.
- [SBPM09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF – Eclipse Modeling Framework*. Eclipse Series. Addison-Wesley, second edition, 2009.
- [Sch90] Walter Schnyder. Embedding planar graphs on the grid. In *SODA '90: Proceedings of the First Annual ACM-SIAM Symposium on Discrete Algorithms*, pages 138–148. SIAM, 1990.

- [Sch01] Falk Schreiber. *Visualisierung biochemischer Reaktionsnetze*. PhD thesis, Universität Passau, Fakultät für Informatik und Mathematik, 2001.
- [Sch06] Douglas C. Schmidt. Model-driven engineering. *Computer*, 39(2):25–31, February 2006. doi:10.1109/MC.2006.58.
- [Sch11] Christoph Daniel Schulze. Optimizing automatic layout for data flow diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2011.
- [Sch14] Enno Schwanke. Generierung von UML Klassendiagrammen aus Java Code in Eclipse. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2014.
- [SDvH14a] Miro Spönemann, Björn Duderstadt, and Reinhard von Hanxleden. Evolutionary meta layout of graphs. In *Proceedings of the 8th International Conference on the Theory and Application of Diagrams (DIAGRAMS'14)*, volume 8578 of *LNAI*, pages 16–30. Springer, 2014.
- [SDvH14b] Miro Spönemann, Björn Duderstadt, and Reinhard von Hanxleden. Evolutionary meta layout of graphs. Technical Report 1401, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, January 2014. ISSN 2192-6247.
- [See97] Jochen Seemann. Extending the Sugiyama algorithm for drawing UML class diagrams: Towards automatic layout of object-oriented software diagrams. In *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of *LNCS*, pages 415–424. Springer, 1997.
- [Sel03] Bran Selic. The pragmatics of model-driven development. *IEEE Software*, 20(5):19–25, September 2003. doi:10.1109/MS.2003.1231146.

Bibliography

- [SFvH09] Miro Spönemann, Hauke Fuhrmann, and Reinhard von Hanxleden. Automatic layout of data flow diagrams in KIELER and Ptolemy II. Technical Report 0914, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2009.
- [SFvHM10] Miro Spönemann, Hauke Fuhrmann, Reinhard von Hanxleden, and Petra Mutzel. Port constraints in hierarchical layout of data flow diagrams. In *Proceedings of the 17th International Symposium on Graph Drawing (GD'09)*, volume 5849 of *LNCS*, pages 135–146. Springer, 2010. doi:10.1007/978-3-642-11805-0.
- [SGW94] Bran Selic, Garth Gullekson, and Paul T. Ward. *Real-Time Object-Oriented Modeling*. John Wiley & Sons, 1994.
- [Sie09] Martin Siebenhaller. *Orthogonal Graph Drawing with Constraints: Algorithms and Applications*. PhD thesis, Universität Tübingen, Mathematisch-naturwissenschaftliche Fakultät, 2009.
- [Spö09] Miro Spönemann. On the automatic layout of data flow diagrams. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, March 2009.
- [SSM⁺13] Miro Spönemann, Christoph Daniel Schulze, Christian Motika, Christian Schneider, and Reinhard von Hanxleden. KIELER: building on automatic layout for pragmatics-aware modeling (showpiece). In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*, San Jose, CA, USA, September 2013.
- [SSRvH14a] Miro Spönemann, Christoph Daniel Schulze, Ulf Rüegg, and Reinhard von Hanxleden. Counting crossings for layered hypergraphs. In *Proceedings of the 8th International Conference on the Theory and Application of Diagrams (DIAGRAMS'14)*, volume 8578 of *LNAI*, pages 9–15. Springer, 2014. doi:10.1007/978-3-662-44043-8_2.

- [SSRvH14b] Miro Spönemann, Christoph Daniel Schulze, Ulf Rüegg, and Reinhard von Hanxleden. Drawing layered hypergraphs. Technical Report 1404, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, April 2014. ISSN 2192-6247.
- [SSSvH14] Christoph Daniel Schulze, Miro Spönemann, Christian Schneider, and Reinhard von Hanxleden. Two applications for transient views in software development environments (show-piece). In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'14)*, Melbourne, Australia, July 2014.
- [SSvH12a] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. Transient view generation in Eclipse. In *Proceedings of the First Workshop on Academics Modeling with Eclipse*, Kgs. Lyngby, Denmark, July 2012.
- [SSvH12b] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. Transient view generation in Eclipse. Technical Report 1206, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, June 2012. ISSN 2192-6247.
- [SSvH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. Just model! – Putting automatic synthesis of node-link-diagrams into practice. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'13)*, pages 75–82, San Jose, CA, USA, 15–19 September 2013. doi:10.1109/VLHCC.2013.6645246.
- [SSvH14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. Drawing layered graphs with port constraints. *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout*, 25(2):89–106, 2014. doi:10.1016/j.jvlc.2013.11.005.
- [ST99] Janet M. Six and Ioannis G. Tollis. Circular drawings of biconnected graphs. In Michael Goodrich and Catherine

Bibliography

- McGeoch, editors, *Algorithm Engineering and Experimentation*, volume 1619 of *LNCS*, pages 57–73. Springer, 1999.
doi:10.1007/3-540-48518-X.
- [ST06] Janet M. Six and Ioannis G. Tollis. A framework and algorithms for circular drawings of graphs. *Journal of Discrete Algorithms*, 4(1):25–50, 2006. doi:10.1016/j.jda.2005.01.009.
- [Ste01] Alexander Stedile. JMFGGraph—A modular framework for drawing graphs in Java. Master’s thesis, Institute for Information Systems and Computer Media (IICM), Graz University of Technology, November 2001.
- [STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiro Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, February 1981.
- [Sug02] Kozo Sugiyama. *Graph Drawing and Applications for Software and Knowledge Engineers*, volume 11 of *Software Engineering and Knowledge Engineering*. World Scientific, 2002.
- [SV06] Thomas Stahl and Markus Völter. *Model-Driven Software Development: Technology, Engineering, Management*. John Wiley & Sons, 2006.
- [SvH14] Christoph Daniel Schulze and Reinhard von Hanxleden. Automatic layout in the face of unattached comments. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC’14)*, Melbourne, Australia, July 2014.
- [Tam87] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal of Computing*, 16(3):421–444, 1987.
- [Tam13] Roberto Tamassia, editor. *Handbook of Graph Drawing and Visualization*. CRC Press, 2013.

- [TAvH11] Claus Traulsen, Torsten Amende, and Reinhard von Hanxleden. Compiling SyncCharts to Synchronous C. In *Proceedings of the Design, Automation and Test in Europe Conference (DATE'11)*, pages 563–566, Grenoble, France, March 2011. IEEE.
- [TDB88] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, 18(1):61–79, 1988.
- [Tet98] Andrea G. B. Tettamanzi. Drawing graphs with evolutionary algorithms. In Ian C. Parmee, editor, *Adaptive Computing in Design and Manufacture*. Springer, 1998.
- [Tha13] Bernhard Thalheim. The conception of the model. In *Proceedings of the 16th International Conference on Business Information Systems (BIS'13)*, volume 157 of *Lecture Notes in Business Information Processing*, pages 113–124. Springer, 2013. doi:10.1007/978-3-642-38366-3.
- [UBSE98] J. Utech, J. Branke, H. Schmeck, and P. Eades. An evolutionary algorithm for drawing directed graphs. In *Proceedings of the International Conference on Imaging Science, Systems, and Technology (CISST'98)*, pages 154–160. CSREA Press, 1998.
- [vGR13] Pieter van Gorp and Louis M. Rose. The petri-nets to state-charts transformation case. In *Proceedings of the Sixth Transformation Tool Contest*, volume 135 of *Electronic Proceedings in Theoretical Computer Science*, pages 16–31. Open Publishing Association, 2013. doi:10.4204/EPTCS.135.3.
- [vH08] Reinhard von Hanxleden. On the pragmatics of model-based design—position statement. In *Pre-Proceedings of the 15th International Monterey Workshop on Foundations of Computer Software, Future Trends and Techniques for Development*, Budapest, September 2008.
- [vHDM⁺14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen

Bibliography

- Mercer, and Owen O'Brien. SCCharts: Sequentially Constructive Statecharts for safety-critical applications. In *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI'14)*, Edinburgh, UK, June 2014. ACM.
- [vHMA⁺13] Reinhard von Hanxleden, Michael Mendler, Joaquín Aguado, Björn Duderstadt, Insa Fuhrmann, Christian Motika, Stephen Mercer, and Owen O'Brien. Sequentially Constructive Concurrency—A conservative extension of the synchronous model of computation. In *Proc. Design, Automation and Test in Europe Conference (DATE'13)*, pages 581–586, Grenoble, France, March 2013. IEEE.
- [VKEH06] Markus Völter, Bernd Kolb, Sven Efftinge, and Arno Haase. From front end to code – MDSD in practice. *Eclipse Corner Articles*, June 2006. www.eclipse.org/articles.
- [Vra09] Dana Vrajitoru. Multiobjective genetic algorithm for a graph drawing problem. In *Proceedings of the Midwest Artificial Intelligence and Cognitive Science Conference*, pages 28–43, 2009.
- [Wad01] Vance Waddle. Graph layout for displaying data structures. In *Proceedings of the 8th International Symposium on Graph Drawing (GD'00)*, volume 1984 of LNCS, pages 98–103. Springer, 2001.
- [WCL⁺05] Sanjiva Weerawarana, Francisco Curbera, Frank Leymann, Tony Storey, and Donald F. Ferguson. *Web Services Platform Architecture: SOAP, WSDL, WS-Policy, WS-Addressing, WS-BPEL, WS-Reliable Messaging and More*. Prentice Hall, 2005.
- [Wei01] René Weiskircher. Drawing planar graphs. In Michael Kaufmann and Dorothea Wagner, editors, *Drawing Graphs: Methods and Models*, volume 2025 of LNCS, pages 23–45. Springer, 2001. doi:10.1007/3-540-44969-8.
- [WEK04] Roland Wiese, Markus Eiglsperger, and Michael Kaufmann. yFiles – visualization and automatic layout of graphs. In

Bibliography

- Michael Jünger and Petra Mutzel, editors, *Graph Drawing Software*, pages 173–191. Springer, 2004. doi:10.1007/978-3-642-18638-7.
- [Wer11] Stephan Wersig. Ein Web Service für das automatische Layout von Graphen. Diploma thesis, Christian-Albrechts-Universität zu Kiel, Department of Computer Science, October 2011.
- [Wil97] Graham J. Wills. NicheWorks—interactive visualization of very large graphs. In *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of LNCS, pages 403–414. Springer, 1997.
- [WMS10] Michael Wybrow, Kim Marriott, and Peter J. Stuckey. Orthogonal connector routing. In *Proceedings of the 17th International Symposium on Graph Drawing (GD'09)*, volume 5849 of LNCS, pages 219–231. Springer, 2010. doi:10.1007/978-3-642-11805-0.
- [WPCM02] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. Cognitive measurements of graph aesthetics. *Information Visualization*, 1(2):103–110, 2002.
- [YHG11] Pei Shan Yap, J. Hosking, and John C. Grundy. Automatic diagram layout support for the Marama meta-toolset. In *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC'11)*, pages 61–64, 2011. doi:10.1109/VLHCC.2011.6070379.

