Text in Diagrams

Challenges to and Opportunities of Automatic Layout

Dipl.-Inf. Christoph Daniel Schulze

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2018

Kiel Computer Science Series (KCSS) 2019/4 dated 2019-10-01

URN:NBN urn:nbn:de:gbv:8:1-zs-00000357-a8
ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

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

The author can be contacted via cd.schulze@gmx.net

Published by the Department of Computer Science, Kiel University

Real-Time and Embedded Systems Group

Please cite as:

▻ Christoph Daniel Schulze. *Text in Diagrams: Challenges to and Opportunities of Automatic Layout* Number 2019/4 in Kiel Computer Science Series. Department of Computer Science, 2019. Dissertation, Faculty of Engineering, Kiel University.

```
@book{Schulze19,
  author    = {Christoph Daniel Schulze},
  title     = {Text in Diagrams:
               Challenges to and Opportunities of Automatic Layout},
  publisher = {Department of Computer Science, CAU Kiel},
  year      = {2019},
  number    = {2019/4},
  doi       = {10.21941/kcss/2019/4},
  series    = {Kiel Computer Science Series},
  note      = {Dissertation, Faculty of Engineering,
               Kiel University.}
}
```


Herstellung und Verlag: BoD – Books on Demand, Norderstedt

1. Gutachter:	Prof. Dr. Reinhard von Hanxleden Christian-Albrechts-Universität Kiel
2. Gutachter:	Dr. Helen C. Purchase University of Glasgow Glasgow

Datum der mündlichen Prüfung: 17.07.2019

Zusammenfassung

Visuelle Programmiersprachen auf Basis von Knoten-Kanten-Diagrammen wirken ihren textuellen Gegenstücken oft überlegen da grafische Darstellungen zumeist als intuitiver verständlich wahrgenommen werden als Text. Diese Wahrnehmung greift allerdings zu kurz. Einerseits kommen auch visuelle Sprachen selten ohne Text aus, welcher zuweilen Ausmaße annehmen kann, die sich negativ auf die Größe der Diagramme auswirken. Andererseits sind Diagramme nur dann effektive Kommunikationsmittel, wenn ihre Elemente sinnvoll platziert sind—eine zeitraubende Aufgabe, von der *automatische Layoutalgorithmen* Benutzer zu befreien suchen.

Diese Arbeit behandelt die Schnittmenge zwischen automatischen Layoutalgorithmen und Text in Diagrammen: die Herausforderungen dabei, Text überhaupt automatisch in Diagrammen zu platzieren, und das sich daraus ergebende Potential, den Umgang von Benutzern mit Text zu verbessern.

Dieses Grundthema wird im Laufe der Arbeit zu verschiedenen Beiträgen entwickelt. Zunächst stelle ich Ansätze vor, den verbreiteten *ebenenbasierten Layoutansatz* sowie einen neuen Layoutalgorithmus für UML-Sequenzdiagramme um die Platzierung von Text so zu erweitern, dass genug Platz für den Text bleibt und, je nach Ansatz, dessen Auswirkung auf die Größe des Diagramms beschränkt bleibt. Des Weiteren untersuche ich, wie Benutzer textuelle Kommentare in Diagrammen so platzieren, dass Zusammenhänge mit anderen Diagrammelementen entstehen. Das führt zu Ansätzen, derlei Zusammenhänge automatisch zu erkennen um Layoutalgorithmen davon abzuhalten, sie zu zerstören. Schließlich zeige ich, wie Text dynamisch angepasst werden kann, sowohl mit als auch ohne Informationsverlust, um die Größe von Diagrammen oder die Menge der darin enthaltenen Informationen zu reduzieren. Derlei Ansätze haben das Potential, die Rolle von Layoutalgorithmen zu verändern: statt lediglich eine Menge fixer Diagrammelemente zu platzieren haben sie nun die Möglich-

keit, während des Layoutprozesses entstehende Informationen zu benutzen, um Text so anzupassen, dass er weniger Einfluss auf die Größe von Diagrammen hat. Diese Ideen werden im Rahmen verschiedener Fallbeispiele untersucht und bewertet.

Alle in dieser Arbeit vorgestellten Ansätze sind in *Open Source*-Projekten implementiert und werden großteils bereits in realen Anwendungen eingesetzt.

Abstract

Visual programming languages based on node-link diagrams are often deemed to be superior to their textual counterparts because of their supposed inherent intuitiveness. This, however, is not the whole story. Even visual languages usually cannot get away without at least a bit of text, some requiring so much that it enlarges diagrams considerably. Also, communicating effectively requires diagram elements to be carefully placed—a time-consuming and potentially tedious process. *Automatic layout algorithms*, studied in the area of *graph drawing*, aim to alleviate users of the need to place elements manually.

This thesis studies automatic layout as it relates to text in diagrams: the challenges of handling text in the first place, and the opportunities that emerge to improve how users work with text in diagrams.

This general theme is developed into different contributions. First, I introduce ways of placing text into the popular *layered approach* to graph drawing and into a new layout algorithm for UML sequence diagrams. They ensure that enough space is available for the text, and some aim to reduce its impact on diagram size. Second, I study how users place textual comments in diagrams to establish relations to other diagram elements. This leads to approaches for inferring such relations with the aim of keeping layout algorithms from destroying them. Finally, I study how text can be dynamically altered, both with and without loss of information, to reduce the size of diagrams or the amount of information contained therein. This can change the role of layout algorithms: instead of simply placing a set of fixed elements, they can now change text while computing a layout since they know how long it can get before having an impact on a diagram's size. I explore and evaluate different approaches of integrating these ideas into applications in the context of several case studies.

All approaches put forth in this thesis are implemented in open source projects, and many are already in use in real-world applications.

Brief Contents

1 Introduction **1**
1.1 Three Visual Languages 6
1.2 Principles 19
1.3 Contributions 21
1.4 Related Work 24
1.5 Publications 30
1.6 Outline 33

2 Foundations **35**
2.1 Basic Terminology 35
2.2 Aesthetics 39
2.3 Box Plots 44
2.4 The Eclipse Platform 45
2.5 The Eclipse Layout Kernel 52
2.6 KIELER Lightweight Diagrams (KLighD) 61

I Laying the Foundations: Automatic Layout **65**

3 Flow-Based Diagrams **67**
3.1 The Layered Approach 70
3.2 Micro Layout of Nodes 89
3.3 Edge Label Placement 128
3.4 Evaluation 154

4 Sequence Diagrams **187**
4.1 Sequence Diagrams as Graphs 188
4.2 Laying Out Sequence Diagrams 192
4.3 The KieSL Language 210

4.4 Evaluation 212

II Reaping the Rewards: Layout Pragmatics 219

5 Comment Attachment 221
5.1 Comments and Secondary Notation 223
5.2 The Comment Attachment Pipeline 225
5.3 Measuring Attachment 227
5.4 Implementing the Pipeline 238
5.5 Evaluation 241

6 Label Management 249
6.1 Label Management Strategies 254
6.2 Integrating Label Management 260
6.3 Designing the User Interface 273
6.4 Evaluation 277

7 Conclusions 291
7.1 Summary 291
7.2 Lessons Learned 294
7.3 Open Problems 297
7.4 Conclusion 299

A Sample Diagrams 301
A.1 SCCharts 301
A.2 UML Sequence Diagrams 312

Bibliography 319

Lists 339

Glossary 355

Chapter 1

Introduction

Information technology is a field with its fair share of controversial discussions. Among them are such diverse disputes as Windows versus Linux, micro kernels versus macro kernels, and of course the all-time favorite, Emacs versus Vi(m). While I would not dare take any particular side in these matters, there is one discussion that touches upon the topic of this thesis: that of textual languages versus visual languages.

After programming computers through toggle switches or punch cards somewhat declined in popularity, textual programming languages were introduced in the 1950s. They became and still are the most popular tool for writing computer programs and have been the subject of much research and development, both industrial and academical. Today, there are many textual programming languages to choose from. Programmers are not even limited to using one of the established general-purpose languages, such as C or Java, but can develop their own Domain-Specific Languages (DSLs), tailored specifically to the problem at hand.

Given the early rise and immense popularity of textual languages, it is no surprise that the available editing tools have evolved from simple text editors to feature-packed Integrated Development Environments (IDEs), Eclipse[1] being a popular example. Programmers have come to expect editors to provide syntax highlighting, content assist, integrated language documentation, graphical debugging, and even support for refactoring source code. The power and might of text editors has grown to the extent where some editors are deemed to be entire operating systems, although whether this statement is to be taken as a compliment or not admittedly depends on the person uttering it.

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

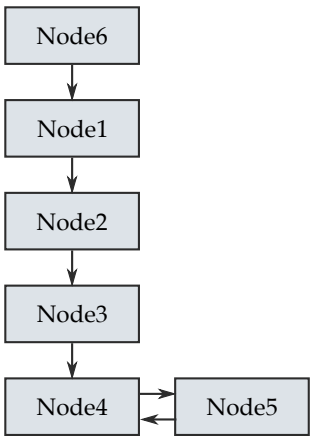
Still, not all is well in the land of textual languages. One problem faced by beginners in particular is that textual languages are not necessarily intuitive. The meaning of keywords, while often borrowed from English, may not be obvious, which is even more true for the various symbols textual languages usually employ. Furthermore, the syntax and semantics (what du Boulay calls the *notation* [Bou86]) are not readily obvious. Programmers often start with an empty text file and few hints as to what they should do with it unless they already know the language. At the same time, however, textual languages are very strict, which contrasts with how humans use spoken languages: there, the basic syntax and semantics much rather seem to serve as a base from which to launch into considerably more adventurous linguistic constructs that, rather amazingly, still manage to convey the intended meaning to others. Of course, this works somewhat less well when programming computers.

A second problem follows from the fact that textual languages are abstract: it can be hard to spot problems. Consider, for example, a simple language for describing graphs consisting of nodes connected by edges. Figure 1.1 shows such a language, used here in an attempt to describe a simple cycle. The attempt's considerable lack of success is not necessarily obvious by looking just at the text; it becomes immediately apparent, however, when looking at the visualization.

Finally, there is the problem of keeping an overview of what is expressed through a textual language. A single screen of text may seem deceptively simple, but even its high-level structure can be almost arbitrarily hard to understand. There seems to be general agreement among most developers that adhering to language-specific formatting guidelines (ranging in specificity from "do use line breaks" to "leave one space character each around either side of an operator") is a necessity to keep this task manageable. The problem is exacerbated by the hundreds of files and tens of thousands of lines of code software systems quickly grow into. Keeping an overview in this context is a challenge, to say the least.

The UML is designed to help keep an overview by visualizing the structure and the behavior of software systems. It is an example of a formally defined [Obj17] visual language. Visual languages have a long history. The origins of flowcharts, for example, can be traced back at least to the

```
node Node1
     edgeTo Node2
node Node2
     edgeTo Node3
node Node3
     edgeTo Node4
node Node4
     edgeTo Node5
node Node5
     edgeTo Node4
node Node6
     edgeTo Node1
```



(a) Textual description **(b)** Graphical view

Figure 1.1. (a) A simple DSL used in an attempt to describe a graph consisting of six nodes connected to form a cycle. Each node must be given a name to be able to refer to it. Due to a mix-up with these names, the graph is not actually a cycle—a problem not readily apparent when looking only at the text. **(b)** A visualization of the graph. The fact that it is not a cycle is immediately apparent.

1920s [GG21], while the first computer-based visual programming languages emerged in the 1960s [Nic95]. Blackwell found that there exists an assumption for visual languages to be inherently intuitive and natural, repeatedly citing the belief that when explaining things to people, we often find ourselves scribbling on a sheet of paper in the hope that this will help [Bla96] (which, depending on our drawing abilities, may or may not in fact be the case). And indeed, the example in Figure 1.1 shows that a good visual representation can go a long way. It is important to realize, however, that while textual languages are not without problems, neither are visual languages.

To start with, diagrams drawn according to formally defined visual languages may actually not be as intuitively understandable as is commonly thought. As Petre points out [Pet95], it can take more time to interpret a diagram than to read a textual representation of its content. In fact, the

placement of elements can even be misleading. For example, proximity and alignment between elements establish relationships between them that may or may not be consistent with the information the diagram is supposed to represent. This is what the term *secondary notation* [Pet06] refers to: "layout, color, other cues to convey extra meaning, above and beyond the 'official' semantics of the language" [GP96]. Using and interpreting secondary notation effectively may be one of the factors that separate novice users from expert users of a given language [Pet95] and has a big impact on a diagram's readability.

A second problem of visual languages is the sheer size of diagrams in the real world: in software systems developed using visual languages, diagrams can easily grow to contain thousands of elements [RG12]. It is obvious that there is no way to display them all on a single screen and still keep them legible (simply painting the screen black would be a much easier way to achieve similarly useful results). The ability to zoom into a diagram is the most common solution, but can only ever show a few elements at a time. It is easy to get lost in the details and loose the context surrounding the displayed elements [CKB08]. While text also quickly exceeds the available screen space, at least users usually only navigate it along one axis.

A third problem is that hardly any visual language gets away without text to define parts of a diagram's semantics. How best to integrate text and visuals is not always obvious, though. In Figure 1.1b, it was the text in the boxes that allowed us to infer exactly where the problem was since the boxes on their own provided no indication as to which semantic element they represented. While this example did not pose much of a problem, it can be a challenge to place text such that its relation to other diagram elements is evident. Also, some visual languages are prone to large amounts of text which can negatively impact diagram size and increase visual clutter—we shall encounter examples of this in the next section.

Finally, editors for visual languages are not nearly as sophisticated as their textual counterparts. While a pencil and a sheet of paper constitute an extremely simple and direct user interface (so much so that it almost seems odd to use that term in the first place), manipulating graphical elements through the interface of a computer is much more cumbersome. Admittedly, many editors have since started to support the user on a small scale, for

example by helping to properly align elements while dragging them across the drawing area. But on a larger scale, users are still pretty much on their own and have to manually devise the overall element placement. That this can grow to become a real problem becomes apparent once we allow for the fact that diagrams evolve and thus require the placement of their elements to evolve with them. Many of today's textual editors support this by providing powerful refactoring features, but visual editors usually deny their users a similar degree of comfort: merely adding a single element to the diagram can require lots of other elements to be moved manually to provide the necessary space for it. In a number of formative interviews among LabVIEW[2] programmers, Henley and Fleming have found this to be a real problem [HF16].

Based on observations from a study in the automotive industry, Klauske and Dziobek estimated that developers spend about 30% of their time on such manual layout adjustments [KD10]. This estimate does have to be taken with a grain of salt, though. This particular study focused on software developers, whose main task is to think about the program logic and how to express it diagrammatically. I believe that the true fraction may be even higher for tasks where the diagram's content itself requires less thinking about, thus focusing more on finding the best placement. Petre argues that the manual placement of elements may actually contribute to the appeal of visual languages: users have a lot of freedom in designing their diagrams, and moving things around may in itself already be a gratifying experience, even if somewhat cumbersome [Pet95]. As she puts it, "[the] *illusion* of accessibility may be more important than the reality." Still, there is no denying the fact that layout adjustments constitute a substantial hit on productivity.

Layout algorithms automate the task of placing a diagram's elements on the canvas, with different algorithms tailored to the requirements of different visual languages. The existence of such algorithms creates opportunities for new kinds of user experiences. For example, when diagrammatic representations do not need to be laid out manually, they can be automatically synthesized and continuously updated to support users in their editing

[2] `https://www.ni.com/labview`

tasks [SSH13]. Similar to what we saw in Figure 1.1, this can help spot problems, or even serve as a graphical navigation aid where selecting an element in the diagram navigates to the corresponding definition in the document being edited. Nevertheless, automatic layout is also rich in challenges, having fueled an extensive body of research over the years. Quite obviously, computing a sensible layout in the first place can already be hard enough. But even once a layout algorithm works sufficiently well, it is the experience of our research group that getting users to choose it over manual placement is a matter of getting the details right, some of them rather subtle.

Despite their shortcomings, the reality is that visual languages are widely used, which is why improving the way users work with them (what we, following Fuhrmann et al., call the *pragmatics* aspects of a language [FH10]) seems like a worthwhile thing to do. The angle from which I approach that goal in this thesis is what textual and visual languages have in common: text.

The first part will be concerned with integrating text into automatic layout, tackling questions such as how to place different kinds of textual elements and how to make sure that there is enough space to do so. During the second part, I will switch to the perspective of what I will call *layout pragmatics*: ways automatic layout can be used to improve the way users work with text in diagrams.

To guide us a little on our journey, we will need a selection of visual languages with which to explain and evaluate the concepts we are about to discuss. Once we have introduced some, we can further drill into the problems to be solved.

1.1 Three Visual Languages

Just like textual languages, visual languages can have different purposes. While the UML for example is mostly meant to visualize the architecture and behavior of systems and is thus descriptive in nature (at least in the ways most people seem to use it [Pet13]), LabVIEW (National Instruments)

is more of a programming language. We will now look at languages from both categories, along with the editing concepts associated with them.

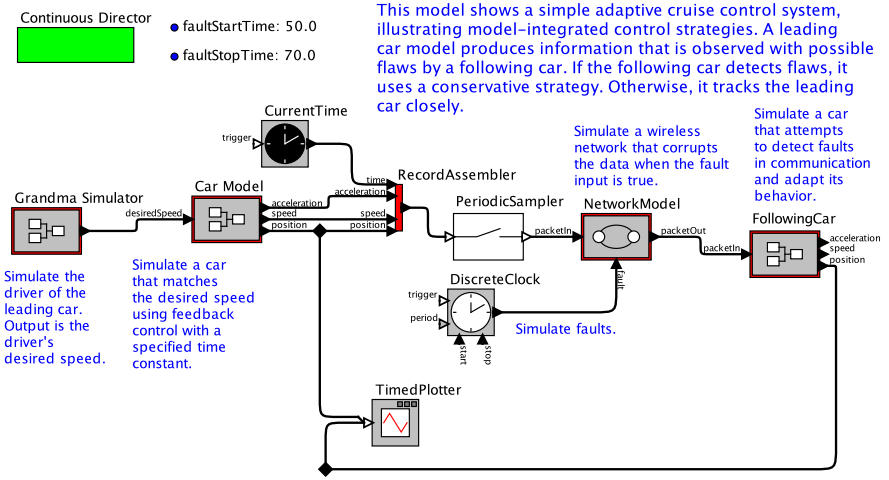
1.1.1 Ptolemy II

Ptolemy II is an "open-source modeling and simulation tool" [Pto14] that offers different syntaxes (not all of them visual) and different models of computation that govern how a model is executed. The focus is on *heterogeneous modeling*: different parts of a single model can make use of different syntaxes and different models of computation. The syntax we will be most interested in for the purposes of this thesis is what the Ptolemy II developers call *block diagrams*.

Block diagrams such as the one in Figure 1.2 are a type of *node-link diagram* where each *node* consumes and/or produces data and thus implements some kind of behavior. Because of this fact, nodes are also called *actors*—hence the term *actor-oriented modeling* [LNW03] for this modeling approach. Data is transferred between the nodes through *links* (or *edges*) that establish connections between two or more actors. It is this flow of data that is usually emphasized when laying out block diagrams: an actor that produces data tends to be placed to the left of an actor that consumes it (in Figure 1.2, only the edge from *FollowingCar* to *TimedPlotter* points leftwards). Diagrams that emphasize this kind of data flow are—to hardly anyone's surprise—called *data flow diagrams*.

Each node provides explicit end points, called *ports*, for edges to connect to. Ports can be input ports or output ports. Comparable to method parameters in textual languages, changing which of a node's input ports an edge connects to changes how the data transmitted through the edge influences the node's computations. Similarly, changing the output port an edge connects to changes which computation result is transmitted through the edge.

Interestingly, edges in Ptolemy II block diagrams are not actually directed, although the description thus far may have made it appear so (not to mention that they look like arrows in the diagrams). The appearance of direction is caused by the fact that ports are rendered as small triangles



Author: Xiaojun Liu and Edward A. Lee

Figure 1.2. A block diagram taken from the example models distributed with Ptolemy II.[3] Note how all but one of the edges point rightwards. The elements in this diagram were placed manually.

that point towards or away from their node. These are easily mistaken for arrowheads.

Ptolemy II allows the exact behavior of a node to be specified through different means. The most obvious one is plain Java code, but a node can also contain another Ptolemy II model executed on the node's input to determine its output (as is the case with *Grandma Simulator*, *Car Model*, and *FollowingCar* in Figure 1.2). This effectively makes that node the parent of another diagram, turning it into what we will call a *hierarchical node*. The contained diagram can use a different model of computation or even another syntax, which is how Ptolemy II supports heterogeneity.

Ptolemy II diagrams make use of text in two ways. First, since not every type of node is represented by a distinct icon, node labels are used for disambiguation. Similarly, port labels describe what purpose each port serves. Both node and port labels are usually rather short. Second, comments can

be used to describe what a model and its different elements do. Comments are usually longer, sometimes consisting of several paragraphs of text. In Figure 1.2, the developers of the model took great care to write comments that explain how the model works, and to place them in a way that makes it immediately clear which element each comment describes.

Ptolemy II ships with an editing environment called *Vergil* (see Figure 1.3) to create and change Ptolemy II models. Vergil is a typical drag-and-drop-based diagram editor: new elements are added by dragging them from a palette of available actors to the drawing area, and existing ones are moved by dragging them across the diagram. Defining links works similarly, with the orthogonal edges automatically routed around any existing diagram elements.

One downside of Vergil, as of many visual editors, is that each diagram is opened in a separate window. While this works fine in simple cases, most Ptolemy II models consist of several nested diagrams. In fact, out of 1087 demo models we analyzed that ship with Ptolemy II, 387 contained between 1 and 57 nested diagrams, for an average of 4.8. Having to view each in a separate window makes it hard to get a good overview of the model, let alone to navigate through it efficiently.

The *KIELER Ptolemy Browser* [RSS+13] was built to help solve this issue when browsing Ptolemy II models. Here, each diagram is shown in the context of the diagram it is embedded in, as Figure 1.4 shows. Hierarchical nodes can be expanded and collapsed at will, depending on what part of the model the user wants to focus on at any given time. Of course, an expanded node will be larger than a collapsed one, possibly overlapping other nodes in the diagram. The responsibility of avoiding these overlaps is delegated to automatic layout algorithms without which this application would be inconceivable.

1.1.2 Sequentially Constructive Statecharts (SCCharts)

In 1987, David Harel introduced a visual language called *statecharts* [Har87] in an attempt to address a lack of scalability in then prevalent *state-transition*

[3]This particular diagram is called 'Car Tracking' and can be found in the list of demo models for the 'Continuous' domain in Ptolemy II.

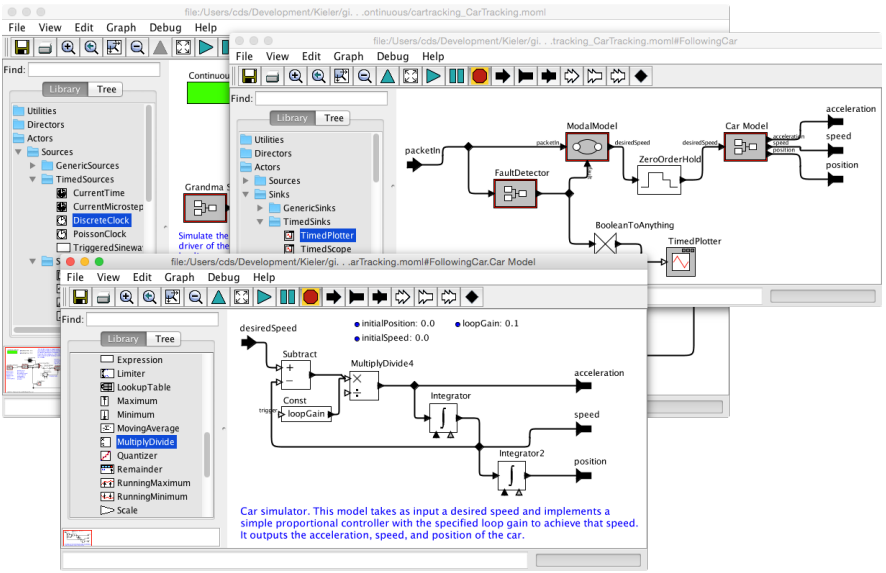


Figure 1.3. The model from Figure 1.2, being edited in the Vergil editor. The palette of available actors is shown at the left hand side of each window while the main area is reserved for the diagram itself. In this case, the model consists of a number or nested diagrams, each of which has to be opened in a separate window. Keeping an overview of everything becomes a challenge when multiple windows are fighting for screen space.

diagrams. These diagrams basically consist of the different states a system can be in (nodes) and the transitions that allow the system to change states (edges). The emphasis here is on the flow of control inside the system, which is why statecharts are an example of *control flow diagrams*.

As systems become more complex, the number of states necessary to describe them explodes. Harel's statecharts address this issue by introducing three concepts: *hierarchy*, *broadcast communication*, and *orthogonality*.

Hierarchy (originally called *depth*) is similar to hierarchy in Ptolemy II in that it allows the behavior of states to be defined in terms of child states. Broadcast communication allows different parts of a system to commu-

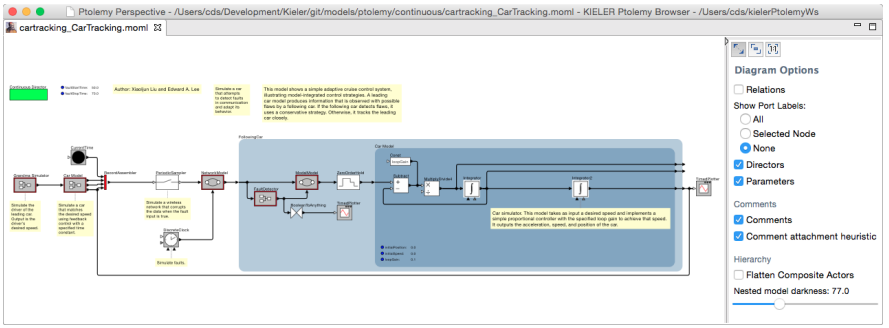


Figure 1.4. The same diagrams as in Figure 1.3, this time shown in the KIELER Ptolemy Browser. Nested diagrams are embedded in their parent actors, which allows them to be viewed in the context of their surroundings. Expanding and collapsing actors requires the layout to adapt dynamically, which is achieved using automatic layout algorithms.

nicate, which decreases the number of necessary states and transitions. Orthogonality allows a state's child states to be partitioned into different *regions*. When the parent state is active, each of its regions can be active as well, in parallel. This effectively implements concurrency and is the main weapon against state space explosions. Where there is concurrency, however, non-determinism usually lurks just around the corner [Lee06]. *Synchronous languages* avoid it by introducing the notion of discrete *ticks* the execution takes place in. During any given tick, each signal must have a unique value. While this solves non-determinism, it has the disadvantage of greatly restricting the set of valid synchronous programs.

SCCharts [HDM+14], developed at our research group, are a synchronous visual language built upon the *sequentially constructive model of computation* [HMA+13] to provide deterministic concurrency without restricting the set of valid programs too much. Figure 1.5 shows a simple SCChart consisting of a hierarchical state with two regions, each containing further states connected by transitions. Besides a mandatory name, each state can declare an *interface* that defines *input/output variables* used to communicate with its environment. It can also declare *local variables* to be used inside the

state, as well as *entry actions* and *exit actions*: things to do when the state becomes active or inactive.

For an active state to become inactive, the system must decide to move to another state through one of its outgoing transitions. Which of these is eligible to be taken can be constrained by imposing *conditions* on them. Of course, if multiple transitions leave a state, several of them can be eligible at the same time. To retain determinism, each transition is assigned a priority. The transition that is taken is the one with the lowest priority value among all eligible transitions. If the transition specifies an *action*, that action is executed at this point. To capture this, the text of transition labels has the following basic syntax:

```
[Priority ':'] [Condition] ['/' Action]
```

The condition and action expressions can easily become rather complex, each involving several signal and variable names. This sets off an unfortunate chain of problems: transition labels can become very long, causing transitions to become very long, causing SCCharts to become rather wide, causing them to fit on the screen only if the zoom level is significantly reduced, causing legibility to suffer.

Unlike Ptolemy II diagrams, SCCharts are not edited visually with a drag and drop editor. Instead, they are defined textually through the Textual SCCharts Language (SCT), a DSL developed specifically for that purpose. While users are working on the text file, the corresponding SCChart is dynamically drawn and updated in a graphical view displayed next to the editor, as shown in Figure 1.6. This editing approach gives users the features they are accustomed to from textual editors (including syntax highlighting and content assist) while allowing them to spot problems quickly in the graphical view that may be harder to find just by looking at the SCT code. The graphical view also serves as a navigation aid: clicking on a graphical element selects its definition in the text.

The exact content of the graphical view can be customized in several ways to address the different use cases that arise while developing software using SCCharts:

- Showing or hiding elements. Depending on the task at hand, information such as state interfaces or certain kinds of actions may not actually be of

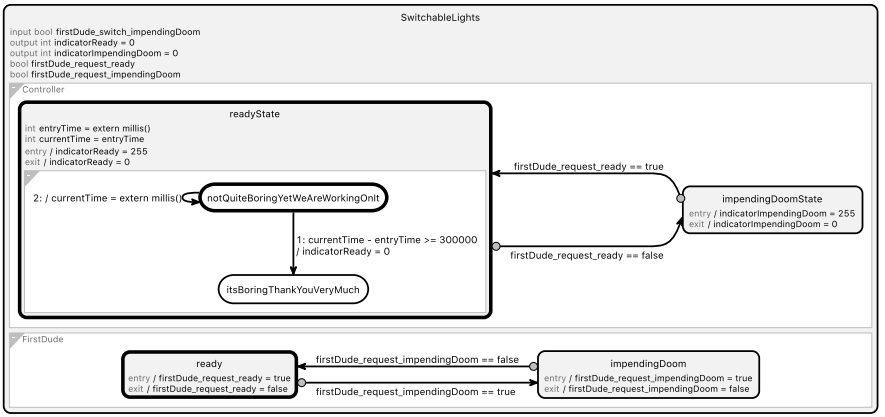


Figure 1.5. A small and simple SCChart created by the author. SCCharts always have an uppermost state (in this case called *SwitchableLights*). As the diagram suggests, they support hierarchy and concurrency through separate regions. Note how labels are used to specify the behavior of the SCChart: they define the interface of states, what happens when entering or exiting states, and when a transition is eligible to be taken as well as what happens once it is. The layout of this SCChart was generated automatically, using methods described in Section 3.3 and Chapter 6.

interest. To prevent them from cluttering up the diagram, they can be switched off.

- Changing automatic layout settings. The default layout configuration chosen by the tool developers may not always yield the best layout for a given diagram. The graphical view thus offers a selection of layout options for users to play around with. In addition, layout options can also be set in the SCT file by annotating elements accordingly.

- Showing intermediate compilation results. The SCCharts IDE provides a complete compilation chain with different compilation targets through several intermediate compilation steps [MSH14]. Instead of showing the original SCChart, the graphical view can display the intermediate result at any point in the chain. It is also possible to trace diagram elements through the process by showing which elements of the original

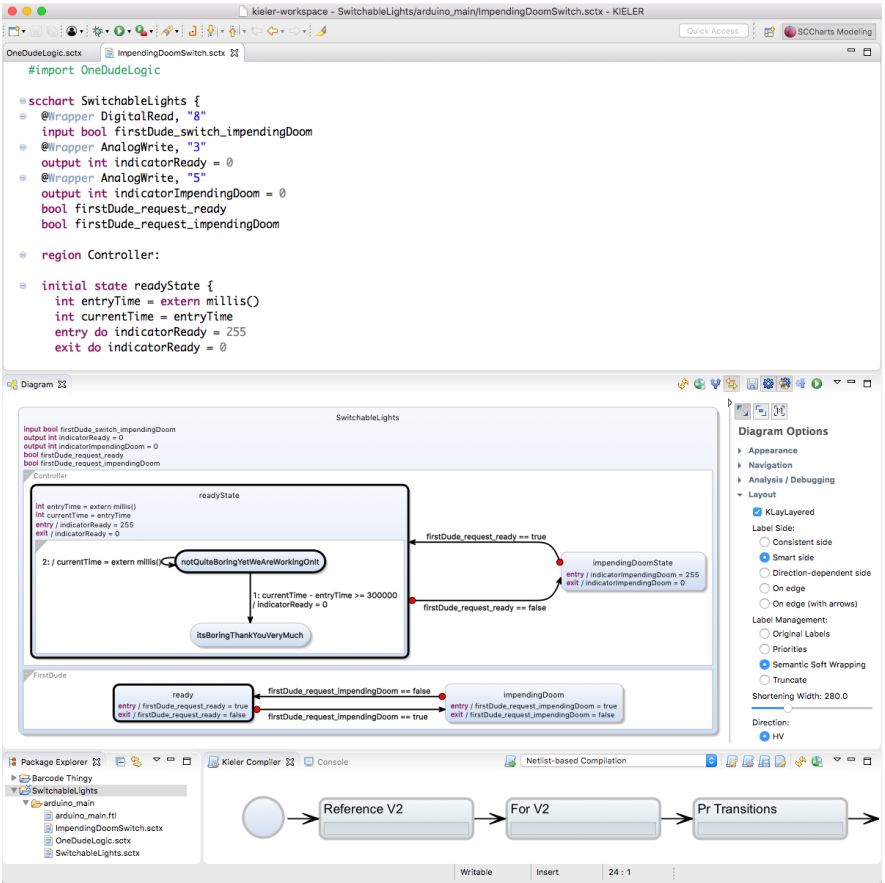


Figure 1.6. The SCChart from Figure 1.5 as shown in the KIELER SCCharts editing environment developed at our research group. The SCChart is defined through the textual SCT language in the text editor at the top. Below it is the graphical view that shows the corresponding SCChart. Note how the sidebar to the right allows the view to be customized, including options to configure the methods to be introduced in Section 3.3 and Chapter 6. At the bottom is the compiler chain which allows for intermediate results to be inspected.

SCChart ended up being compiled to which elements in the (possibly intermediate) result.

- ▻ Enriching the diagram with additional information. The compilation chain can be used by the IDE to add signal dependency [RSM+16] or execution time information [FBH15] to the diagram. Using these can help fix scheduling or performance problems.

By now we have an idea of how dynamic the graphical view is. All of its content is synthesized on the fly, which requires automatic layout algorithms since diagrams synthesized from scratch simply do not have any placement information that could be used (the main algorithm used here is ELK Layered, which we shall look at in detail in Chapter 3). To synthesize its diagrams, the graphical view uses the KIELER Lightweight Diagrams (KLighD) framework which will be introduced in more detail in Chapter 2. It is this framework into which we will integrate several of the methods proposed in this thesis.

1.1.3 UML Sequence Diagrams

The UML defines fourteen kinds of diagrams which it divides into two categories [Obj17, Annex A]: *structure diagrams* and *behavior diagrams*. Sequence diagrams belong to a sub-group of the latter type called *interaction diagrams* that describe the communication between parts of a system, each focusing on different aspects. Sequence diagrams focus on the messages exchanged between communicating parties as part of an *interaction* between them. In a study performed by Petre [Pet14], sequence diagrams ranked among the three types of UML diagrams most commonly used by professional software developers (although it has to be mentioned that this study included only 50 participants).

Figure 1.7 shows a simple sequence diagram while Figure 1.8 shows a more comprehensive overview of some of their elements. The frame around it represents the interaction described by the diagram. Inside the frame is a series of vertical lines with individual headings called *lifelines*, one for each of the interaction's participants. The lifelines are connected by arrows that represent messages being exchanged; in the realm of object-oriented

languages, these might for example be method calls. We say that a message that leaves or arrives at a lifeline is *incident* to that lifeline. Conceptually, time passes as we move downwards along a lifeline. That is, if two messages are incident to the same lifeline, the upper message is expected to precede the lower one in time. Lifelines can be created and destroyed by messages, just like objects can be created and destroyed during the execution of, say, a Java program. Usually, whenever a lifeline is "busy" sending and receiving messages, this is interpreted as specifying a particular execution of whatever the lifeline represents, for example the execution of a method on an object. Such *execution specifications* are visualized by drawing a box along the lifeline which starts at the first and ends at the last message involved. Execution specifications can be nested.

A sequence diagram can include two kinds of hierarchy. First, it can reference interactions specified in other sequence diagrams. This is called *interaction use* and is visualized by drawing a labeled box across the involved lifelines. Second, *combined fragments* (or simply *fragments*) bundle a set of messages to be operands to a specified operator. For example, a given set of messages can be surrounded by a *loop* fragment to emphasize that they are exchanged repeatedly. Combined fragments are visualized by drawing a box around the involved messages that mentions the operator in the top left corner. If the operator requires more than a single set of messages to operate on, the box is divided into different regions by dashed horizontal lines.

Sequence diagrams make use of text in several ways. First, every graphical element has a label to provide necessary details. This includes messages and headings for lifelines. Both can contribute to a sequence diagram's width, but message labels can have the additional problem of crossing other lifelines, which reduces their legibility. While not strictly part of the specification, sequence diagram editors usually also allow users to add comments to their diagrams which can either be free floating or be connected to specific elements.

Software for drawing sequence diagrams comes in two flavors: drag and drop editors (a popular example from the Eclipse world being the Papyrus[4]

[4] `https://eclipse.org/papyrus`

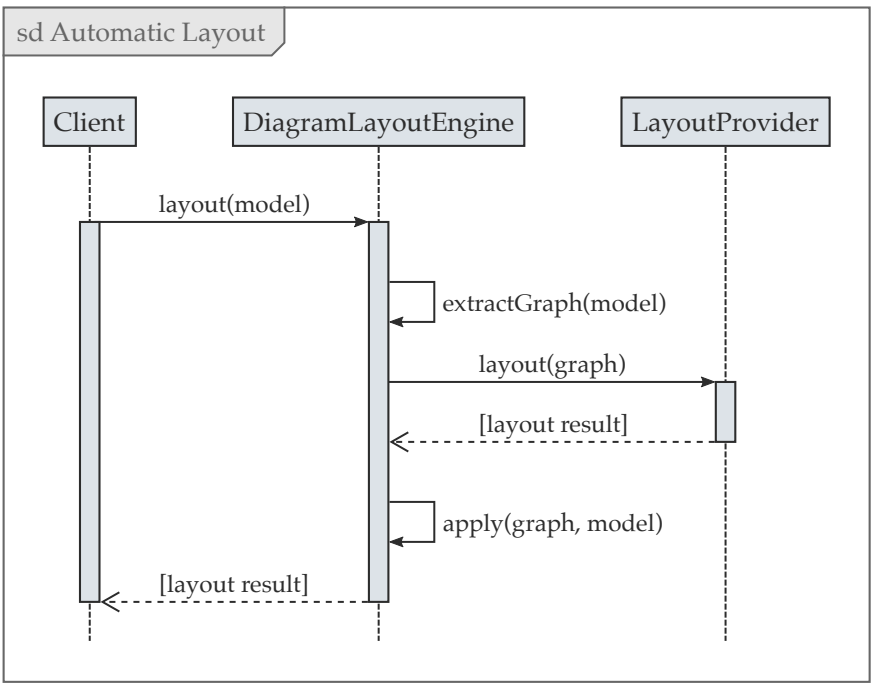


Figure 1.7. A simple example of a UML sequence diagram, laid out using the methods to be introduced in Chapter 4. This particular diagram is a simplified description of how automatic layout is invoked on and applied to a model through the layout infrastructure to be described in Section 2.5.

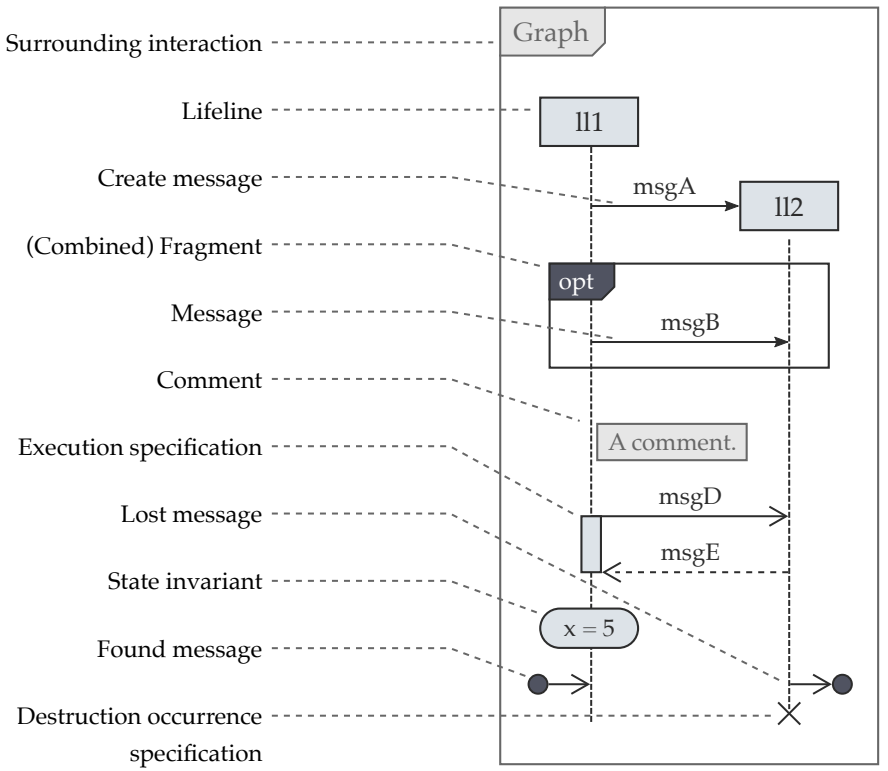


Figure 1.8. The different elements of a sequence diagram. The full specification includes yet more elements, but we constrain ourselves to this subset.

project), and textual editors based on custom DSLs with a synthesized graphical view (WebSequenceDiagrams[5] and SequenceDiagram.org[6] for the browser, Quick Sequence Diagram Editor[7] for the desktop).

Drag and drop sequence diagram editors have an interesting problem to solve: due to the comparatively rigid visual structure of the diagrams, they cannot allow users too much freedom when placing their elements. For example, they will usually want to ensure that lifelines are placed next to each other, along a horizontal line. This can blur the distinction between drag and drop editing and automatic layout: if the editing operations are constrained enough, they will assume much of the functionality that automatic layout would provide. Constraining them to this level, however, carries with it the danger of making the editor harder to understand for users by prohibiting operations they might expect to be possible based on their experience with other editors.

The Open KIELER project[8] offers a sequence diagram editor that employs the same technologies as the SCCharts editing environment (see Figure 1.9). Based on a textual language called KieSL (to be described in Section 4.3) it will serve as a demonstrator for some of the concepts to be explored in this thesis.

1.2 Principles

Now that we have encountered a number of visual languages, this seems like a good time to extract a few guiding principles that we shall observe throughout this thesis. All of them fall under the general theme of text in diagrams and will inform the contributions to be brought forth in subsequent chapters. Each will relate to at least one of these principles.

[5] https://www.websequencediagrams.com
[6] https://sequencediagram.org
[7] https://sourceforge.net/projects/sdedit
[8] https://github.com/OpenKieler

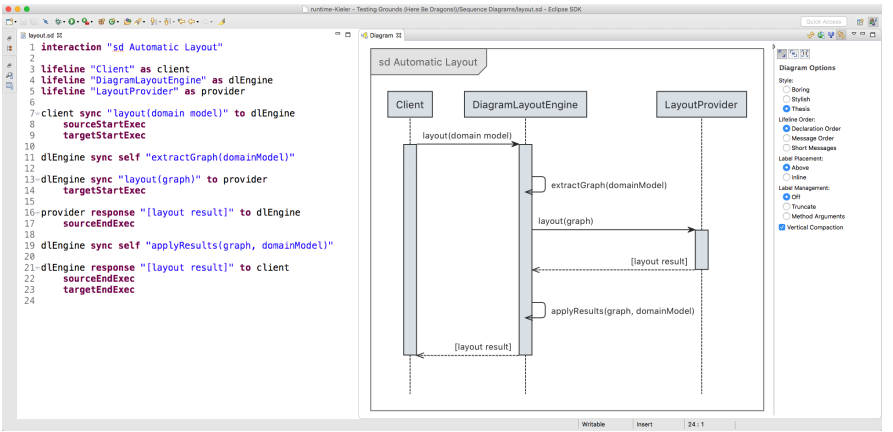


Figure 1.9. The sequence diagram from Figure 1.7 being edited in our sequence diagram editing environment. Similar to SCCharts, sequence diagrams are defined through the textual KieSL language in the text editor to the left. Right next to it is the graphical view that shows the corresponding diagram, with the sidebar to the right allowing users to switch between different display options.

P-PLACEMENT Text needs to be properly placed.

To be usable with as many visual languages as possible, layout algorithms need to support placing text in diagrams. Text can belong to different types of elements: as we have seen, Ptolemy II mainly uses it for nodes, ports, and comments while SCCharts heavily rely on it to describe edges. For layouts to be successful, it needs to be ensured that enough space is available to place text without overlaps and in a way that makes it easy to discern what it relates to.

P-SIZE Large diagrams impede readability.

A diagram growing larger will eventually set off a chain of problems. Fitting such a diagram into the available screen space means sacrificing legibility, often to the point where it is rendered useless by having important features obscured. Users are then forced to zoom and pan their way through the

diagram as they simultaneously try to keep an overview at the back of their minds. If text is the cause of the diagram's size, zooming in enough to read it means not seeing much else of the diagram, making it even harder to keep an overview and easier to get lost in the details. To keep this from happening, producing small layouts must be one of our principles.

P-NOTATION Secondary notation should not be ignored.

To quote Petre again [Pet95], "much of what contributes to the comprehensibility of a graphical representation isn't part of the formal programming notation but a 'secondary notation' of layout." If this is the case, automatic layout algorithms should do their best to preserve deliberate secondary notation present in the diagram they are set to work on. This seems especially true for applications where automatic layout is applied to diagrams that were manually laid out by users, as is the case in Ptolemy II and our KIELER Ptolemy Browser.

P-CLUTTER Too much information clutters up diagrams.

Visual languages such as SCCharts cram lots of information into a diagram. How much is actually required, however, depends on what the user is currently trying to accomplish: getting an idea of a diagram's overall structure requires less information to be shown than trying to understand parts of it in detail. Always showing everything does not recognize that this may not be ideal for a given use case, adds visual clutter, and makes it harder to keep an overview. Appropriate *filtering* is what we shall strive for.

1.3 Contributions

Working along the lines of the principles outlined in the foregoing section has yielded contributions to the four different areas discussed below. The vast majority of these were implemented and added to the ELK project, a library of graph layout algorithms which we shall learn about in greater detail in Section 2.5. Thus, this section mentions both the general contri-

butions as well as where they found their way into implementations to be used in the real world.

Flow-Based Diagrams (Chapter 3) Data flow and control flow diagrams are often laid out using the *layered approach* first introduced by Sugiyama et al. [STT81]. The original approach did not pay any attention to labels, a drawback I set out to remedy.

I introduce the *cell system* to compute the size of nodes required to properly place their node and port labels, and then actually place the labels subject to a range of different placement options. This addresses the **P-PLACEMENT** principle.

Turning to edge labels, I consider different strategies for their placement that address both the **P-PLACEMENT** and **P-SIZE** principles. The first set of strategies aims at deciding where alongside an edge its labels should be placed, whereas the second set is all about deciding on which of the edge's sides its labels should end up. I also introduce strategies to make the direction of an edge obvious without requiring users to look at its end points. I evaluate all strategies based on aesthetic criteria and investigate label side selection strategies further through a controlled user study as well as two surveys among users.

The cell system is now available to layout algorithms in ELK and is used, among others, by the ELK Layered algorithm [SSH14], an existing implementation of the layered approach to be covered in subsequent chapters. The edge label placement strategies have been added to ELK Layered as well, and some of the label side selection strategies were also implemented in the ELK Sequence algorithm, a layout algorithm for UML sequence diagrams.

UML Sequence Diagrams (Chapter 4) UML sequence diagrams have a rigid visual structure, which makes it seem like they do not constitute a very interesting layout problem. I show that this assumption is wrong once one exploits the few degrees of freedom sequence diagrams do allow for.

In a small case study, I argue that the graph structure provided by ELK is flexible enough to represent diagrams that are not simple hierarchies of nodes and edges, which is the case with sequence diagrams.

I then introduce ELK Sequence, a layout algorithm for sequence diagrams which differs from other algorithms in that it can reorder lifelines according to different criteria as well as compute layouts that result in diagrams with smaller height, thus addressing the **P-SIZE** principle. I evaluate ELK Sequence based on aesthetic criteria.

ELK Sequence is available as part of ELK. It acts as the foundation of a sequence diagram editor which is part of the Open KIELER open source project and uses a textual language called KieSL, briefly described in Section 4.3.

Comment Attachment (Chapter 5) As mentioned before, layout algorithms are usually not aware of secondary notation. As a case in point, automatically laying out Ptolemy II block diagrams that contain comments will often destroy any implicit relationships between them and the elements they refer to which were clear previously through clues such as proximity.

To address the **P-NOTATION** principle, I introduce a *comment attachment pipeline* that aims at inferring implicit relations between comments and diagram elements so that layout algorithms can then react properly. The pipeline requires ways to measure the likelihood of a comment referring to a given diagram element, so I introduce and evaluate a number of such measures. The success of inferring attachments is evaluated in an experiment.

An implementation of the pipeline is part of ELK as a general comment attachment framework to be used as a preprocessing step before running automatic layout. The framework is used in the KIELER Ptolemy Browser, available as part of the Open KIELER project.

Label Management (Chapter 6) Visual languages can suffer from the problem of text becoming so long that it enlarges diagrams considerably. Fuhrmann [Fuh11] proposed *label management* as a solution which dynamically changes the text in a diagram.

I describe a number of label management strategies, both of a general nature as well as specific to SCCharts, to address both the **P-SIZE** and **P-CLUTTER** principles. I then introduce a *label management framework* that allows label management to be performed either before or during auto-

matic layout. I also discuss user interface design issues of integrating label management into an application.

I evaluate label management in general based on three case studies. I also evaluate different label management strategies implemented in the SCCharts editing environment through several surveys among its users.

Label management was successfully implemented and ships with the KIELER Ptolemy Browser, SCCharts, and our KieSL editor. More generally, the label management framework is available as part of the ELK and KIELER Lightweight Diagrams (KLighD) open source projects, and the ELK Layered and ELK Sequence algorithms have been changed to support label management.

1.4 Related Work

As Frederick Brooks wrote in 1987, "a favorite subject for Ph.D. dissertations in software engineering is graphical, or visual, programming, the application of computer graphics to software design." [Bro87, page 10]. Thankfully, this does not just apply to Ph.D. dissertations, so let us review publications related to what we have set out to discuss in this thesis.

Label Placement

Label placement in general has a long history in cartography. In a classic paper [Imh75], Imhof described six principles for good map labeling, which Kakoulis and Tollis distilled down to three rules for edge label placement [KT97]. They also provided a definition of the *edge label placement problem*, which is about placing edge labels in diagrams whose other elements have already been placed. Existing algorithms, of which they provided an overview [Tam13, Chapter 15], usually run the risk of producing overlaps with other diagram elements, yielding placements where it is unclear which label belongs to which element, or hiding or at least scaling down labels to avoid violations—all undesirable for visual programming languages.

I consider label placement a part of automatic layout, thereby ensuring that there will always be enough space available to remedy these

problems. There are algorithms that follow the same approach. Klau and Mutzel [KM99] for instance integrated node label placement into the topology-shape-metrics approach to graph drawing, although they placed labels always outside of nodes and the results they showed did not always seem to produce clear placements. The *Graphviz dot*[9] algorithm, an implementation of the layered approach, provides a rather extensive configuration for node labels that includes rendering information. Since our layout algorithms do not include rendering, I am not interested in such flexibility, but introduce configurability regarding how a node's size is computed and how its labels are to be placed.

Graphviz dot [GKN+93] handles edge labels by introducing dummy nodes, as did Castelló et al. [CMT01] in their statechart layout algorithm based on the layered approach; this is an approach I follow as well. They did not, however, describe any strategies regarding where edge labels end up with regard to their edge. Castelló et al. [CMT01] placed labels on edges, which is one of our label placement strategies as well, but neither did they discuss graphical design considerations nor did they evaluate whether doing so may have a negative impact on the ability of users to read the resulting drawing.

There have been more radical proposals, most notably by Wong et al., who replaced the edges themselves by their labels [WMP+05]. That approach would not work with long edges or orthogonal edge routing, but the on-edge label placement strategy to be introduced in Section 3.3.4 could be interpreted as a less extreme version of this technique.

There have been investigations into how edge direction can be communicated. Xu et al. [XRP+12] found that doing so through curvature is inferior to using straight arrows. Holten and van Wijk [HW09] additionally investigated methods such as changing edge thickness or color from tail to head. While methods like these can be successful in small graph drawings, our applications include diagrams too large to fit on a single screen. Since edges can grow rather long in such diagrams, it would be hard for users to spot the subtle gradients that would be visible in a limited excerpt of an edge currently displayed on screen. In a subsequent paper, Holten

[9] http://www.graphviz.org

et al. [HIW+11] also investigated animating edges to indicate direction and rendering them as a sequence of arrows. While both can be valid solutions, they have two drawbacks. First, they arguably increase visual clutter more than the methods to be described in this thesis do. Second, they require the rendering of edges to be changed, which may not always be possible. For example, *LabVIEW* (National Instruments) distinguishes different types of edges through their rendering. And finally, animations simply do not work in print, which should not be excluded as a use case (at least not yet). My approaches for indicating edge direction do not change an edge's rendering, but indicate its direction either through the label's placement side or through additional label decorators.

Sequence Diagram Layout

As opposed to UML class diagrams, sequence diagrams have received comparatively little attention from researchers. A paper by Wong and Sun [WS05] on desirable aesthetics of class and sequence diagrams is a case in point: while they referenced four papers just on the layout of class diagrams, sequence diagrams were represented by only two papers, and they did not even describe layout algorithms, but merely general aesthetics.

This is surprising considering that in a study on the use of UML among 50 professional software developers [Pet13], Petre found sequence diagrams to be among the top three most commonly used diagrams (along with activity diagrams and, of course, class diagrams). One reason for this situation might be that compared to class diagrams, sequence diagrams offer rather less freedom when it comes to their layout.

Nevertheless, although I am not aware of published layout algorithms, some have at least been developed. Bennett et al. [BMS+08] have implemented a sequence diagram viewer as part of a tool used in a study to determine which features were important to users. They did not, however, publish any details on the algorithm.

There are several sequence diagram editors available that turn a DSL-based specification into a diagram. Examples are the *Quick Sequence Diagram Editor*, *WebSequenceDiagrams*, and *SequenceDiagram.org*. The first provides a rather cryptic syntax and only distinguishes between synchronous and

asynchronous messages. The other two use more or less cryptic operators that are supposed to resemble different types of arrows to distinguish between message types. The language to be described in Section 4.3 was designed to make the textual representation easy to understand without prior knowledge. None of their layout algorithms changes the order of lifelines to optimize for any aesthetic criteria, or allows messages to share vertical coordinates to reduce the height of the diagram. Both are features of the layout algorithm that I describe in detail in Section 4.2.

Poranen et al. [PMN03] described and partly formalized aesthetic criteria they believed to be desirable for the layout of sequence diagrams. Wong and Sun [WS05] picked up on those criteria and justified them with principles from perceptual theories. I introduce the relevant criteria in Section 2.2.2.

Comment Attachment

The problem that comment attachment has to solve is to infer which diagram element, if any, a comment refers to. So far, I am not aware of any studies on how to do so, or even on how developers use and place comments in visual languages in the first place. Looking at textual code documentation systems hardly helps. Javadoc, for example, was explicitly designed to make it obvious which semantical element a comment describes [Kra99], but that renders any attempts at inferring such relations superfluous and thus does not advance our cause.

Knowing about relations between comments and diagram elements enables reasoning about them. Staying with the example of Javadoc, researchers have used such information to infer unit tests for methods from comments [TMT+12] and to generate documentation about conditions that cause exceptions to be thrown by a method [BW08]. Comment attachment might pave the way towards similar opportunities for visual languages.

Comment attachment requires knowledge about secondary notation, a concept introduced by Green and Petre [GP96; Pet06]. Sadly, there has not been much work on characterizing or even measuring it. Thus, the heuristics I use to characterize comment usage are derived in part from our experience of looking at diagrams. Heuristics such as proximity, however, are rooted in Gestalt principles, a perceptual theory introduced by Wertheimer [Wer23].

Heuristics based on alignment or font size are informed by established principles in graphic design.

Label Management

Label management goes back to the concept of the *view* as a *presentation filter* in the well-known model-view-controller paradigm [Ree79]. Label management itself was to the best of our knowledge first proposed by Fuhrmann [Fuh11], who described it as one important part of filtering views. He introduced the concept of a *label manager* and the need for it to be customizable to specific languages. He also described several such label managers, which I will extend and add to in Section 6.1. I will build on this foundation by presenting a label management framework, including ways to integrate label management into automatic layout, and by evaluating the concept through case studies and user surveys.

Several publications have recognized the "effective use of screen real estate" [BBB+95] as a challenge for visual languages. The infamous "Deutsch Limit," named after Peter Deutsch who it allegedly goes back to, asserts that "the problem with visual programming is that you can't have more than 50 visual primitives on the screen at the same time." [Beg96, Section 2.1.3]. Cockburn et al. [CKB09] called attention to the fact that the usual zooming and panning that ensues when working with larger diagrams causes what they called *temporal separation*. Brooks recognized this problem as well, going so far as to claim that "nothing even convincing, much less exciting, has yet emerged from [graphical programming]. I am persuaded that nothing will." [Bro87, page 10] (the world seems to have disagreed, though). With the increased size of diagrams being a problem for the ability of users to understand them [Stö14], label management is one approach to make large diagrams less unwieldy.

Over the years, different other methods have been proposed to that effect. Optically distorting fisheye views [LA94] show areas of interest with increased magnification and possibly additional guidelines to aid comprehension [SLP+13], with results that mimic the classic fisheye lens effect. Applied to our use cases, the distortion would negatively impact legibility. Graphical fisheye views [SB92] remove the distortion by magnifying the

nodes themselves and applying automatic layout to keep nodes, edges, and labels straight. In contrast, label management changes the text of labels instead of applying any kind of magnification. Both techniques might well be combined, though.

Been et al. [BDY06] applied label filtering concepts to dynamic map exploration. As the user zooms in and out, labels are shown and hidden depending on the space available for them. Especially if combined with a notion of a label's importance, this approach seems to work very well for map exploration. However, it causes a label to be either completely visible or completely hidden. Label management changes a label's actual content, and with it its size, to allow more diagram elements to fit on the screen. This can include, but is not limited to, hiding a label altogether.

Musial and Jacobs used a similar idea and applied filtering concepts to UML diagrams [MJ03]. They gradually reduced the amount of details classes were visualized with as their graph-theoretical distance increased to classes the user focusses on. While this does change the diagram's layout to show more classes to the user, labels are again either completely shown or hidden, but do not have their level of detail changed. Also, Musial and Jacobs concentrated on modifying node labels based purely on focus and context information. I focus on edge labels and comments and investigate ways to take layout information into account when making label management decisions.

Castelló et al. [CMT01] restricted labels in statecharts to a constant fixed width. They divided labels into their three semantical components, each put on a different line. The components were then wrapped such that they did not exceed the fixed width. I have implemented a similar strategy in our SCCharts editing environment, but it is not limited to a fixed width. Also, Castelló et al. did not generalize their approach to arbitrary use cases.

Regarding the decision of how much detail an element should be shown with, *focus and context* is an important concept [CMS99]. Herein, the diagram elements are divided into the set of elements the user is currently focussing on and surrounding elements that provide context. I show how label management decisions can be made based on this method. Osman et al. [OCP14] described a method to automatically infer the importance of elements in

reverse-engineered UML diagrams to show them in appropriate levels of detail. Such approaches could well be used to guide label management.

1.5 Publications

Parts of each chapter of this thesis were published at conferences during my time as a PhD student. Some also build upon work done by students under my supervision.

Relevant Publications

Out of 17 peer-reviewed publications I have been involved in thus far, the following are the ones most relevant to this thesis.

[SH14] 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, pp. 41–44. DOI: 10.1109/VLHCC.2014.6883019

The first paper on comment attachment (Chapter 5) was the first foray into this topic, using only a single heuristic to make attachment decisions. It received the conference's "Best Short Paper" award.

[SPH16a] Christoph Daniel Schulze, Christina Plöger, and Reinhard von Hanxleden. "On comments in visual languages". In: *Proceedings of the 9th International Conference on the Theory and Application of Diagrams (DIAGRAMS '16)*. LNCS. Springer, 2016, pp. 219–225. ISBN: 978-3-319-42333-3. DOI: 10.1007/978-3-319-42333-3_17

The sequel to the first paper on comment attachment evaluated more heuristics for making attachment decisions and improved the success rate significantly.

An extended version of this paper, which goes into more details regarding the heuristics and the attachment pipeline to be introduced in Section 5.2, is available as a technical report [SPH16b].

[SLH16] Christoph Daniel Schulze, Yella Lasch, and Reinhard von Hanxleden. "Label management: keeping complex diagrams usable". In: *Proceedings of the*

IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '16). Sept. 2016, pp. 3–11. DOI: 10.1109/VLHCC.2016.7739657

Building upon ideas first introduced by Fuhrmann [Fuh11], this paper explores the concept and implementation of label management (Chapter 6).

[Sch16b] Christoph Daniel Schulze. "Two opportunities and challenges of automatic layout in visual languages". In: *Proceedings of the ACM Student Research Competition at MODELS 2016 co-located with the 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*. 2016

This motivates thinking about automatic layout in terms of *challenges* and *opportunities*, illustrated using comment attachment (Chapter 5) and label management (Chapter 6) as examples.

[SWH18a] Christoph Daniel Schulze, Nis Wechselberg, and Reinhard von Hanxleden. "Edge label placement in layered graph drawing". In: *Proceedings of the 10th International Conference on the Theory and Application of Diagrams (DIAGRAMS '18)*. LNCS. Springer, 2018, pp. 71–78. ISBN: 978-3-319-91376-6. DOI: 10.1007/978-3-319-91376-6_10

This paper introduces some of the layer and side selection strategies for edge labels (Section 3.3) and summarizes our evaluations of those strategies.

An extended version, which has more details on the selection strategies and also discusses optimal-width layer assignment, is available as a technical report [SWH18b].

[SHH18a] Christoph Daniel Schulze, Gregor Hoops, and Reinhard von Hanxleden. "Automatic layout and label management for UML sequence diagrams". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '18)*. 2018

This paper introduces ELK Sequence, the layout algorithm for UML sequence diagrams, and what it can do to compact these diagrams (Chapter 4).

An extended version of this paper, which contains a more complete description of ELK Sequence, is available as a technical report [SHH18b].

Relevant Advised Theses

Out of 20 theses I have advised thus far, the following are the ones most relevant to this dissertation:

[Car12] John Julian Carstens. "Node and label placement in a layered layout algorithm". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jjc-mt.pdf. Master's thesis. Kiel University, Department of Computer Science, Sept. 2012

This thesis took a first look at proper label placement in the layered approach. Node and port label placement have since been considerably improved and expanded (Section 3.2), as has edge label placement (Section 3.3).

[Hoo13] Gregor Hoops. "Automatic layout of UML sequence diagrams". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/grh-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, Apr. 2013

The first incarnation of the sequence diagram layout algorithm (Section 4.2) was developed as part of this thesis.

[Jah15] Daniel Jahn. "Eine textuelle Sprache zum automatischen Generieren von Sequenzdiagrammen". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/dja-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2015

The textual sequence diagram language and an initial version of the accompanying visualization (Section 4.3) were first developed in this thesis.

[Plö15] Christina Plöger. "Improving comment attachment algorithms". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cpl-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2015

After the first paper on comment attachment [SH14], this thesis was about developing and evaluating additional attachment heuristics. Since then, a complete attachment framework was developed and integrated into the KIELER Ptolemy Browser.

[Las15] Yella Lasch. “Label Management in Graph Layout Algorithmen”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ybl-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2015

This thesis provided first implementations and evaluations of label management strategies (Chapter 6). The label management framework has since been completely re-implemented and expanded.

1.6 Outline

This thesis is structured as follows. We shall begin with foundations in Chapter 2, including definitions as well as an overview of the technical landscape that we shall be dwelling in.

Part I will be all about computing automatic layouts with a focus on the **P-PLACEMENT** and **P-SIZE** principles. We will start with flow-based diagrams in Chapter 3 (although some of the approaches here are applicable to other types of diagrams as well) before turning towards UML sequence diagrams in Chapter 4.

Part II, then, will build on automatic layout and investigate challenges it might pose in terms of the **P-NOTATION** principle (comment attachment, in Chapter 5) as well as opportunities it might enable regarding the **P-SIZE** and **P-CLUTTER** principles (label management, in Chapter 6).

Not keen on breaking with tradition, we close with a conclusion and open problems in Chapter 7.

Chapter 2

Foundations

Before diving into the details starting with Part I, we should spend a bit of time to familiarize ourselves with the necessary foundations. First up are basic graph theoretical definitions in Section 2.1, which we will be using—and occasionally be adding to—throughout this thesis. These are followed by a discussion of aesthetic criteria in Section 2.2 that give us objective measures for the quality of a graph drawing. For the rest of the chapter we will work through the technical environment that the methods and algorithms were implemented in. Since almost everything is implemented upon or makes use of the *Eclipse* platform in some way, it makes sense to introduce that first in Section 2.4. The Eclipse Layout Kernel (ELK), while technically part of the Eclipse project, deserves its own section since this is where the layout algorithms described in Part I found their home. Finally, an introduction to the KIELER Lightweight Diagrams (KLighD) framework will describe how textual descriptions are turned into diagrams, which will serve as a foundation for understanding how the interactive techniques introduced in Part II were implemented.

2.1 Basic Terminology

Figure 2.1 shows a drawing of a hierarchical graph, along with some of the terminology to be introduced in this section.

Definition 2.1 (Undirected graph)**.** An *undirected graph* is a tuple $G = (V, E)$ of a finite set of *nodes* (or *vertices*) V and a set of unordered pairs $E \subseteq V \times V$ called *edges*. An edge $(v_0, v_1) \in E$ is said to be *incident* to v_0 and v_1, and the two nodes are said to be *adjacent*. If $v_0 = v_1$, the edge is called a *self-loop*. □

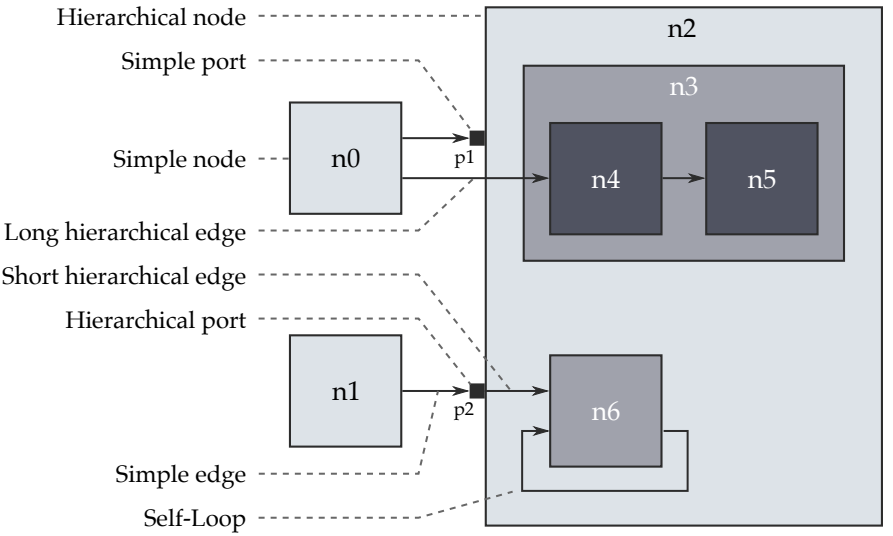


Figure 2.1. Drawing of a directed hierarchical graph with ports along with some of the terminology that goes with it. It might not be obvious that the edge from *n2* to *n6* is a hierarchical edge. This does make sense, however: after all, the two nodes have different parents. See Figure 2.2 for the inclusion tree of this graph.

Definition 2.2 (Directed graph)**.** A *directed graph* is a tuple $G = (V, E)$ of a finite set of *nodes* (or *vertices*) V and a set of ordered pairs $E \subseteq V \times V$ called *edges*. For $e = (v_0, v_1) \in E$, we call v_0 the *source* of e ($\text{source}(e) := v_0$) and v_1 the *target* of e ($\text{target}(e) := v_1$). We call e an *outgoing edge* of v_0 and an *incoming edge* of v_1. For $v \in V$, the *predecessors* of v are $\{v' \in V : (v', v) \in E\}$ and its *successors* are $\{v' \in V : (v, v') \in E\}$. □

Definition 2.3 (Induced subgraph)**.** Given a graph $G = (V, E)$ and a non-empty subset $N \subseteq V$, we call $G_N = (N, E_N)$ with

$$E_N = \{(v_1, v_2) \in E : v_1 \in N \wedge v_2 \in N\}$$

the *subgraph of G induced by* E_N. □

Definition 2.4 (Path)**.** Let $G = (V, E)$ be a directed graph. A sequence of nodes $v_1, \ldots, v_k$ is called a *directed path* if $(v_i, v_{i+1}) \in E$ for $1 \leqslant i \leqslant k - 1$. It

is called an *undirected path* if $(v_i, v_{i+1}) \in E \vee (v_{i+1}, v_i) \in E$ for $1 \leqslant i \leqslant k-1$. The path is called *simple* if all nodes are distinct. □

So far, edges connect directly to nodes. Let us consider an electrical circuit, such as a graphics card in a computer. Its microprocessors expose a number of pins for lines (or wires) to connect to. Since each pin leads to different parts of the microprocessor's internal circuit, the exact pin a line connects to makes a substantial difference. If we were to model such a circuit, we would thus need a way to model the pins. As introduced by Gansner et al. [GKN+93] and greatly expanded upon by Spönemann et al. [SFH+10], we will call them *ports*.

Definition 2.5 (Graph with ports)**.** A *graph with ports* is a tuple $G = (V, P, \rho, E)$ of a finite set of *nodes* V, a finite set of *ports* P, a function $\rho : P \to V$ which assigns each port to a node, and a set of ordered pairs $E \subseteq (V \cup P)^2$ called *edges*. □

The drawings throughout this thesis will show ports as filled black rectangles, as in Figure 2.1.

From time to time, we may need the ability to connect more than a single source to more than a single target, or to have an edge appear multiple times in a graph.

Definition 2.6 (Directed hypergraph)**.** A *directed hypergraph* is a tuple $G = (V, E)$ of a finite set of *nodes* V and a set of edges $E \subseteq (\mathcal{P}(V) \setminus \varnothing)^2$. □

Definition 2.7 (Multigraph)**.** A *multigraph* is a tuple $G = (V, E)$ which is a directed or undirected graph, but where E is a multiset. □

In the diagrams we have seen in Chapter 1, nodes may well contain child nodes. The next definition captures this fact by establishing parent-child relationships.

Definition 2.8 (Hierarchy function)**.** Given a graph $G = (V, E)$, the *hierarchy function* $\tau : V \to V \cup \{\top\}$ maps each node $v \in V$ to its parent node or to $\top$ if it does not have a parent node. For τ to be valid, we require that there is no sequence of nodes $v_1, \ldots, v_n$ with $\tau(v_{i+1}) = v_i$, $1 \leqslant i < n$, and $\tau(v_1) = v_n$.

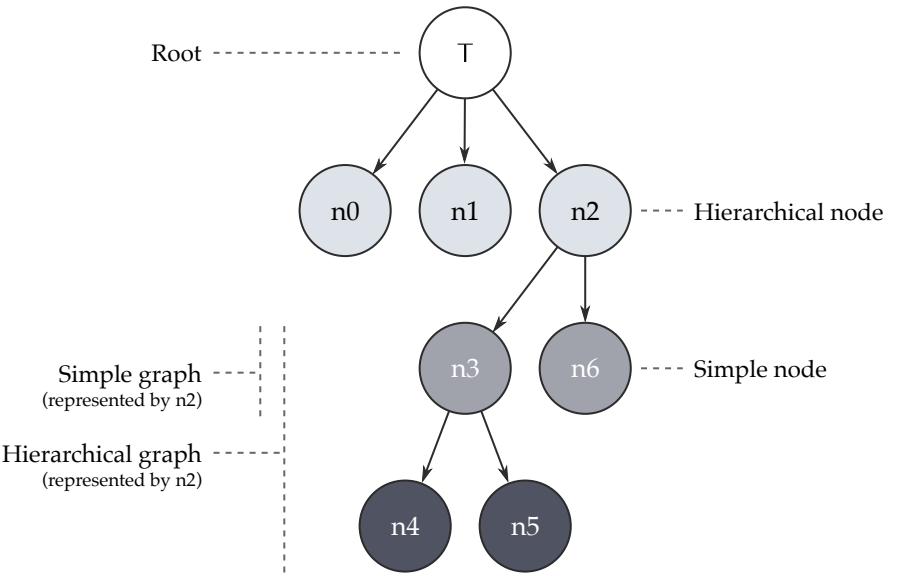


Figure 2.2. The inclusion tree of the graph shown in Figure 2.1. Each node in the original graph is represented by a node in the inclusion tree. An additional node represents the tree's root, which is the parent of the graph's top-level nodes.

A graph extended with a hierarchy function is called a *hierarchical graph*. The directed graph $T = (V \cup \{\top\}, I)$ with

$$(v_1, v_2) \in I \quad \Leftrightarrow \quad \tau(v_2) = v_1$$

is called the *inclusion tree* of G with regard to τ. □

Figure 2.2 shows the inclusion tree that belongs to the graph in Figure 2.1.

Definition 2.9 (Terminology in hierarchical graphs)**.** Given a hierarchical graph $G = (V, E)$ with hierarchy function τ, we use the following terms.

▷ A node $v \in V$ is called a *hierarchical node* (or *compound node*) if $\tau^{-1}(v)$ is not empty. Otherwise, v is called a *simple node*.

- ▷ Given a hierarchical node $v \in V$, the nodes $\tau^{-1}(v)$ are called the *children of* v, while v is called their *parent*.

- ▷ Given a hierarchical node $v \in V$, the subgraph of G induced by the children of v is called the *simple graph represented by* v. The subgraph of G induced by the descendants of v is called the *hierarchical graph represented by* v.

- ▷ Let $e \in E$ and V_e be the set of the nodes it is incident to, either directly or through ports. If $\exists v_1 \neq v_2 \in V_e$ such that $\tau(v_1) \neq \tau(v_2)$, then e is called a *hierarchical edge*. If $\tau(v_1) = v_2$ (or vice versa), e is called a *short hierarchical edge*, otherwise a *long hierarchical edge*. If e is not a hierarchical edge, it is called a *simple edge*.

- ▷ In a graph with a set of ports P, a port $p \in P$ is called a *hierarchical port* if it is the end point of at least one hierarchical edge. Otherwise, p is called a *simple port*. □

2.2 Aesthetics

There are endless ways of laying out any given graph, but not all of them are equally good. Quite the opposite, in fact: most layouts will not be helpful, often to the point of obfuscating the diagram. While layout algorithms usually produce layouts that are at least readable (in the sense of "being able to discern all elements"), that does not mean that the results are good. To be able to assess the quality of layouts and thereby get an idea of the kinds of results we desire from layout algorithms, we need measures: more or less formal criteria that we can optimize for. Put differently, we need a proper working definition of "good".

That finding such a definition might be a challenging enterprise to say the least becomes obvious by realizing how many different types of diagrams we have already seen so far: SCCharts (Figure 1.5) look completely different than sequence diagrams (Figure 1.7), and the inclusion tree graph we saw (Figure 2.2) seems to have been laid out according to entirely different layout goals still.

Even if a diagram is optimized for a certain set of aesthetic criteria, that does not mean that the diagram succeeds at getting its semantic content across to viewers. Different types of diagrams will require emphasizing different aesthetics. For the remainder of this section, we will discuss commonly used aesthetic criteria before concentrating on the more specialized aesthetics of sequence diagrams.

2.2.1 Common Aesthetic Criteria

Common aesthetic criteria, including Gestalt principles as they apply to graph drawings, are neatly summarized by Bennett et al. [BRS+07], but here we shall hone in merely on those that will become important in subsequent chapters.

One of the most commonly cited criteria is the *number of edge crossings* (meaning crossings between two edges). While being "by far the most agreed-upon edge placement heuristic," as Bennett et al. put it, the effect of minimizing the number of edge crossings has been challenged when it comes to large diagrams [KPS14].

If two edges cross, their *crossing angle* can determine how well viewers can follow either edge through the crossing [WPC+02]. In general, more perpendicular angles seem to be preferable.

The kinds of angles that can appear in a drawing are determined by several factors, one of which is the *routing style*. In this thesis, we distinguish between *polyline edges* formed by a sequence of straight edge segments, *orthogonal edges* which restrict those edge segments to be either horizontal or vertical (also called *Manhattan routing*), and *spline routing*. There often exist routing conventions for different diagram types; data flow diagrams, for example, largely use orthogonal edges.

Grounded in perceptual grouping, the second criterion is the *number of edge bends*, which should be minimized. The principle of continuation suggests that it is easier to follow a straight edge than it is to follow an edge split by bend points into several segments, especially if those segments join at right angles.

If a graph is directed, which is the case for all of our three example languages, a *consistent flow direction* may be of importance [Pur02]. In Ptolemy II

block diagrams, for example, this results in diagrams that emphasize the flow of data starting its journey at the diagram's left side, passing rightwards through the nodes as it is processed to produce results at the diagram's right side.

Edge length is another popular aesthetic, but it is less obvious what to optimize for, exactly. Three commonly cited possibilities are to minimize the sum of all edge lengths, to minimize the maximum edge length, or to try and keep edge lengths as uniform as possible. It is not clear what to go for, although it seems sensible to assume that it is generally easier to follow a short edge than it is to follow a long edge through a diagram.

Closely related to edge length is the overall *diagram size* (or *area*). In general, smaller diagrams seem preferable, as for example Störrle demonstrated for UML diagrams [Stö14]. As another case in point, diagrams for engine controllers in the automotive industry often grow so large that developers often go through the trouble of introducing additional levels of hierarchy just to end up with diagrams small enough to be printed on A4 paper.[1]

Even if a diagram's area is small, however, its *aspect ratio* (calculated by dividing its width by its height) can make it ill-suited for being displayed in the area available to it, or even be unpleasant to look at [TR05] (although the latter appears to be a somewhat diffuse concept). In this thesis, we focus on the former criterion, for along with the size the aspect ratio determines the zoom level a diagram can be displayed with to still fit into a prescribed drawing area. This concept is captured in the following definitions:

Definition 2.10 (Maximum and full-width scale)**.** Given the width and height of a drawing area $w_a, h_a \in \mathbb{R}$ and of a drawing $w_d, h_d \in \mathbb{R}$, the *maximum scale* of the drawing with respect to the drawing area is defined as follows:

$$\min\left\{\frac{w_a}{w_d}, \frac{h_a}{h_d}\right\}.$$

The *full-width scale* of the drawing with respect to the drawing area is defined as follows:

$$\frac{w_a}{w_d}.$$

□

[1] This observation was related to me in personal communication with a manager of a company which produces software for the automotive industry.

While the maximum scale, already defined by Rüegg [Rüe18], fits the whole drawing into the available area, the full-width scale describes how large the drawing must be to fit into the width of the area without regard for its height.

Scalings will become particularly important in Chapter 6 when we will compare different approaches in terms of the maximum and full-width scales they are able to achieve. We will do so based on the following definition:

Definition 2.11 (Scale factor)**.** Given a *reference scale* s_r and a *comparison scale* s_c, the *scale factor* of s_c with respect to s_r is defined as

$$\frac{s_c}{s_r}.$$ □

The reference scale will usually come from a diagram laid out with previous methods, while the comparison scale will usually be derived from the same diagram laid out with methods introduced in this thesis. Our aim, of course, will be to achieve scale factors larger than 1.

Note that if we compute scaling factors of full-width scales, the scale factor simplifies to dividing the reference drawing's width by the comparison drawing's width.

Obviously, some aesthetics contradict others. Avoiding a crossing between two edges, for example, may require one of them to take a detour through the diagram, increasing both its length and the number of its bend points. Also, maximizing the number of straight edges in, say, Ptolemy II diagrams tends to come at the cost of increasing a diagram's height [BK02]. The actual selection of aesthetics to optimize for should follow the requirements of what a particular type of diagram is supposed to communicate. If the flow of data is essential, for instance, it may be best to have consistent flow direction take precedence over other aesthetics it conflicts with.

Many aesthetics do not address secondary notation. The number of edge crossings, for example, will arguably capture less extra meaning beyond the pure semantics of the diagram than aesthetics such as *symmetrical node placement*. Chapter 5 will return to this topic when we evaluate how well different measures capture relations between comments and the diagram elements they refer to.

2.2.2 Aesthetics of Sequence Diagrams

Since sequence diagrams look entirely different from standard node-link diagrams such as Ptolemy II block diagrams or SCCharts, commonly accepted aesthetic criteria differ in applicability, as noted by Poranen et al. [PMN03]. Edge crossings, for example, are not quite that, crossings between two edges, but crossings between edges and lifelines, or rather: messages and lifelines. As per the UML standard [Obj17], edges are always straight and are usually drawn horizontally. The length of an edge is closely related to the number of lifelines it crosses, and is even exactly proportional if lifelines are evenly spaced. A sequence diagram's size and aspect ratio are mostly functions of its structure, but its height in particular can be variable, as we shall see in Section 4.2. (In this point I disagree with Poranen et al., who consider the size of a sequence diagram to follow entirely from its structure.)

It is not just that existing aesthetics must be adapted to sequence diagrams or do not apply in the first place, it is also that additional criteria may be called for. Poranen et al. introduce three such criteria: *subset separation*, the *number of long edges*, and *slidability* (which they call *sliding*).

Subset separation prefers the set of lifelines to be partitioned into subsets placed next to each other such that most messages connect lifelines that are part of the same subset. This is similar to the Gestalt principle of proximity, which layout methods such as the force-directed approach are naturally good at, at least provided that many edges run inside the subsets and only few between.

Minimizing the number of long edges is one way that optimizing for the common edge length criterion can be interpreted. It thus seems debatable whether this is really a new aesthetic specific to sequence diagrams.

The opposite is true for slidability. To get an idea of this criterion, imagine browsing through a sequence diagram zoomed in enough for it to not fit on the screen anymore. Reading the diagram requires the viewport to be moved downwards. Slidability is maximized if the end points of all messages visible in the viewport are themselves visible in the viewport as well. If this is not the case for all messages, the viewport needs to be moved sideways to see which lifelines a message connects. In a way, slidability can

be understood as being negatively correlated with the number of times the viewport needs to be moved sideways.

Three last aesthetics are ones which Poranen et al. even elevate to constraints each sequence diagram must satisfy. The *starting object* criterion requires that the lifeline which initiates the interaction be the leftmost lifeline in the sequence diagram. The *vertical distance* criterion stipulates that the vertical distance between messages should be uniform. The *horizontal distance* criterion, finally, demands the horizontal space between adjacent lifelines to be uniform, although it is not clear whether the authors mean the distance between the lifelines themselves or the distance between the lifeline headers. Since the headers can vary in width, applying this constraint to the lifelines themselves would either cause their headers to overlap or cause space to be wasted. Poranen et al. do acknowledge that longer message descriptions may also require increasing the distance between certain lifelines.

2.3 Box Plots

For most of the methods to be presented in this thesis, it is hard if not outright impossible to formally prove that they meet certain quality criteria. We rely on empirical evidence instead, running experiments to measure the effect different methods have on aesthetics. One of the simplest ways to visualize the resulting data are bar charts, but they are usually limited to showing only few summarizing statistics about the underlying data. If the data's distribution is of interest instead, we resort to *box plots*, a type of chart specifically tailored to this application. Necessarily being more complex as well, they probably deserve a bit of introduction before we can use them in this thesis, though. For a more thorough overview of different kinds of box plots, I refer to McGill et al. [MTL78].

Figure 2.3 shows box plots of four sets of data points, randomly generated based on different distributions. Box plots are based around the concept of *quartiles*, which are three points in the data set that partition the data into four equally-sized subsets. The second quartile is the median, while the first and third quartiles are the middle points between the me-

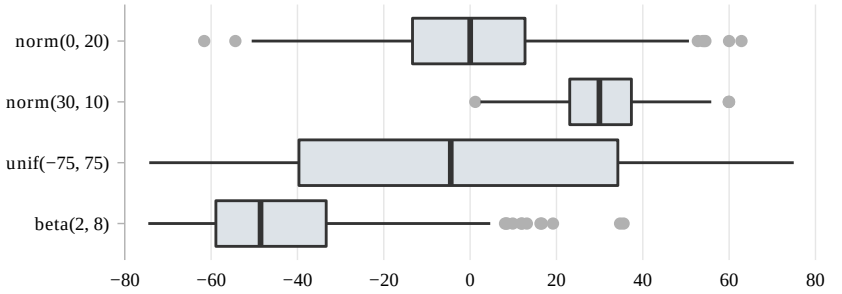


Figure 2.3. Box plots of random data generated according to several distributions: normal distribution (characterized by mean and standard deviation), uniform distribution (characterized by the interval the values are sampled from), and beta distribution (characterized by alpha and beta parameters and mapped from the interval $[0, 1]$ to the interval $[-75, 75]$ to be comparable).

dian and the lowest and highest number in the data set, respectively. The quartiles are also referred to as the 25th, 50th, and 75th *percentiles* according to the fraction of data points lower than or equal to the respective quartile.

In the plot, the three quartiles appear as the boundaries of and the thick line through the box itself. The lines that extend from the box are aptly called its *whiskers*. They extend towards the lowest and highest points of the data set, but are limited in length to 1.5 times the distance between the first and the third quartile (what is also called the *inter-quartile range*). Data points outside of this range are considered to be *outliers* and are shown as explicit points beyond the whiskers.

2.4 The Eclipse Platform

Among developers, Eclipse is probably best known as an IDE for the Java programming language. This, however, is not the whole story. Primarily, Eclipse offers a platform to build applications upon, which the Java Development Tools (JDT)[2] are just one example of.

[2] `https://www.eclipse.org/jdt/`

Eclipse began its life inside IBM. In November 2001, it was released as an open source project through founding the *Eclipse Board of Stewards*, which, besides IBM, initially included companies such as Borland, Red Hat, and Rational Software. In February 2004, the independent *Eclipse Foundation* was brought to life to manage the platform's continued development.

Eclipse enjoys strong backing from the industry, with many companies having chosen Eclipse as a strategic development platform. Two characteristics seem to be essential in making this possible. First, virtually all of the code contributed to Eclipse is released under the Eclipse Public License (EPL). An open source license approved by the *Open Source Initiative*,[3] the EPL allows open source code to be sold and distributed together with closed source code, which makes it appealing to companies. And second, the Eclipse Foundation makes each software release go through a review process designed to ensure that no intellectual property rights are infringed, which gives companies a certain amount of legal security.

Eclipse would not be much of a platform if it were not for its extensibility. At its heart lies *Equinox*,[4] which is an implementation of the Open Service Gateway Initiative (OSGi) core framework specification,[5] a dynamic component system for Java applications. In this system, an application is composed of *bundles*: Java archive files that contain files and meta data. Let us briefly explore how that works through the example of a (simple) spell checker to be integrated with a (simple) text editor.

Each bundle provides functionality, which in our example would be the ability to spell check text. Their implementation can be based on other bundles, which introduces *dependencies* to those bundles. Our spell checker might, for example, make use of a (simple) text parser implemented in a separate bundle (see Figure 2.4). At runtime, that bundle would need to be present for the spell checker to work, which is what the dependency expresses.

Obviously, our spell checker would be useless if the text editor was not able to use it. Its mere existence does not achieve that feat, which is why bundles can define *extension points* and *extensions*. An extension point

[3] https://opensource.org/licenses/

[4] https://www.eclipse.org/equinox/

[5] https://www.osgi.org/developer/downloads/

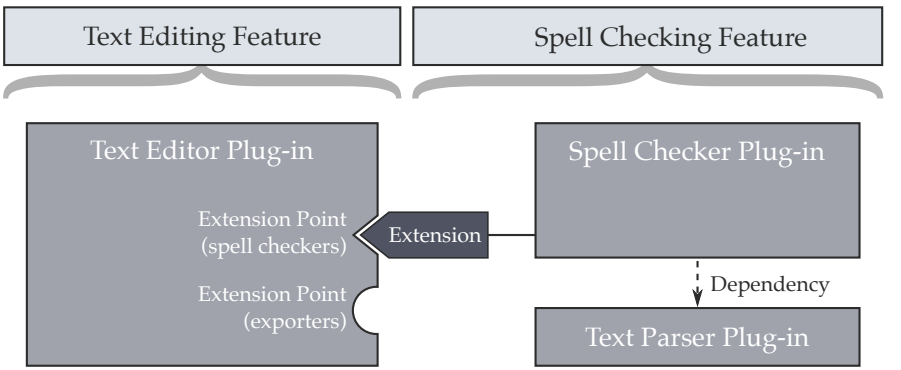


Figure 2.4. Eclipse is based on *bundles* (shown as rectangles). Bundles can either be *plug-ins* (which usually contain code) or *features* (installable units that link to all the plug-ins necessary to provide a certain piece of functionality). Plug-ins work together by offering *extension points* which other plug-ins plug into through *extensions*.

defined by a bundle provides a way for other bundles to interface with it. To do so, they register extensions with the extension point. Continuing our example, the text editor might define an extension point to register spell checkers. Each extension would have to specify a class that implements an interface provided by the text editor that it wants to use to access the spell checking functionality. This is exactly what our spell checker would do: implement the text editor's spell checking interface and register with its extension point.

In the Eclipse world, bundles are further divided into *plug-ins* and *features*. Plug-ins are small pieces of functionality, such as our spell checker. Projects such as the JDT are composed of many such plug-ins which, installed together, provide users with a major piece of functionality. To keep from having to install each plug-in separately, developers combine them into features for easier installation.

2.4.1 Noteworthy Eclipse Projects

Since all of the code developed as part of this thesis was developed in the form of Eclipse bundles, it interfaces with other Eclipse projects, either to make use of their functionality or to contribute new functionality to them.

Eclipse Modeling Framework

The Eclipse Modeling Framework (EMF)[6] is arguably one of the most successful and most widely used projects in the Eclipse ecosystem. At its heart, EMF is a code generator aimed at generating Java code that implements complex data models: instead of repeatedly writing similar code manually, one specifies the data model in some other way and lets EMF generate the actual implementation. The data model's specification is called the *meta model*. Similar to a UML class diagram, the meta model defines all the data types in the data model, their attributes and operations, as well as the relationships between them [SBP+09].

The generated code offers most of the functionality usually required in data models, examples being built-in support for listening to change events or for loading and saving models in a format based on the XML Metadata Interchange (XMI) standard. Finally, the generated code can just as well be used in pure Java applications not built upon the Eclipse platform.

In general, EMF-generated code tends to be more heavy-weight compared to code written by hand. However, configuring the code generator appropriately can limit the severity of this potential downside.

All of the more complex data models described in this thesis are ultimately based on and implemented with EMF.

Xtext and Xtend

Xtext[7] consists of a library and a complementary set of tools for defining and implementing textual DSLs. Based on a grammar definition, Xtext generates code for a text editor that supports standard editing features such as syntax highlighting and content assist. The textual representation in each

[6] https://eclipse.org/modeling/emf/

[7] https://eclipse.org/Xtext/

such editor is always backed by an EMF-based data model that is updated whenever the text changes.

Xtext integrates with EMF and provides support for loading and saving EMF data models using their corresponding DSL syntax instead of the XMI-based one. The generated editor code is highly configurable, with aspects such as code formatting, content assist suggestions, and code validation rules all eligible for customization.

Implemented with Xtext, *Xtend*[8] is a Java dialect designed to be more flexible and expressive. Xtend offers full integration into the Eclipse IDE and compiles to Java code.

Some of the code developed for this thesis was implemented in Xtend while the KieSL language to be introduced in Section 4.3 was implemented with Xtext.

Graphical Editing Framework

The Graphical Editing Framework (GEF)[9] provides several end-user tools as well as a framework for building graphical editors. Editors for visual languages are backed by arbitrary data structures which can be—but are not limited to—EMF-based data models.

2015 marked the first release of version 4, a major overhaul of the framework based on *JavaFX*, focussing on a more modern and lightweight implementation. The project has since evolved to version 5, with the pre-version 4 legacy components still available, but not actively developed any more.

GEF supports the concept of automatic layout out of the box, shipping with implementations of several layout algorithms. For our layout infrastructure to be described in Section 2.5, there have already been discussions with the GEF team on how best to integrate the two technologies to make it as easy as possible for developers to integrate automatic layout into their applications.

[8] https://www.eclipse.org/xtend/

[9] https://eclipse.org/gef/

Graphical Modeling Framework

While GEF provides everything required to implement visual editors, the GMF[10] builds on GEF to provide code generation for visual languages backed by EMF models. The syntax of a language—what elements are available, how they relate to the EMF model, and what they look like—is defined in a model, which is (again) used as the input for a code generator. The code generated by GMF aims to be very flexible and customizable, but that comes with the usual price to be paid in terms of complexity.

Still based on the legacy components of GEF, it is the experience at our research group that visual editors implemented with GMF have by now fallen behind the usability and visuals provided by other, more modern editors.

Similar to GEF, GMF supports the concept of automatic layout through what it calls *layout providers*. Our layout infrastructure to be described in Section 2.5 integrates with GMF-based editors.

Sirius

Sirius[11] allows developers to create whole editing environments to edit EMF-based models using a declarative approach that does not involve code generation. The editing environment can provide different kinds of editors of which graphical editors are but one. Graphical editors in Sirius are built on GMF.

According to its project description,[12] Sirius claims two main advantages. First, the declarative approach together with the absence of code generation supposedly allows for very fast development cycles as the effect of changes to declarations can be evaluated very quickly. And second, the declarative approach is assumed to require less technical knowledge than other approaches based on the manipulation of Java code.

Based on the GMF integration, the Sirius team provides an experimental interface to make it easy for tool developers to integrate our layout infrastructure to be described in Section 2.5 into their graphical editors.

[10] https://www.eclipse.org/modeling/gmp/

[11] https://eclipse.org/sirius/

[12] https://projects.eclipse.org/projects/modeling.sirius

Graphiti

Similar to GEF, Graphiti[13] is a framework for building editors for visual languages. The language's visuals are defined by a *pictogram model* and drawn by a rendering engine. Again, changes in the editor are reflected by changes in the underlying domain model. Graphiti was intended as a more lightweight alternative to GMF, but does not seem to be under active development any more.

Graphiti has a built-in concept of so called *layout features*, but these do not match our concept of automatic layout algorithms in this thesis. For us, computing a layout means calculating positions for all diagram elements. In Graphiti, layout features only ever handle a single diagram element, which will usually be composed of several graphical components whose positions must change when the diagram element's size changes. The node micro layout concepts to be introduced in Section 3.2 bear some similarity to Graphiti's layout features.

Our layout infrastructure to be described in Section 2.5 integrates with Graphiti-based editors.

Papyrus

Papyrus[14] is an environment for editing EMF-based models, with special support for the UML and related languages. Based on GMF, it aims to provide editors for all different kinds of UML diagrams. The particular focus of Papyrus is customizability.

Since the UML diagram editors provided by Papyrus are based on GMF, our layout infrastructure to be described in Section 2.5 integrates with them. Furthermore, the UML sequence diagram layout algorithm described in Chapter 4 was originally developed for Papyrus.

[13] https://eclipse.org/graphiti/
[14] https://eclipse.org/papyrus/

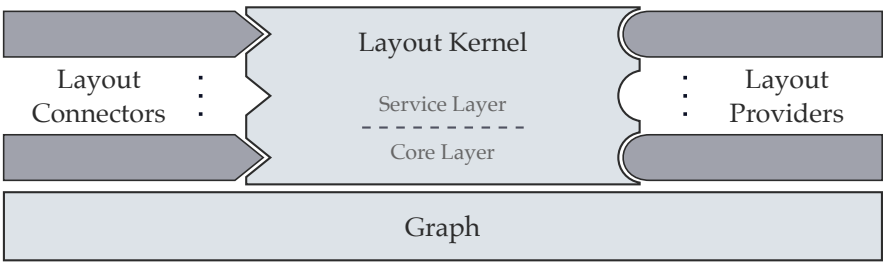


Figure 2.5. The basic structure of ELK. The graph and layout kernel components (lighter color) are core components while further layout connectors and layout providers (darker color) can be contributed by clients.

2.5 The Eclipse Layout Kernel

The Eclipse Layout Kernel[15] is an Eclipse project that provides an interface between diagram viewers on one side and layout algorithms on the other. To allow the two parties to talk to each other, ELK defines a graph data structure optimized for describing layout problems. By connecting viewers and algorithms not directly to each other but to ELK, new viewers get immediate access to all layout algorithms connected so far. Conversely, connecting new algorithms makes them immediately available to all viewers that are already connected. Quite simply, then, ELK can be understood as being structured into the four components shown in Figure 2.5.

The *ELK graph* constitutes the data structure the different components use to talk about layout problems. It is important enough to be examined in greater detail in the next section.

Diagram layout connectors provide the link between a diagram viewer and ELK. Given a viewer or the data structure it displays, a layout connector knows how to produce an instance of ELK's graph data structure that describes the corresponding layout problem. Once layout algorithms have finished executing, the layout connector also knows how to apply the computed layout information back to the viewer. Writing a new layout connector is a two-step process. First, the actual connector must be writ-

[15] http://www.eclipse.org/elk

ten by implementing the `IDiagramLayoutConnector` interface. Second, ELK must be made aware of the layout connector's existence by implementing the `ILayoutSetup` interface and registering it with ELK's `layoutConnectors` extension point. The setup is used by ELK to obtain new instances of the layout connector. For convenience, ELK already ships with layout connectors for some of the more popular diagram editor frameworks which can be customized to the specific use case.

Layout providers are implementations of layout algorithms. Each provider will usually support *layout options* to adjust it to specific requirements (we will look at how layout options work when discussing the graph structure in the upcoming section). Similar to writing new layout connectors, developing a new layout algorithm consists of two steps. First, the `AbstractLayoutProvider` class must be extended and implemented as it will later become the entry point to the layout algorithm. Information about the layout provider—such as its name, description, main class, and supported layout options—must be declared by writing and occasionally updating a Meta Data for ELK (MELK) file, which the ELK tooling compiles into a number of Java classes. In the second step, the main of these classes must be registered with ELK's `layoutProviders` extension point to make ELK aware of the new layout algorithm and the layout options it supports. ELK ships with implementations of a number of layout algorithms, among them a tree-based algorithm (aptly named *Mr. Tree*), a force-based algorithm, and a layer-based algorithm (we will look at the layer-based algorithm in detail in Chapter 3).

The *layout kernel* provides the link between diagram layout connectors and layout algorithms. It has two responsibilities: first, to keep track of the available connectors and algorithms; and second, to provide the entry points to actually invoke automatic layout (this will be examined more closely in Section 2.5.2).

Even though ELK is an Eclipse project, it can also be used outside of Eclipse. The layout kernel is in fact split into a *core layer* which does not require Eclipse and a *service layer* which does. Layout algorithms can also be invoked directly without involving the layout kernel at all, which can work just fine. A JavaScript version compiled from the original Java code

is available as well,[16] although it is not an official component of the ELK project.

Before working through how the layout process actually works, it is now time to familiarize ourselves with ELK's graph data structure.

2.5.1 Graph Structure

ELK's graph data structure (simply called the *ELK graph*) is an EMF-based model designed to serve as a complete specification of a layout problem and its computed result. There are three aspects to this: a graph's structure, its coordinates, and layout options. We will now examine each of these in turn.

In the graph-theoretical sense, graphs in ELK are hierarchical directed hyper- and multigraphs with ports (although we will constrain ourselves to regular edges instead of hyperedges in this thesis). At their most basic, they are composed of nodes (`ElkNode`) connected by directed hyperedges (`ElkEdge`), as shown in Figure 2.6.

Sources and targets of edges are not simply nodes but *connectable shapes* (`ElkConnectableShape`), the reason being that there are two ways for edges to connect to nodes. First, they can connect to a node directly, as they do in SCCharts (see Figure 1.5 in Section 1.1.2). Second, they can connect to a node through one of its ports (`ElkPort`), as they do in Ptolemy II (see Figure 1.2 in Section 1.1.1).

Nodes, ports, and edges are all *graph elements* (`ElkGraphElement`). Each graph element can be decorated with *labels* (`ElkLabel`), which are graph elements as well. Note that this implies that labels can themselves be decorated with further labels.

Finally, nodes can contain an arbitrary number of child nodes, thereby introducing hierarchy into the graph. In fact, the graph itself is always represented by a single root node, and the parent-child relationships of the graph's nodes constitute its inclusion tree. Edges can connect nodes at arbitrary levels of the hierarchy, but are always attached to a parent node.

The central graph structure of an automatic layout framework would be delightfully pointless if it did not capture layout information as well.

[16] `https://github.com/OpenKieler/elkjs`

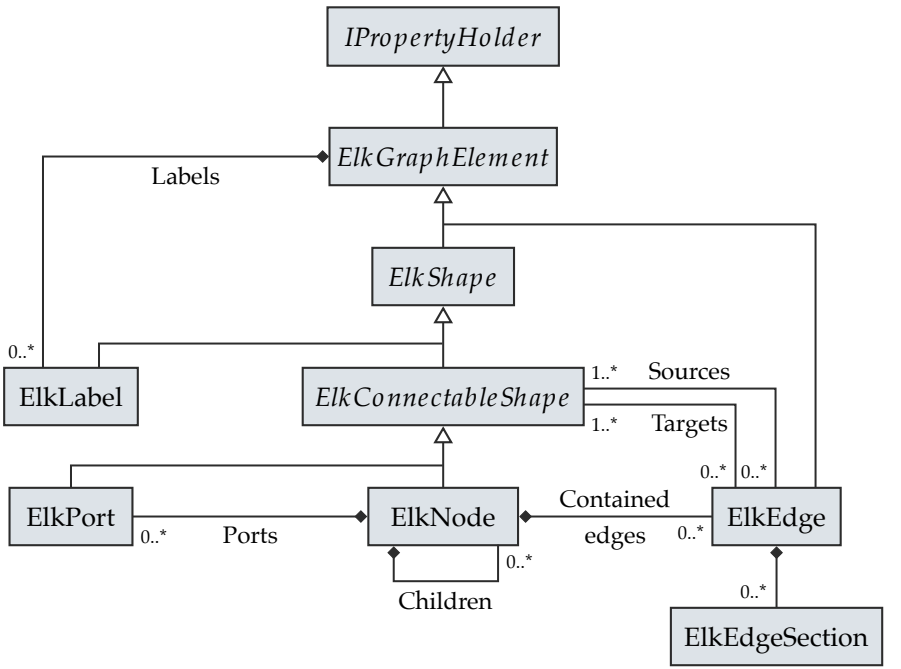


Figure 2.6. The ELK graph data structure, simplified to show only those parts relevant to this thesis. Interfaces and abstract classes are set in italics. Multiplicities are 1 unless defined otherwise.

All rectangular elements of the graph—nodes, ports, and labels—are *shapes* (`ElkShape`) defined by the x and y coordinates of their top left corner as well as their width and height. Even if they are not rectangular in the final drawing, ELK abstracts that fact away and is only concerned with their rectangular bounding box (although a specialized layout algorithm may choose to provide layout options that can carry more details about an element's actual shape).

Representing routing information of hyperedges is rather more complex. Graph models usually represent edge layouts by specifying the coordinates of the two end points as well as of a list of bend points. Since a hyperedge

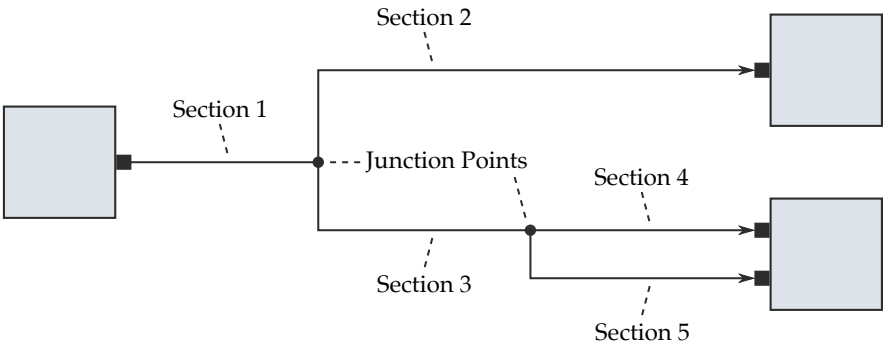


Figure 2.7. To describe the routing of hyperedges, ELK splits them into edge segments which have a single start and end point and may touch other edge segments at junction points (drawn here as circles). Beyond its start and end points, each section als has a (possibly empty) list of bend points.

is not a simple line from start to finish, however, this will not do. The ELK graph's representation divides a hyperedge's route into distinct *edge sections* (`ElkEdgeSection`), as shown in Figure 2.7. The routing of each edge section is similar to the routing of a simple edge: it has coordinates of its two end points and a list of bend points. In addition, it keeps lists of the edge sections it touches at its end points. The points where two edge sections meet are called *junction points*. A complex hyperedge routing can thus be reduced to a number of simple edge sections glued together. Note that for simple edges, not much has changed; the routing information that would normally be directly associated with the edge has simply moved to a single edge section.

So far, we have seen how the ELK graph captures information about structure and layout. What is still missing for it to complete the specification of layout problems is a way to configure layout options. This is the job of *properties* (`IProperty`). A property can be thought of as the specification of an available layout option, complete with an identifier and type information; in fact, during this thesis the terms *property* and *layout option* will be used interchangeably. Each graph element is a *property holder* (`IPropertyHolder`),

which means little more than that it stores mappings from properties to property values. It is through these that layout options are configured:

```
ElkNode node = ...;
node.setProperty(LayoutOptions.DEBUG, true);
```

The set of available options depends on the layout algorithm chosen for the graph's layout computation. If the algorithm does not support a given option, it will simply ignore it. If a layout option that it does support has no explicit value set, the algorithm will assume an algorithm-specific default value.

2.5.2 The Layout Process

The central entry point into automatic layout is ELK's `DiagramLayoutEngine`. While it offers different ways of invoking automatic layout, all of them expect the caller to say what they actually want to have laid out, either by supplying the viewer or an object that represents what is shown in the viewer (usually called the *view model*).

The diagram layout engine then looks through registered extensions to find an `ISetup` that claims to support the viewer or the view model. It uses that setup to instantiate all required classes through dependency injection. One of the most important classes of course is an appropriate `IDiagramLayoutConnector`, which is used by the diagram layout engine to turn the viewer or view model into an ELK graph that describes the actual layout problem.

Control now passes from the Eclipse-specific service layer down to the core layer, which has no concept of viewers or view models. Instead, it expects the layout problem to be presented in the form of a fully configured ELK graph. Since the diagram layout engine just acquired exactly that, it passes control to an implementation of the `IGraphLayoutEngine` interface. This will usually be the default `RecursiveGraphLayoutEngine`, although clients are free to change this through dependency injection.

Working its way from the bottom to the top of the inclusion tree, the recursive graph layout engine obtains the layout algorithm configured for each simple graph and executes it. Once all child graphs in a graph have

been laid out and have thus had a size computed and assigned, layout can be executed on that graph itself. Note that this works regardless of whether or not a layout algorithm explicitly supports hierarchy. If it does, layout can be configured such that the algorithm does not limit itself to laying out only a simple graph, but instead lays out a whole sub tree of the inclusion tree in one go. If the algorithm does not support hierarchy, calling the algorithm once for each simple graph in the inclusion tree still makes it possible for the algorithm to be run on hierarchical graphs.

Once the graph layout engine is done, the diagram layout engine finishes the job by asking the diagram layout connector to apply the computed layout information back to the viewer or view model.

Note that the computed layout is not limited to be specified only in terms of positions, sizes, and bend points. Instead, layout algorithms are free to change properties in the layout graph as well. This will become important once we discuss the implementation of label management techniques in Chapter 6.

2.5.3 Layout Creation and Layout Adjustment

As hinted at in Section 1.4, Misue et al. [MEL+95] distinguish between layout creation and layout adjustment algorithms. Layout creation algorithms compute a graph layout from scratch, only based on the input graph's structure. Layout adjustment algorithms, on the other hand, take existing layout information into account when computing the layout, often in an attempt to change the existing layout as little as possible. Misue et al. argue that this supports users by preserving the correctness of their mental map of the diagram.

It is easy to see that the ELK infrastructure supports both kinds of algorithms. As long as the code that turns a layout problem into its ELK graph representation includes information about an existing layout, algorithms are free to make use of or to ignore these information.

This freedom implies a subtle, but interesting point: by being able to decide which of the existing layout information to take into account and which to discard, layout algorithms can resist attempts to neatly divide them into the two categories. The layout algorithm to be described in Chapter 3

is a layout creation algorithm at its core, but can be configured to take existing layouts into account for certain placement decisions. The algorithm to be introduced in Chapter 4 is harder to categorize still. Thus, although appearing to be two discrete categories, the distinction between layout creation and layout adjustment instead establishes a continuous spectrum with algorithms at either end as well as along the middle.

The question of which kind of algorithm to apply to which layout problem is too unspecific to be properly answered [RLP+16, Section 6]. The requirements of any given use case have to be compared to the specific mixture of aesthetics emphasized by any given layout algorithm. Even if the use case were to exclude layout creation algorithms, this would not allow us to conclude that any layout adjustment algorithm would do. One algorithm may try to move elements as little as possible while another may only preserve a diagram's general topology. A use case may call for the former behavior while rejecting the latter, or vice versa.

On a final note, since layout adjustment scenarios often imply some kind of user involvement, algorithms in the ELK project that offer layout adjustment features advertise them under the term of *interactive* modes of operation. In this thesis, we will use the two terms interchangeably.

2.5.4 A Bit of History

Originally, ELK started out as the KIELER Infrastructure for Meta Layout (KIML) [Spö15] at Kiel University's Real-Time and Embedded Systems group. In August 2015, ELK was officially accepted as a new Eclipse project under the umbrella of the *Eclipse Modeling Project*,[17] with the author serving as the project lead and Miro Spönemann and Ulf Rüegg being the initial committers. The project produced its first release in July 2016.

Given that the move to Eclipse required lots of code changes anyway to ensure that names of plug-ins and packages conform to Eclipse guidelines, the team took the opportunity to start solving some of the issues that had accumulated in KIML over the years.

[17] `https://eclipse.org/modeling`

▷ In KIML, information about layout options supported by an algorithm and their default values were scattered throughout the code. They first had to be registered with one of KIML's extension points, along with their default value. To actually access the option, the algorithm then had to declare a constant in its code, with the default value duplicated. This led to the usual problems with keeping things synchronized.

The introduction of MELK files (as mentioned on page 53) solved this issue by consolidating an algorithm's meta data into a single file.

▷ KIML provided developers with at least five different ways to set layout options. The most straightforward way was to simply set properties on the different layout graph elements, as described in Section 2.5.1. Another way was supplying a layout configuration class to the layout engine when invoking layout. Three more ways were based on one of KIML's extension points.

As it turned out, this was far too complex. For ELK, the configuration methods based on extension points were dropped in favor of having developers configure layout programmatically.

▷ KIML required diagram layout connectors to be registered directly. ELK introduces a layer of abstraction: not the connector is registered, but an implementation of the `ILayoutSetup` interface. ELK then uses dependency injection configured by that implementation to obtain instances of all core classes involved in the layout process. This allows for greater flexibility and customization.

▷ Finally, KIML's graph data structure was designed with the goal of separating structural information from layout information. The latter were attached to graph elements through implementations of the `KLayoutData` interface, resulting in code such as this for setting coordinates and properties of a node:

```
KNode node = ...;
KShapeLayout shapeLayout = node.getData(KShapeLayout.class);
shapeLayout.setXpos(...);
shapeLayout.setYpos(...);
```

```
shapeLayout.setProperty(...);
```

While separating the two concerns can make sense, I redesigned ELK's graph data structure during my work on this thesis to specialize on layout problems. It thus integrates both layout information and properties directly into the data structure. I also took the opportunity to introduce hyperedges by allowing edges to have more than a single source and target.

2.5.5 How ELK Relates to This Thesis

The relevance of ELK to this thesis differs for its two parts. All of the code developed for Part I, chiefly concerned with automatic layout algorithms, ended up at the ELK project. Much of the basic framework code developed for Part II can now be found in ELK's code base as well.

Some of the more high-level framework code, however, wandered into the KLighD project, which is what we will be discussing next.

2.6 KIELER Lightweight Diagrams (KLighD)

Developed at our research group under the lead of Christian Schneider, the KIELER Lightweight Diagrams (KLighD) framework was designed to be a simple way to produce transient views of arbitrary data structures [SSH13]. The term "transient" in this context refers to the fact that, instead of being persisted, views are always generated from scratch, building upon automatic layout to position their elements. For the actual rendering, KLighD defers to the *Piccolo* graphics framework [BGM04]. We have already seen KLighD in action in this thesis: the SCChart displayed in the center of Figure 1.6 was generated with it, for example.

Figure 2.8 shows the basics of how KLighD works. The input is a *domain model*: an arbitrary data structure which is to be visualized (Step 1 in Figure 2.8). This can be as simple as a string or as complex as a model of a large database. The domain model is not displayed directly since KLighD would not know how to do so. Instead, it must be transformed into a

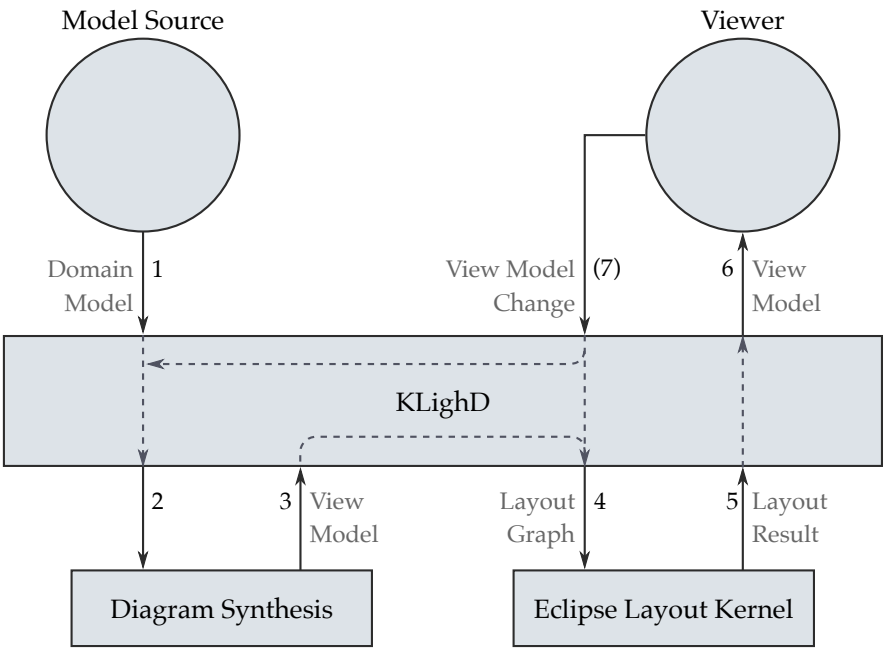


Figure 2.8. The way KLighD generates and displays views. Note how the (optional) feedback loop to the right allows users to influence the view model to adapt the view to their requirements.

corresponding view model, a description of the visualization that KLighD knows how to display.

The *view model* is a graph with nodes, ports, labels, and edges annotated with information on how they are to be rendered. The actual transformation of a domain model into an appropriate view model is performed by a *diagram synthesis* (Step 2). Each diagram synthesis is registered with KLighD through an extension point and knows how to transform a particular type of domain model into a view model. For KLighD, turning the domain model into its view model is simply a matter of finding the right diagram synthesis.

The view model thus generated (Step 3) will usually not contain any layout information yet. Displaying it in this state would result in an incom-

prehensible pile of graphical objects drawn at the origin of the coordinate system. Choosing to be useful instead, KLighD triggers ELK to compute a proper layout (Step 4).

Once the layout information are computed, KLighD applies them to the view model (Step 5). This is not limited to the position and size of graphical elements; it can also include changes to, for example, the text of labels, which is how label management works as we will see in Chapter 6.

Finally, the application can display the finished diagram through a viewer component provided by KLighD (Step 6).

Depending on the diagram synthesis that produced the view model, users may be presented with different options to customize the view to their current task (optional Step 7).

First, a synthesis may specify a set of *diagram options* and *layout options* for the user to modify the way the view model is generated. Changing diagram options can cause the whole process to be repeated (starting at Step 2) while taking the new settings into account. Changing layout options only causes automatic layout to be executed again (starting at Step 4).

Second, diagram elements can be associated with *actions* that are executed upon certain events. For example, double-clicking a compound node can cause an action to change its rendering and display the graph contained in the node, causing automatic layout to be invoked again (starting at Step 4). This particular action is called the *collapse-expand action* because it toggles nodes between a collapsed and an expanded state. Being able to switch between the two is what lies at the heart of the Ptolemy II viewer introduced in Section 1.1.1.

Now that the graph theoretical and the technical foundations have been laid, it is time to look at the layered approach to graph drawing as well as how it can be enhanced in accordance with the principles from Section 1.2.

Part I

Laying the Foundations

Automatic Layout

Chapter 3

Flow-Based Diagrams

This chapter will be all about text in flow-based diagrams, such as Ptolemy II block diagrams or SCCharts. The context of our discussions will be the layered approach to layout, the most popular layout algorithm for flow-based diagrams—to be introduced in detail in Section 3.1. Starting with Section 3.2, we will turn to my contributions. Addressing the **P-PLACEMENT** *principle, we will discuss how to place node, port, and edge labels, all the while making sure to reserve enough space for the labels to be properly placed. Regarding the* **P-SIZE** *principle, we will look at how to place edge labels such that we minimize a diagram's width.*

In Section 1.1 we have already seen several visual languages. One of them was the block diagram language which is part of Ptolemy II. An example of a language based on data flow diagrams, its objective is to show how data are transmitted between actors that constitute a piece of software. Another visual language were SCCharts, which emphasize the flow of control through the different states that comprise a piece of software, thus making them examples of control flow diagrams. Whatever it is that "flows through the diagrams," it is the very flow that is the focus of these representations. For the purposes of this chapter, we thus stop distinguishing between what flows and focus on the flow itself instead. The following definition aims to capture this concept.

Definition 3.1 (Flow-Based Diagram)**.** A *flow-based diagram* is a node-link diagram whose main objective is to emphasize the flow of data or control through the diagram. □

The emphasis on flow is achieved by laying out diagram elements such that the flow largely follows a uniform direction. This helps establish the In

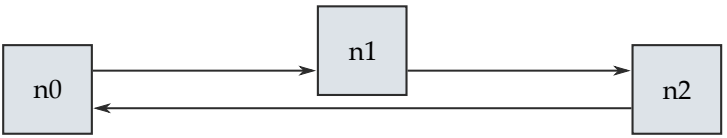
fact, breaking with this principle by, say, sending data through the diagram in a zigzag pattern would be a promising approach to annoy people who have to read it.

Conforming with the reading direction of western languages, the flow is usually directed rightwards or downwards. It is an interesting question whether readability suffers if the flow direction does not match the reading direction of a viewer's native language. If it does, automatic layout algorithms may be helpful by seamlessly changing the flow direction of existing diagrams to match the viewer's expectation. This is a research question, however, that will have to wait for another thesis.

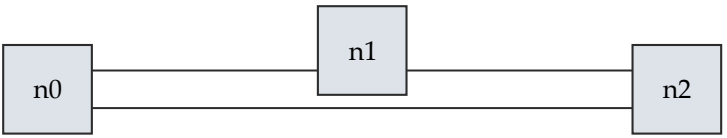
The perception of a diagram's prevalent flow direction can be supported by using directed edges. Interestingly, however, not all flow-based languages follow this approach. As we have seen in Section 1.1.1, Ptolemy II's block diagrams in fact use undirected edges since data can conceptually travel along an edge in both ways. The prevalent data flow direction then mainly follows from the placement of nodes. In the absence of further visual cues, undirected edges can make a flow-based diagram harder to read or even misleading, as Figure 3.1 shows. If the flow direction along an edge follows only from the placement of its end points, it can become impossible to draw *feedback edges*, that is, edges that oppose the prevalent flow direction. It is for this reason that the diagram shown in Figure 3.1b fails. Ptolemy II's directional port symbols, similar to the ones shown in Figure 3.1d, neatly get around that problem.

In this chapter, we will concern ourselves with two common challenges that arise in the layout of flow-based diagrams: the layout of whatever belongs to a node, and placing edge labels. The ways we solve some of these issues will also serve as foundations we will build on in Part II of this thesis.

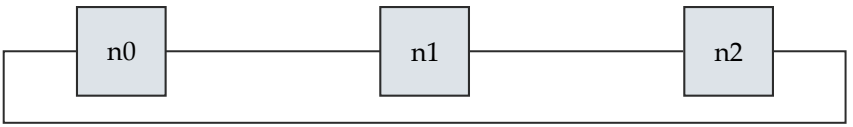
Before we can get to work on those problems, however, we need to introduce the most important layout method for flow-based diagrams.



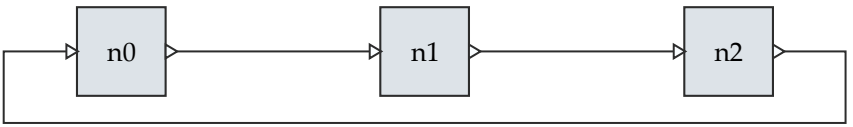
(a) Directed edges



(b) Undirected edges



(c) Undirected edges with fixed connection sides



(d) Undirected edges with explicit ports

Figure 3.1. Different ways to draw edges vary in how successful they are in showing flow direction. **(a)** A clear representation. **(b)** The edge from *n2* to *n0* does not look like a feedback edge. **(c)** Following a convention such as data always leaving nodes through their right and entering them through their left border can solve this problem, although this requires additional knowledge on the viewer's part. **(d)** Ptolemy II connects actors through undirected edges that do not have arrows. However, its ports are drawn as arrowheads, which helps infer edge directions.

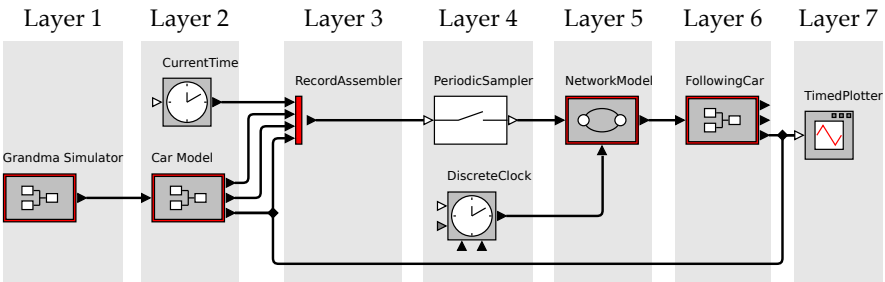


Figure 3.2. A simplified version of the diagram shown in Figure 1.2 as laid out with ELK Layered, a layer-based algorithm. The different layers are highlighted.

3.1 The Layered Approach

The *layered approach* to graph drawing was first introduced by Sugiyama, Tagawa, and Toda in 1981 [STT81]. Given an acyclic directed graph, its goal is to produce drawings in which all of the graph's edges point in the same direction. In the authors' terms, this is achieved by establishing what they call a *hierarchy* between the nodes, partitioning them into distinct *levels*, or *layers*, such that edges always point from lower to higher levels in the hierarchy (see Figure 3.2). Note that this usage of the term "hierarchy" differs from how we use it in this thesis, namely to refer to the concept of establishing parent-child relationships between nodes. We thus refrain from calling the layered approach "hierarchical layout," otherwise used synonymously in the graph drawing literature.

Being a perfect example of breaking a complex problem into more manageable pieces, the layered approach consists of five phases, of which the first and last did not appear in the original paper:

1. Cycle breaking
2. Layer assignment
3. Crossing minimization
4. Node placement

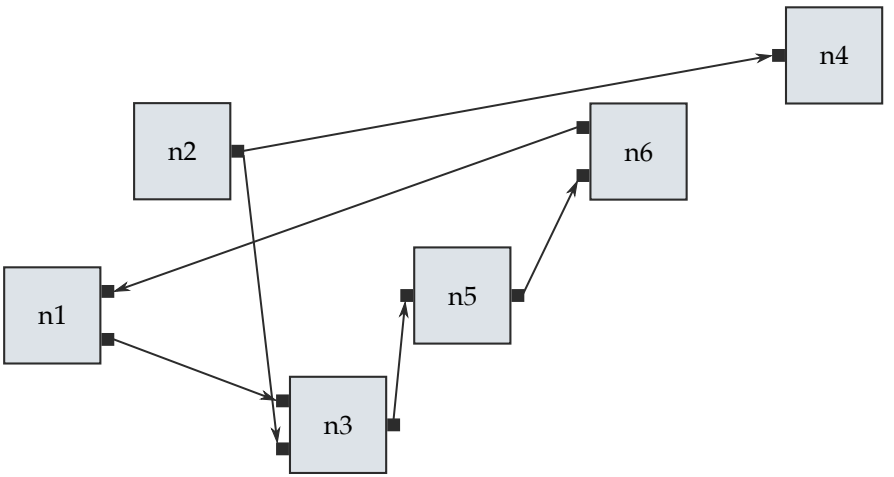


Figure 3.3. This simple input graph will serve as our example while working through the layered approach. This graph already features ports, which are not part of the basic layered approach. We will see how ports are handled in Section 3.1.7.

5. Edge routing

The following sections will briefly explain each phase as it does its magic on a simple input graph shown in Figure 3.3. We will then turn to specific details concerning how this approach was implemented in the ELK Layered algorithm that ships with the ELK project [SSH14]. This overview of the layered approach will close by looking at the problems that have to be solved once hierarchy is introduced. For a more comprehensive introduction to the topic, I refer to Healy and Nikolov [Tam13, Chapter 13].

Note that when discussing the layered approach, the graph drawing literature usually assumes a downwards layout direction. In keeping with the usual aesthetics of data flow diagrams, however, we assume a rightwards direction instead.

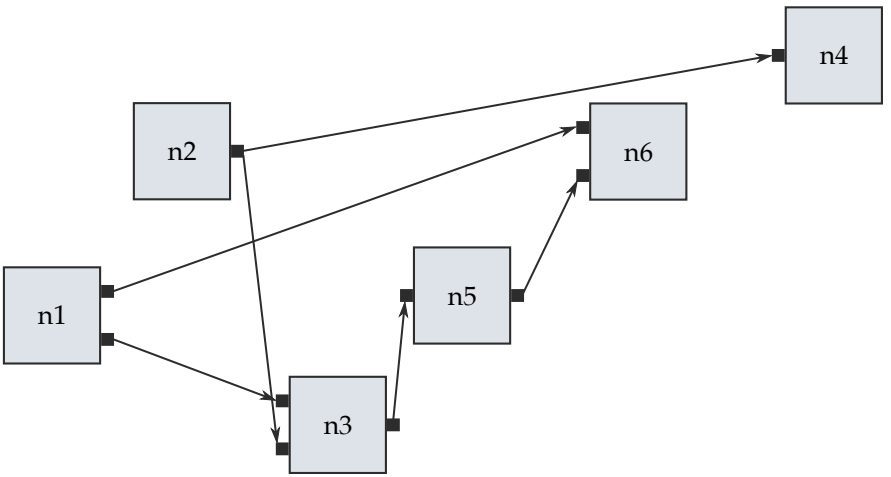


Figure 3.4. The first phase reverses edges to make the graph acyclic. In this case, the edge from *n6* to *n1* was reversed.

3.1.1 Phase 1: Cycle Breaking

It is hardly an intellectual challenge to see that in order to produce a drawing in which all edges point in the same direction, the input graph must be acyclic. A layout approach that simply assumed this to be the case, however, would share the fate of many a well-intentioned plan by failing upon the first encounter with the real world. The first phase is thus in charge of turning a graph with cycles into an acyclic graph for the duration of the algorithm by reversing the direction of some of its edges (Figure 3.4). In the final drawing, these edges will end up as feedback edges, opposing the prevalent layout direction.

Healy and Nikolov note that the original direction of reversed edges can be restored after the second phase [Tam13, Chapter 13] since only that phase solves a problem that strictly requires the graph to be acyclic. While this is certainly true, the implementation of subsequent phases is usually simplified if they, too, can operate under the premise of an acyclic graph.

Formally, the main problem to be solved during cycle breaking is to find a *feedback arc set*, to adopt the terminology used by Eades et al. [ELS93].

Definition 3.2 (Feedback arc set)**.** Given a directed graph $G = (V, E)$ and $R \subseteq E$. R is a *feedback arc set* of G if, after reversing its edges in the graph, G is acyclic. □

One basic approach to solve the problem chooses an ordering $S = v_1, \ldots, v_{|V|}$ of the graph's nodes and reverses all edges (v_j, v_i), $1 \leqslant i < j \leqslant |V|$ (or all edges with $i > j$). Of course, the number of reversed edges is a direct consequence of the chosen ordering—unfortunate choices may lead to drawings where up to 50% of the edges oppose the intended layout direction, which quite obviously defeats one of the most important goals of the layered approach. Instead, we usually want to keep the number of feedback edges small by finding a minimum cardinality feedback arc set. The problem of finding such a set is called the *minimum feedback arc set problem*. The associated decision problem of whether a feedback arc set can be constructed with at most $k \in \mathbb{N}$ edges has been shown to be *NP*-complete [Kar72].

A number of heuristics have been proposed over the years, for example based on depth-first search to find cycles [GKN+93], on modified sorting algorithms [BH11], or on the number of incoming and outgoing edges of each node [BS90].

Eades, Lin, and Smyth [ELS93] proposed a popular heuristic that falls into the latter category, known as the *greedy cycle breaking* algorithm. Based on the observation that edges connected to a source or a sink in the graph cannot be part of a cycle, the heuristic removes sources and sinks from the graph and inserts them into the ordering. If there are still nodes left, the heuristic removes the one that will result in the lowest number of feedback edges and repeats the whole process until no nodes are left.

A low number of feedback edges is not always of prime importance. Layout adjustment requirements may instead call for retaining the exact set of feedback edges seen in a previous drawing of a graph, even if it is not optimal with respect to the smallest possible number of feedback edges. A simple method proposed by Spönemann that he calls *sketch-driven layout* [Spö15] addresses this requirement. Assuming a rightwards layout

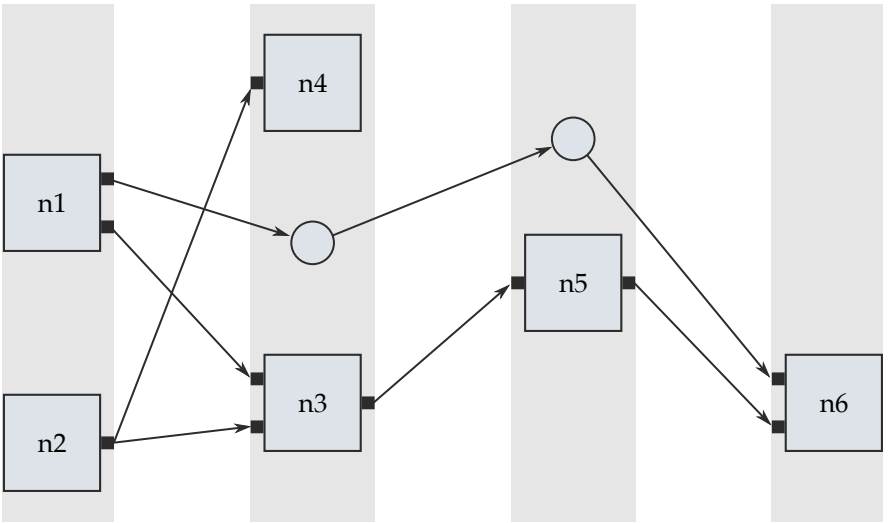


Figure 3.5. The second phase partitions the nodes into different layers. Note how edges only point rightwards, and how the long edge from *n1* to *n6* was broken by two dummy nodes.

direction, the method compares the coordinates of each edge's end points in the previous drawing (the *sketch*). If the target point is to the left of the source point in the sketch, the edge is reversed.

3.1.2 Phase 2: Layer Assignment

The layered approach would hardly deserve its name if it did not compute a *layering* at some point. Luckily, doing so is the responsibility of its second phase (Figure 3.5).

Definition 3.3 (Layering)**.** Let $G = (V, E)$ be a directed graph and $\mathcal{L} = L_1, \ldots, L_k$ be a partition of V. For $v \in V$, $\mathcal{L}(v)$ denotes the index i such that $v \in L_i$. $\mathcal{L}$ is called a *layering* of V if for each $(v_1, v_2) \in E$ it holds that $\mathcal{L}(v_1) < \mathcal{L}(v_2)$. L is called a *weak layering* if $\mathcal{L}(v_1) \leqslant \mathcal{L}(v_2)$. L is called a *proper* layering if $\mathcal{L}(v_1) = \mathcal{L}(v_2) - 1$. We call the elements of $\mathcal{L}$ *layers*. □

The layers can be thought of as columns, placed from left to right in the final drawing according to their index in the layering, while the nodes in each layer are placed below one another. Note that while the layered approach in its pure form calls for a proper layering, ELK Layered allows for a weak layering with certain constraints [SSH14].

We can distinguish different types of edges depending on how many layers they span.

Definition 3.4 (Long / short / in-layer edge)**.** Let $G = (V, E)$ be a directed graph and let $\mathcal{L}$ be any kind of layering of V. We call $e = (v_1, v_2) \in E$ a *long edge* if $\mathcal{L}(v_1) < \mathcal{L}(v_2) - 1$, a *short edge* if $\mathcal{L}(v_1) = \mathcal{L}(v_2) - 1$, and an *in-layer edge* if $\mathcal{L}(v_1) = \mathcal{L}(v_2)$. □

We can turn a layering into a proper layering by inserting *dummy nodes* to break long edges and thereby turn them into sequences of short ones. The dummy nodes are removed again once the algorithm has finished, thereby restoring the original long edges.

Different layering algorithms have been proposed that try to optimize for different goals. The *longest-path layering* algorithm [RDM+87] is an easy and fast method that produces a minimal number of layers (as dictated by the length of the graph's longest simple path), but can result in a lot of nodes being assigned to the last layers in particular, as well as in more long edges than necessary [HN02b].

It is the latter problem that the *network simplex layering* algorithm addresses [GKN+93]. It reduces the problem to a minimum-cost flow problem and solves it using the network simplex algorithm. Although not proven to be polynomial, the authors claim that the algorithm is fast in practice—an observation supported by experience at our research group. Network simplex layering minimizes the number of dummy nodes in the whole graph, not in each layer.

To state the obvious, the choice of layering influences the width and the height of the resulting drawing. Trying to minimize both the number of layers as well as the number of nodes per layer, which is the aim of the Coffman-Graham algorithm [CG72], for example, is an NP-hard problem [ES90], even without taking node sizes into account. When taking the size of regular and dummy nodes into account and bounding the height of

each layer, the layering problem is *NP*-complete [BLM+01]. Besides algorithms that solve versions of this problem to optimality [HN02b; HN02a], the *MinWidth* algorithm by Tarassov, Nikolov, and Branke [TNB04; NTB05] takes a heuristic approach based on modifications to the longest path layering algorithm. Rüegg et al. [RAC+17] compare the performance of several space-minimizing heuristics as well as traditional algorithms.

All of these approaches try to do something about the size of a drawing, but they cannot overcome a basic problem of the layered approach: each layer is as wide as its widest node, which leads to wasted space if node sizes differ. Hierarchical graphs are a prime example: since the hierarchical nodes contain further nodes themselves, they are usually much wider than the graph's typical simple nodes. Rüegg et al. [RSG+16] show how to solve this problem by post-processing the computed layout, sacrificing the clean visual appearance of layers in favour of more compact drawings. Rüegg and von Hanxleden [RH18] also show how to insert deliberate "line breaks" into very wide graphs, effectively trading width for height in an attempt to produce more screen-friendly aspect ratios.

In layout adjustment scenarios, all optimizations are discarded in favor of maintaining a previous drawing's topology. Spönemann's sketch-driven layout [Spö15] derives a layering from a sketch by considering the horizontal coordinate as well as the width of each node.

3.1.3 Phase 3: Crossing Minimization

With nodes assigned to layers, the third phase computes the order of nodes inside each layer in an attempt to reduce edge crossings in the final drawing (Figure 3.6).

To quote Di Battista et al., "the number of edge crossings in a drawing of a layered digraph does not depend on the precise position of the vertices, but only on the ordering of the vertices within each layer" [DET+99]. If edges are drawn as straight lines, this is indeed true and an ordering can be computed efficiently [BJM02]. If, however, they are drawn orthogonally (as they are in Figure 1.2, for example) the exact position of the vertices does make a difference [EGB03; San04] and we find ourselves in the middle of an unfortunate circular dependency: to count crossings accurately and thereby

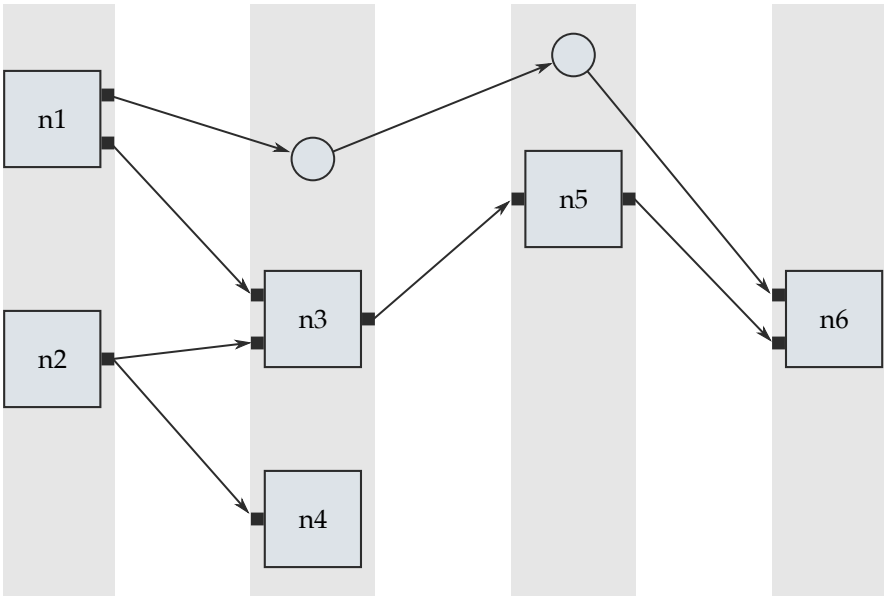


Figure 3.6. The third phase orders the nodes such that the number of edge crossings are reduced. In this case, *n4* was moved to the bottom of its layer.

to judge which node order to prefer, the crossing minimization phase needs node coordinates that can only be computed after crossing minimization has finished. Spönemann et al. [SSR+14] show how to circumvent the problem by replacing exact crossing numbers with estimates.

Finding a node order that minimizes the number of crossings is, alas, NP-complete [GJ83]. The statement continues to hold if the graph consists of only two layers and even if the order of one of them is fixed [EMW86]. While there are methods to compute optimal solutions, for example by Jünger et al. [JLM+97], people usually resort to heuristics.

The most popular such heuristic, called the *layer sweep heuristic*, was proposed by Sugiyama, Tagawa, and Toda in their original paper [STT81] (they called it the "down-up procedure"). It does not reduce crossings between all layers at once. Instead, it sweeps back and forth, only reducing

crossings between pairs of adjacent layers at a time of which one is fixed. Nodes in the fixed layer are assigned *ranks*, which are usually numbers that correspond to their order in the layer. Nodes in the free layer are ordered according so some value computed based on the ranks of their neighbors in the fixed layer. Researchers have come up with different ways do compute these values, for example using *barycenters* [STT81] or *medians* [EW86].

Once we introduce hierarchical nodes to the problem, the usual approach is to lay out every level of hierarchy separately (see Section 2.5.2). It is, however, easy to construct examples where this kind of "isolated" crossing minimization leads to unnecessary crossings in the final drawing [For02]. Fuhrmann [Fuh12] provides a comprehensive review of solutions proposed for this problem, while Schelten [Sch16a] shows how to extend the layer-sweep approach to hierarchical graphs.

Again, for layout adjustment scenarios Spönemann's sketch-driven layout [Spö15] shows how to derive node orders from existing node coordinates.

3.1.4 Phase 4: Node Placement

The fourth step of the layered approach assigns y coordinates to all nodes (Figure 3.7). As a side-effect, this also determines the height of the final drawing. There are different goals one can strive for in the process.

One such goal is to minimize the height of the drawing. A trivial method which does so places all nodes as close together as spacing constraints allow, and then optionally centers them vertically. While this method is interesting in that it defines a lower bound on the drawing's height, it tends to produce placements that seem "crowded" and have many bend points.

Other goals include producing a *balanced* placement (centering nodes with respect to their neighbors in other layers), minimizing the length of edges, and straightening as many edges as possible. Node placement methods differ not only in performance or in whether they solve the problem approximately or optimally, but also in the goals they try to achieve in the first place.

The *linear segments* method proposed by Sander [San96] regards all dummy nodes that split a given long edge to constitute a linear segment.

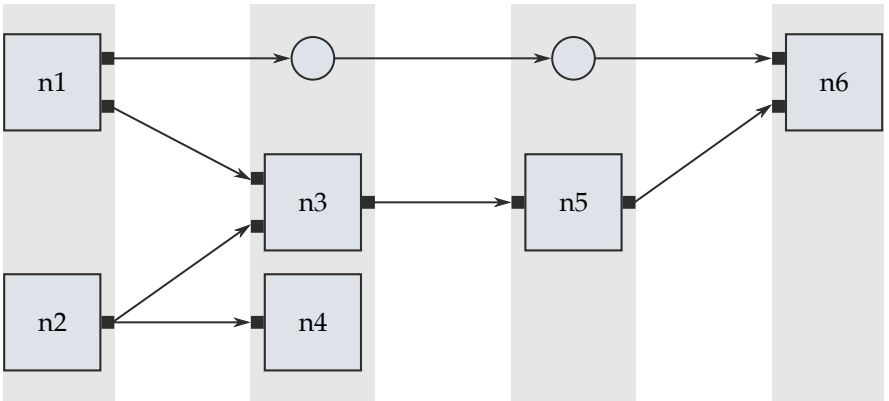


Figure 3.7. The fourth phase assigns y coordinates to all nodes. Note how in this case, several edges can be drawn straight, including the long edge broken by the two dummy nodes.

Each regular node is part of a trivial linear segment that contains only that node. The method then strives for a balanced placement not between nodes, but between linear segments, causing long edges to be drawn as straight lines (at least between their first and last dummy nodes). The computations are based on the physical analogy of regarding the linear segments as a system of pendulums connected by strings.

Gansner et al. [GKN+93] formulate an optimization problem to minimize differences in the y coordinate of edge end points, favoring straight long edges over straight short ones. They solve the problem by constructing an auxiliary graph that can be fed to the network simplex algorithm. This not only results in an optimal solution with respect to their optimization goals, but also does so very quickly, according to the authors. Additionally, their formulation allows them to cater for edges that connect to a node through a port placed at a fixed position at the node's border.

Building upon ideas by Buchheim, Jünger, and Leipert [BJL01], Brandes and Köpf [BW97] propose a method that tries to maximize the number of straight edges. Carstens [Car12] and later Rüegg et al. [RSC+15] extend the

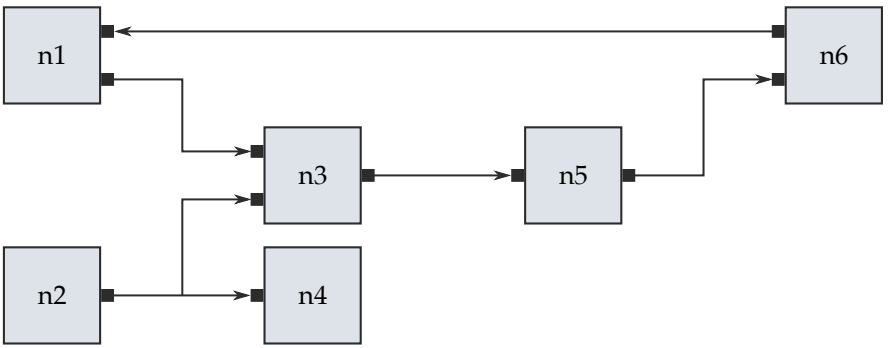


Figure 3.8. The final phase routes edges and assigns x coordinates to all nodes. In this case, edges are routed orthogonally. Note how the edge from *n6* to *n1*, originally reversed by the first phase, is now restored to its original direction.

algorithm to support non-uniform node sizes and ports, effectively allowing more than one edge per node to be drawn straight.

3.1.5 Phase 5: Edge Routing

The final phase of the layered approach routes edges between each pair of consecutive layers (Figure 3.8), which also determines the amount of space that needs to be left between them. A byproduct of this phase thus is that it computes the x coordinates of all nodes and with it the width of the whole graph.

There are different styles of edge routing. The simplest one, which we shall call *direct edge routing*, was assumed in the original paper by Sugiyama, Tagawa, and Toda [STT81] and draws edges simply as direct lines between the nodes they connect (which is why that paper does not even propose a dedicated edge routing phase). This implies that only long edges can have bend points, and that those bend points coincide with the positions of the dummy nodes used to break them.

Once we remove the restriction that bend points can only occur at dummy nodes, we have what is usually called *polyline edge routing*. The additional freedom allows us to solve two problems of direct edge routing.

First, direct edge routing cannot route self-loops while polyline edge routing is free to insert the necessary bend points. Second and perhaps more importantly, if nodes do not have uniform sizes, there will be some that do not extend to their layer's boundaries. With direct edge routing, edges that connect to such nodes can cross other nodes when entering the layer's realm, while polyline edge routing inserts a bend point at the layer boundary.

Orthogonal edge routing constrains polyline edge routing such that edges are routed as sequences of alternating horizontal and vertical edge segments. Unless the end points of a given edge can be connected by a single horizontal edge segment, the edge will have at least one vertical segment and two bend points. Overlapping vertical segments of different edges are avoided by assigning them different x coordinates. The choice of coordinates influences the number of crossings. Sander determines the coordinates using a simple sweep line approach for graphs without hyperedges [San96] or by breaking cycles in a weighted auxiliary graph for graphs with hyperedges [San04], while Baburin [Bab02] solves a vertex coloring problem on an auxiliary graph.

Spline edge routing finally abolishes abrupt bend points by smoothly routing edges through the diagram as splines. The undertaking is complicated by the fact that smooth curves need space. Gansner, North, and Vo [GNV88] solve the problem by enlarging dummy nodes to reserve the necessary space. As the authors themselves acknowledge in a later paper [GKN+93], however, their technique can still result in sharp bends when the reserved space turns out to be insufficient. They replace the original method by a new two-step method. The first step computes a set of touching rectangles which represents the region in the diagram an edge may be safely routed through. The second step then smoothly routes a spline through these rectangles. Note how this method solves the edge routing problem globally instead of reverting to the usual reduction to a series of subproblems involving only consecutive layers. Toepffer [Toe14] takes the latter approach when he evaluates the suitability of different spline representations and applies methods for assigning x coordinates to vertical segments of orthogonal edges to the routing of splines between consecutive layers.

3.1.6 ELK Layered

The ELK project provides an implementation of the layered approach called *ELK Layered*, the roots of which go back to a predecessor called KIELER Layout for Data Flow Diagrams (KLoDD), developed in 2009 by Spönemann [Spö09]. While ELK Layered already existed before this thesis, the concepts yet to be introduced in this chapter have been implemented in the context of that algorithm.

As shown in Figure 3.9, ELK Layered is surrounded by an import and and export step. This is because internally, the algorithm does not use the ELK graph, but a graph data structure which actually knows about layers, called the *layered graph* (an immense help to an algorithm based on them).

The main part of the algorithm is divided into the five phases outlined above, each covered by different implementations for clients to choose from (Table 3.1 lists the major ones). Before control passes from one phase to another, however, it moves through an *intermediate processing slot* which houses an arbitrary number of *intermediate processors*: pieces of code that process the layout graph in some way [Sch11]. The idea is to keep the phases themselves as simple as possible by factoring out some of the processing, particularly as it pertains to special cases. The layer assignment phase is a good and simple example. Most layer assignment algorithms, while of course producing a layering, do not necessarily produce a *proper* layering. An intermediate processor executed right after layer assignment can take care of splitting any long edges it finds; another one, executed after the last phase, can then restore the original long edges.

Implementing things this way has two advantages: it keeps the phases themselves as simple as possible, and it reduces code duplication. Different layer assignment algorithms can, after all, simply share the same code to produce proper layerings. These advantages come at a certain cost in terms of runtime performance because each executed processor will usually add another iteration over the whole graph. Also, making everything run smoothly requires to carefully keep track of dependencies between intermediate processors [SSH14]. To illustrate, imagine a processor which places ports around a node. Imagine further a different processor which computes the amount of space around that node occupied by its ports. That processor

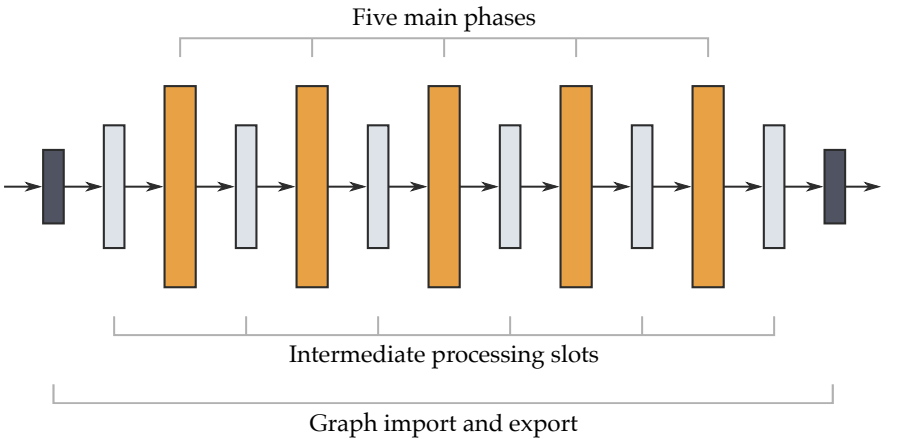


Figure 3.9. ELK Layered is structured according to the five phases of the layered approach (light colored boxes). Before, between, and after the phases are *intermediate processing slots* that any pre and post processing can take place in. Since the algorithm uses a custom graph data structure, it is surrounded by import and export code.

will of course have to be executed after the ports have actually been placed, not before. Managing these dependencies is somewhat simplified by the fact that most processors can only be sensibly executed between a certain pair of phases, thus drastically reducing the number of possible dependencies.

Which intermediate processors need to be executed depends both on the active phase implementations as well as on the features of the input graph. If there are no edge labels, for instance, there is no use executing a processor that takes care of them. By taking such cases into account, ELK Layered dynamically adapts itself to the layout problem at hand. The intermediate processing pattern has proven so useful over the years that it is now available as a general framework to other ELK-based layout algorithms.

For the rest of the section on the layered approach, we will look at how ELK Layered solves some more specific problems we will refer back to in subsequent sections.

Table 3.1. Implementations for the different phases of the layered approach available in ELK Layered.

Phase	Implementations
Cycle breaking	Greedy Interactive
Layer assignment	Coffman Graham Interactive Longest path Minimum width Network simplex Stretch width
Crossing minimization	Interactive Layer sweep
Node placement	Brandes Köpf Interactive Linear segments Network simplex Simple
Edge routing	Orthogonal Polyline Spline

3.1.7 Ports

At its inception the layered approach did not have any concept of restricting where edges could connect to nodes—the important thing was that they did connect. Over time, ports crept up in some of the research. Gansner et al. [GKN+93] have ports with fixed positions built into their node placement algorithm. The port side cannot be influenced, however: incoming and outgoing edges will always connect to the node's left and right side, respectively. Sander alleviated this restriction by proposing different levels of *port constraints* that define where ports can be placed [San94]. Spönemann et al. [SFH+10] added yet more levels of constraints to meet the requirements of data flow diagrams.

In ELK Layered, port constraints are set on nodes, as the following definition captures [Spö15].

Definition 3.5 (Port constraints)**.** A *port constraint* restrict the freedom a layout algorithm has in placing the ports of a node. The set of available port constraints is

$$\mathbb{P} = \{\textsc{Free}, \textsc{Fixed Side}, \textsc{Fixed Order}, \textsc{Fixed Ratio}, \textsc{Fixed Position}\}.$$

Given a graph with ports $G = (V, E, P, \rho)$, port constraints are assigned to nodes through a *port constraint assignment function*

$$\mathrm{pc}\colon V \to \mathbb{P}.$$

The port constraints are to be interpreted as follows:

FREE	The algorithm is free to place ports anywhere it deems fit.
FIXED SIDE	The side of the node a port may be placed at is fixed, but the position along that side can be freely determined by the algorithm.
FIXED ORDER	The order of ports at each side is fixed, but the exact position is still up to the algorithm.
FIXED RATIO	The position of each port is fixed to a fraction of the length of the side it is assigned to.
FIXED POSITION	The position of each port is fixed regardless of node size.

□

Since ports can be assigned to different sides, we of course need a way to do so.

Definition 3.6 (Port side)**.** A *port side* describes which of the four sides of a node a port shall be placed at. The set of available port sides is

$$\mathbb{S} = \{\textsc{North}, \textsc{East}, \textsc{South}, \textsc{West}\}.$$

Given a graph with ports $G = (V, E, P, \rho)$, port sides are assigned to ports through a *port side assignment function*

$$\mathrm{ps}\colon V \to \mathbb{S}.$$ □

If port constraints are FREE, ELK Layered will assign ports with incoming and outgoing edges to the WEST and the EAST sides, respectively, in order to match the flow direction. As ELK Layered progresses through its phases, the constraints get more and more restrictive as the algorithm takes more and more decisions. For example, once the crossing minimization phase has determined the order of nodes in each layer, the order of ports on each node can be computed to minimize crossings and then be fixed, thus raising the port constraints to at least FIXED ORDER. Previous publications offer further details about how port constraints are integrated into the layered approach [SSH14].

3.1.8 Hierarchical Nodes

As noted in Section 2.5.2, laying out hierarchical graphs is usually implemented by laying out each simple graph separately, working from the leaves of the inclusion tree up to its root. This works perfectly fine as long as there are no hierarchical edges in the graph. If there are, this method breaks down because there is no simple graph they are completely contained in. To see why, we need to revisit the definitions of simple graphs and hierarchical edges that we encountered in Section 2.1. Let $G = (V, E, \tau)$ be a hierarchical graph, $v_0 \in V$ be a hierarchical node, and $G_{v_0} = (V_{v_0}, E_{v_0})$ be the simple graph represented by v_0. Since G_{v_0} is the subgraph of G induced by the nodes in V_{v_0} and since all nodes in V_{v_0} are children of v_0, the following property holds by definition:

$$\forall e \in E_{v_0} \colon \tau(\text{source}(e)) = v_0 \wedge \tau(\text{target}(e)) = v_0.$$

That is, each simple graph contains only simple edges. If this was the whole truth regarding hierarchical layout, the consequence would be that hierarchical edges would simply be neglected. They would end up all over the place, but not be properly routed through the diagram. This is in fact the case for layout algorithms that have no concept of hierarchy, but cannot be true for algorithms that do.

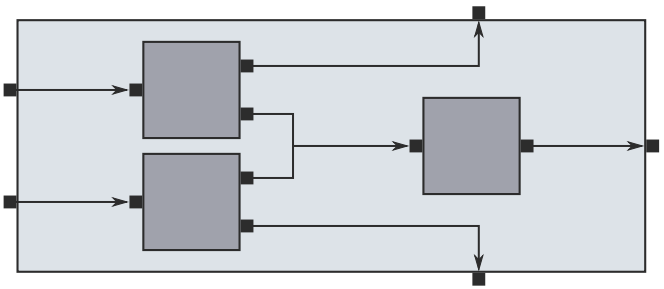
If ELK Layered was set to work on G_{v_0}, it would take into account more edges:

$$E'_{v_0} = E_{v_0} \cup \{e \in E : \text{source}(e) = \tau(\text{target}(e)) = v_0 \\ \vee \tau(\text{source}(e)) = \text{target}(e) = v_0\}.$$

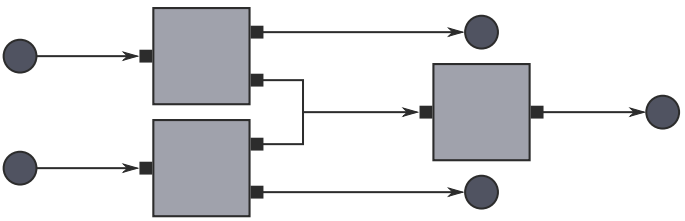
It would lay out not just the simple edges, but also short hierarchical edges, that is, edges that connect v_0 to one of its children [Sch11]. Doing so makes sense: these edges will have to be drawn in the diagram produced for G_{v_0} anyway.

This still leaves out long hierarchical edges: the kind which connects a node $v \in V_{v_0}$ to a node that is neither v_0, nor itself part of V_{v_0}. This is where ELK Layered's *hierarchy mode* comes in handy. In hierarchy mode, the algorithm is run on a whole subtree of the inclusion tree. It can thus include any edge whose end points are part of that subtree. Internally, however, ELK Layered reduces this rather complex case to the simple one described previously by temporarily splitting long hierarchical edges at the hierarchy boundaries. For the remainder of this section, we can thus limit the discussion to simple graphs plus short hierarchical edges.

As soon as ELK Layered finds a hierarchical edge, it "imports" all ports of v_0 into the current layout graph. Importing them in this case means creating dummy nodes to represent both the simple and the hierarchical ports. While it may seem counter-intuitive to import not just the hierarchical ports, this is in fact necessary. Imagine that our hierarchical node v_0 had three ports p_1, p_2, p_3 assigned to the NORTH side and with port constraints set to FIXED ORDER. Let the order they should appear in be the order we just listed them in. Now assume that p_1 and p_3 have connections to children of v_0 and are thus proper hierarchical ports, but p_2 does not. If we only represented p_1 and p_3 while laying out G_{v_0}, we would certainly adhere to the ordering constraints between the two, but might end up placing them in such close proximity that there would not be any space left for p_2. Even worse, once the child graph's layout has been computed, hierarchical ports cannot be moved anymore so as not to mess up the edge routing inside. This means setting port constraints to FIXED POSITION, which implies that p_2 must have had a position calculated for it in order to be placed properly.



(a) Hierarchical node with hierarchical ports



(b) Internal representation with dummy nodes

Figure 3.10. The way ELK Layered handles hierarchical ports. **(a)** The final drawing of a hierarchical node. **(b)** In the internal representation, all hierarchical ports have dummy nodes created for them.

We conclude that we need to import all ports as soon as a node has at least one hierarchical port.

As Figure 3.10 shows, the way dummy nodes are created depends on the side of the node the ports will end up at [Sch11]. For ports at the WEST and EAST sides, dummy nodes are created and placed in a separate first and last layer, respectively. For ports on the NORTH and SOUTH sides, dummy nodes are created and marked such that they will end up at the top or bottom of their respective layers.

Hierarchical edges are imported in a way that they connect one of the hierarchical port dummies with one of the nodes in V_{v_0}. A layout is then

computed as usual, except for some special provisions to cater to fixed hierarchical port positions. In this case, the algorithm needs to ensure that the dummy nodes are placed at the correct coordinates after node placement to not mess up the edge routing later.

Once a layout is computed, the dummy node positions are applied to the external ports and routing information are applied to the imported hierarchical edges. Note that all port constraints are supported for hierarchical ports. An important implication is that from this moment on, both the size of v_0 as well as the positions of all of its ports are fixed.

3.2 Micro Layout of Nodes

With an explanation of the layered approach out of the way, let us now take a moment to look at the diagrams in Chapter 1 again. It is obvious that nodes are more than just rectangular shapes that serve as mere end points for edges. Indeed, nodes consist of the following graphical elements, which, with the exception of the client area, can all be found in the Ptolemy II block diagram in Figure 1.2:

Outline The basic shape of the node. In Ptolemy II's block diagrams, this is usually a rectangle (as the term "block" helpfully suggests), but other languages may use different outlines to distinguish different types of nodes. Flowcharts, for example, use rhombi to distinguish decision nodes.

Node labels An arbitrary number of labels displaying text that serve to describe the node or show information related to it. In Figure 1.2, a single node label above each node shows the node's name, which defaults to its type. Simulation environments such as ETAS EHANDBOOK[1] can use node labels to display a node's current state during simulations, as shown in Figure 3.11.

Ports Graphical representations for a node's ports. The example diagram uses triangles with different fill colors to distinguish different kinds of

[1] https://www.etas.com/en/products/ehandbook.php

ports. Even if a graphical language uses ports, however, they may not have explicit graphical representations.

Port labels Just like nodes, ports can have an arbitrary number of labels. In the example diagram, the labels describe the purpose of each port to prevent the user from shoving a stream of integers into a port which actually expects boolean values. Again, simulation environments can use port labels to display data arriving at or leaving a port.

Additional graphics Additional graphical elements inside a node's outline that do not fall into any of the categories above. In Figure 1.2, these are used along with the aspect ratio of each node's outline to distinguish different types of nodes.

Client area The client area of a node is a rectangular area in its center not occupied by either labels or ports. This area, if it exists, can be used for other purposes, such as additional graphics or to place a child graph if the node happens to be hierarchical.

The exact size and placement of the additional graphics are in the hands of the application. Determining the position of the node itself, however, is at the core of the layout problem. We will call this the *macro layout* of a diagram. The positions of node labels, ports, and port labels are not part of the macro layout (except for hierarchical nodes, as discussed in Section 3.1.8). Determining these values is what we call the *micro layout* of a node. While all layout algorithms support computing a macro layout (after all, that is the very definition of the layout problem), few support computing a micro layout. In fact, they might not even support nodes that are more than a dot on the plane.

In using the terms "macro layout" and "micro layout," we borrow from definitions introduced by Schneider et al. [SSH13], but apply them to a different context. Schneider et al. defined the terms in the context of the KLighD framework, where micro layout refers to positioning what we call a node's additional graphics, and where macro layout refers to whatever the layout algorithm does to the diagram. Here, we drill down into the latter and apply the terms afresh: the part of the layout process where we

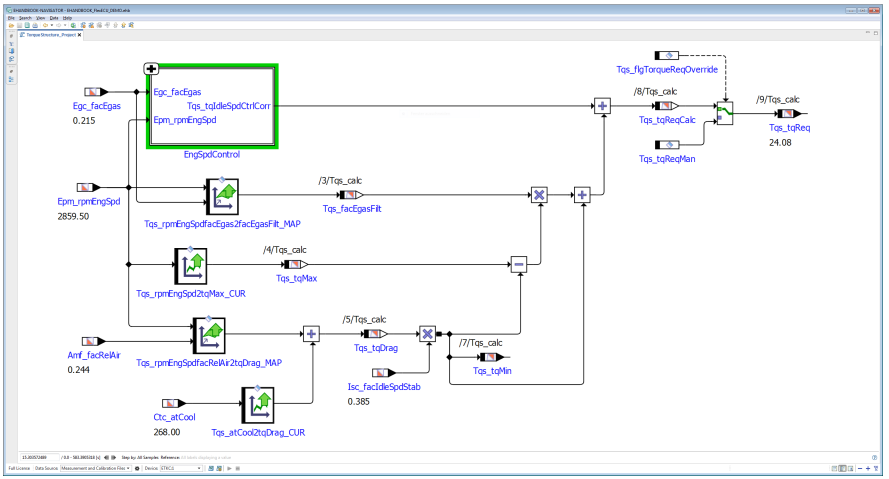


Figure 3.11. ETAS EHANDBOOK, a tool for car engine calibrators that allows them to interactively browse through documentation that contains data flow diagrams, uses node labels to show measurements during simulation runs. See, for example, the numbers below the labels of the leftmost nodes.

place each node's ports and labels is what we now call the micro layout, while placing nodes and routing edges is what we call the macro layout. Put differently, we use the idea of distinguishing between a more global and a more local problem and apply it to what Schneider et al. originally considered the more global problem.

Whether or not an application computes micro layouts itself or leaves that to the layout algorithm depends on the application. What the layout algorithm usually has to support, however, is to take the micro layout into account during its layout decisions. For example, suppose that a node v_1 has three ports on its EAST side, each with a long label placed next to it outside the node. If a layout algorithm knows about v_1's micro layout, it will make sure to place another node v_2 a good distance apart such that it does not overlap any of the labels. If on the other hand a layout algorithm only considers v_1's outline, it may end up placing v_2 on top of the labels, making them unreadable. Users will not be happy about the result and may

Figure 3.12. A spectrum with examples of how layout algorithms support taking the size of nodes into account when placing them, from no to full support.

end up not pressing the layout button ever again, which in turn makes the layout algorithm developers unhappy. Proper support for micro layout is thus important.

The extent to which layout algorithms pay regard to the size of nodes lets us place them along a spectrum, as shown in Figure 3.12. At one extreme, force-directed layout algorithms initially modeled nodes as mere points on the plane [Ead84]. What may suffice for visualizing the structure of social networks will certainly not suffice for all types of applications; indeed, for flow-based diagrams it does not.

Moving down the spectrum, other layout algorithms drop the notion of nodes as points, but continue to require them to have the same size, or at least treat them as if they did [STT81]. If this requirement is not met, clients can work around that fact by enlarging all nodes to a uniform size. This works well enough if node sizes do not differ too much, but layout quality deteriorates once they do, usually at the expense of too much white space.

It is interesting to note that the layered approach started out exhibiting this problem both horizontally and vertically. As far as the horizontal case is concerned, the nature of the approach requires nodes to be partitioned into layers that are not allowed to overlap in the final drawing. The width of each layer is determined by its widest node, thereby potentially causing the abundance of white space just lamented, as Friedrich and Schreiber observed [FS04]. The vertical direction does not suffer from such a conceptual problem and it was easy enough to adapt node placement algorithms to different node sizes [GKN+93].

This neatly takes us further down the spectrum, where many of today's algorithms allow nodes to differ in size. That size becomes part of the layout problem's specification, never to be changed again (at least not while the layout algorithm is running). This approach works well if node sizes are already known and fixed prior to calling the layout algorithm, as they often are in interactive editing scenarios. Diagram editors such as *yFiles yEd*[2] allow users to change the size of nodes as they see fit, while editors such as *Papyrus* make each node large enough to accommodate all of its information while the user edits the diagram. These applications effectively compute each node's micro layout themselves.

If an application only has structural information about a node, such as the ports and labels it is supposed to have, it may want to leave computing the micro layout to the layout algorithm. Layout algorithms which can do so constitute our next stop along the spectrum. A popular specimen of this kind of algorithm is *Graphviz dot*, which allows users a lot of freedom in this regard, as Figure 3.13 shows. A node's label can be composed of rather complex table-like structures, even allowing HTML-like text and tables, with the node made large enough to display the label in its entirety. What's more, different parts of the label can be designated as the node's ports. This kind of node size calculation is not intrinsically linked to the inner workings of a layout algorithm and can well be performed before the algorithm itself kicks in. This way, layout algorithms that already support fixed node sizes can be augmented to also support micro layout computation.

Our final stop along the spectrum is ELK Layered. Its micro layout system, called the *cell system*, is an implementation of the algorithm to be described in this section. The cell system can use information that become available gradually while the algorithm is running to determine the positions of ports and labels, but it can also be used as a preprocessing step prior to executing other layout algorithms.

The cell system does not support the kind of complex table layout features that Graphviz supports. This is because of four reasons. First, ELK's graph data structure has explicit support for modeling ports and through port constraints requires more positioning flexibility than Graphviz's table

[2] `https://www.yworks.com/products/yed`

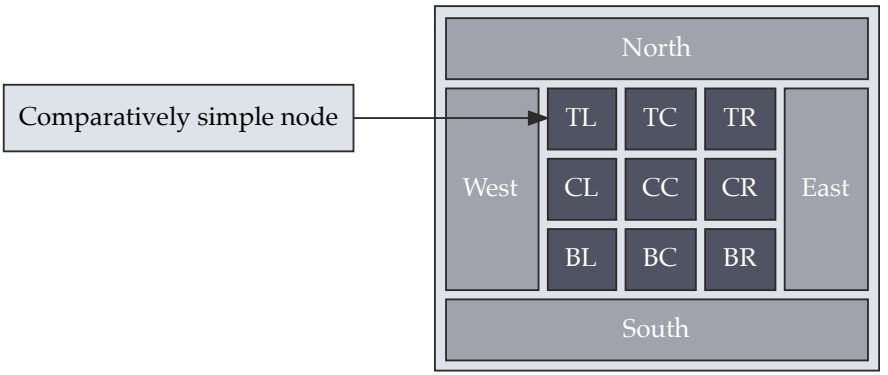


Figure 3.13. This figure was produced by Graphviz and shows the extensive freedom it gives its users to design the appearance of nodes, going to great lengths to support even complex tabular structures. Cells in such structures can be designated ports for edges to connect to.

layout would be able to provide. Second, extending Graphviz's system to properly support port constraints as we defined them would be a considerable undertaking. It would not not even be clear where to stop; for example, if there is already support for HTML tables, why not support Cascading Style Sheets (CSS) as well? Third, Graphviz is not limited to computing layout information, but can also render the results. Its table layout thus provides ways to influence not only how things are positioned, but also what they should look like, which adds further complexity that ELK has no need for. Finally, any application that requires micro layouts complex enough to justify such an elaborate system might be better off just implementing a custom and comparatively simple micro layout algorithm.

The cell system specifically supports the requirements of the ELK graph data structure, including ports and port constraints. Throughout the rest of this section, we will look at node labels, ports, and port labels in turn and think about possible placement requirements. We will then look at a number of options to influence how the size of a node is calculated before discussing the cell system, which implements these options and requirements.

3.2.1 Node Labels

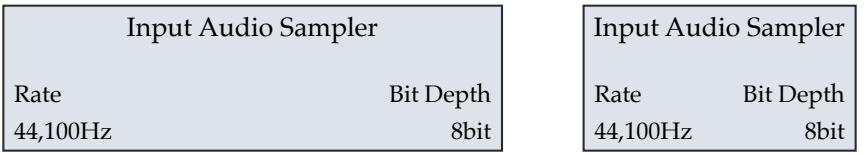
Dismissing a full-blown table-based implementation does not mean that there cannot be any degree of freedom left. The first is whether a node label is to be placed inside or outside of its node.

If a label is to be placed on the inside, two additional degrees of freedom are its horizontal and vertical alignment. As mentioned before, the cell system is not supposed to support an arbitrary table-like grid structure as Graphviz dot does. Instead, it allows for a fairly typical set of placement options: left, centered, and right for horizontal as well as top, centered, and bottom for vertical alignment. This yields the grid of nine possible label locations shown in Figure 3.14. Since there can be labels at multiple locations, there must be a configurable amount of space between each adjacent pair of grid locations (in CSS, this would correspond to a table's border spacing). Also, the amount of space between the grid and the node boundary must be configurable for all four sides (in CSS, this would correspond to a table's padding).

In a strictly tabular grid, the height and width of a location as determined by the labels placed therein will influence the height and width of other locations in the same row or column, respectively. If a location is larger than required by its label, the label is of course placed according to its horizontal and vertical alignment inside the space available at its location. We will call this *strict mode*, and while it can work well in some use cases, it may be utterly confusing in others.

For example, the node in Figure 3.15a uses a small set of labels to display its name as well as the values of two parameters, laid out using the grid structure from Figure 3.14. With the technical background just acquired, it is clear to us that the two labels in the bottom row avoid the space below the top label because the left and right columns have no way of bleeding into the center column. Just looking at the result, however, there does not seem to be any reason why the two labels should not be moved below the node's name. This is exactly what happened to them in Figure 3.15b, thus demonstrating the grid's *relaxed mode* of operation. If relaxed mode is active, the grid is not treated as a rigid table with three by three cells, but as three independent rows. The size of a cell accordingly only influences

Figure 3.14. The nine inside and twelve outside node label locations. The three lines in each of the locations hint at the horizontal and vertical alignment of labels at each location as well as how they are stacked if there are several.



(a) Strict grid **(b)** Relaxed grid

Figure 3.15. A strict tabular structure for inside node labels may not always be desired. **(a)** The grid structure can cause awkward label placements and may use much more space than necessary. **(b)** Relaxing the grid in that each row is handled separately can help solve these problems.

the height of its siblings in the same row, but not the width of cells in other rows. Whether relaxed mode should be active or not should be configured according to the use case at hand.

If we can have a relaxed mode along the horizontal axis, there is nothing to stop us from introducing a relaxed mode along the vertical axis as well. It gets more complicated, however, if we allow both to be active at the same time. Activated on their own, both relaxed modes have the property that locations never overlap: in the horizontal case, for example, location s in each row never overlap, and rows do not overlap either (the same holds for columns in the vertical case). If we relax along both axes, locations could very well overlap. Preventing this would entail moving locations until they do not overlap anymore, which, in extreme cases, would make the grid look like it was not in any kind of relaxed mode after all. We will try to avoid this kind of complexity and keep things simple.

If a label is to be placed outside of its node, the next decisions to be made are on which of the node's four sides to place the label, and how to align it along that side. Again, going with the typical set of three alignment options yields a total of twelve outside locations shown in Figure 3.14. A label should always be placed as near to the node's border as possible while leaving a configurable amount of space between them.

Since a node can have several labels, nothing stops clients from making them share the same location. We solve that by stacking labels vertically, with a configurable amount of space between adjacent labels.

Before moving on to ports and port labels, we have yet to determine how to specify a label's desired target location. First, let us define the following three sets:

$$\begin{aligned} \mathit{Area} &:= \{\textsc{Inside}, \textsc{Outside}\} \\ \mathit{Alignment}_H &:= \{\textsc{Left}, \textsc{Centered}, \textsc{Right}\} \\ \mathit{Alignment}_V &:= \{\textsc{Top}, \textsc{Centered}, \textsc{Bottom}\} \end{aligned}$$

Given a set of node labels L, the target location for each label could then simply be a three-tuple involving the sets above:

$$\mathit{Area} \times \mathit{Alignment}_H \times \mathit{Alignment}_V.$$

With this definition, however, we run into a problem. Consider the tuple $(\textsc{Outside}, \textsc{Left}, \textsc{Top})$. Does it refer to the leftmost label location on

the NORTH side, or to the topmost location on the WEST side? We need a way to disambiguate these cases and do so by introducing a fourth set:

$$\mathit{Priority} := \{\textsc{Horizontal}, \textsc{Vertical}\}.$$

Horizontal priority means that the horizontal alignment determines whether the label will be assigned to an EAST or WEST location, while vertical priority means that the vertical alignment determines whether the label will be assigned to a NORTH or SOUTH location. These considerations lead to the following definition:

Definition 3.7 (Node label placement assignment)**.** Given a set of node labels L, node label placements are assigned to labels through a *node label placement assignment function*

$$\mathrm{nlp}\colon L \to \mathit{Area} \times \mathit{Alignment}_H \times \mathit{Alignment}_V \times \mathit{Priority}. \qquad \square$$

Note that instead of introducing the *Priority* set, we could have redefined *Area* to distinguish between all four outside areas. This, however, would have introduced confusion as to what it means for a label assigned to a western location to be left-aligned or right-aligned.

As a final remark, note that there are two configurations that still seem problematic:

$$(\textsc{Outside}, \textsc{Centered}, \textsc{Centered}, \textsc{Horizontal}),$$
$$(\textsc{Outside}, \textsc{Centered}, \textsc{Centered}, \textsc{Vertical}).$$

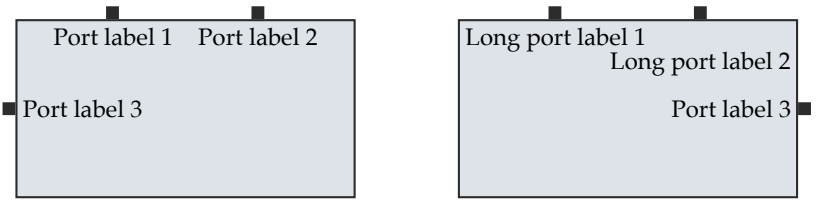
Centering an outside label along both axes would end up placing it inside the node. We thus consider these configurations to be illegal.

3.2.2 Ports and Port Labels

As with node labels, the basic degree of freedom when placing port labels is whether to place them inside or outside of their node:

Definition 3.8 (Port label placement assignment)**.** Given a graph with ports $G = (V, E, P, \rho)$, port label placements are assigned to nodes through a *port label placement assignment function*

$$\mathrm{plp}\colon V \to \mathit{Area}.$$



(a) Sufficient space **(b)** Insufficient space

Figure 3.16. Placing port labels inside their node. **(a)** Doing so works well if there is enough space available. **(b)** If this is not the case, especially labels of NORTH and SOUTH ports can overlap. Depending on the situation, it can be possible to offset such labels vertically.

The port label placement assigned to a node applies to all of its port labels. □

When placing port labels inside simple nodes, they can easily be placed next to the port, as shown in Figure 3.16a. We need to be careful with NORTH and SOUTH ports, though. For them, it can quickly lead to overlapping labels if there is not enough space (which users, as it turns out, are not particularly fond of). As Figure 3.16b suggests, offsetting the labels vertically to remove overlaps can improve the situation somewhat. With hierarchical nodes, the labels cannot always be placed next to the ports because that is where they may be approached by edges (although placing labels there anyway would be an easy way to achieve a strike-through kind of font style). In these cases, we can simply offset the label enough to avoid incident edges.

When placing port labels outside, we run into the same problem. The labels thus need to be offset, with a choice between two directions to offset them in. The goal must be to make the association between a port and its label as obvious as possible. As Figure 3.17 shows, how that goal can best be achieved depends on the number of ports along a given side. If there are two ports (NORTH and WEST sides in the example), each port's label can be offset away from the other port. If there are more (or fewer) ports (SOUTH and EAST sides), a uniform offset direction ensures clear associations.

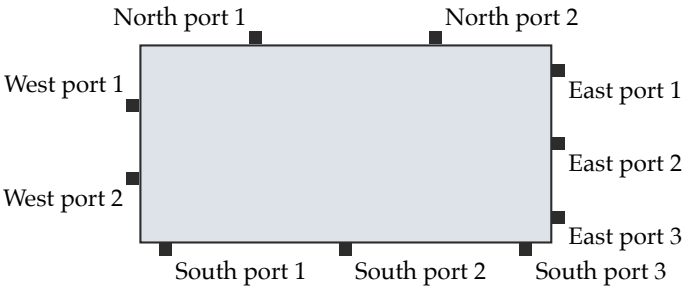


Figure 3.17. If port labels are to be placed outside of their node, we usually place them with a uniform offset per port side. If there are only two ports on a side, however, we take advantage of the opportunity to make it even clearer which label belongs to which port.

Note that while perhaps not as neat, the amount of space required for ports and their labels can be reduced by allowing an outermost port's label to break the uniformity. For example, by placing the *South port 1* label in Figure 3.17 to the left of its port instead of to the right, the port could be moved a lot closer to the middle port, thus allowing the node to be smaller.

Whatever port label placement clients settle on, the most comfortable situation we can find ourselves in is if ports can be placed far enough apart for the port-label associations to be perfectly clear. This is the case if every label is closest to the port it actually labels while being noticeably further from other ports. Whether or not we have this freedom mainly depends on the port constraints: if set to FIXED POSITION, there may be little we can do. For inside port labels on the NORTH and SOUTH sides, we can try our best to offset them to avoid overlaps (since edges usually do not cross port labels placed inside, label overlaps are all we need to worry about in this case). For outside port labels on these sides, we may attempt to at least offset one of the labels along another direction as it normally would be to avoid it being crossed by an edge connected to another port (see Figure 3.18).

While other port constraints may appear to give us more leeway, we may still be constrained by a fixed node size. How clients can give us the freedom to determine the node size such that we can achieve an optimal port and label placement is the subject of the next section.

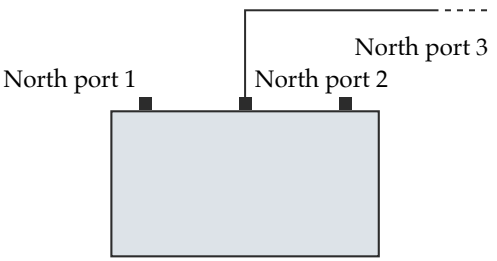


Figure 3.18. With three ports on the NORTH side, we would usually try to be consistent in our choice of which port side to place the labels on. In this case, however, we can avoid an edge-label crossing in exchange for breaking consistency. It would be easy to extend this example to the point where we have no way of avoiding edge-label crossings.

3.2.3 Configuring Micro Layout

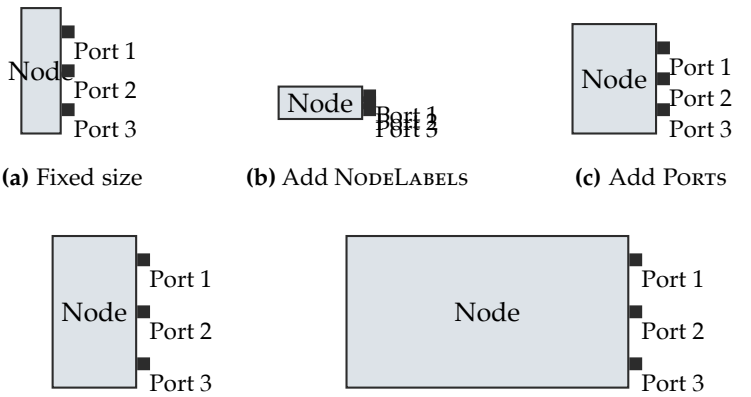
We have already encountered opportunities for clients to influence how node labels, ports, and port labels are placed. Arguably the most important one is whether we are allowed to make the node large enough to properly place all of those elements while adhering to spacing constraints. This is what the following two definitions, illustrated in Figure 3.19, aim to capture.

Definition 3.9 (Size constraints)**.** *Size constraints* restrict the freedom a layout algorithm has in resizing a node and are defined as follows:

$$\mathit{SizeConstraints} = \{\textsc{NodeLabels}, \textsc{Ports}, \textsc{PortLabels}, \textsc{MinSize}\}.$$

The size constraints are to be interpreted as follows:

NODELABELS	The node can be made large enough to place all node labels without violating spacing constraints.
PORTS	The node can be made large enough to place all of its ports without violating spacing constraints.
PORTLABELS	If PORTS is active, not only the ports themselves are considered, but also their labels.
MINSIZE	The node must meet a given minimum size. □



(a) Fixed size **(b)** Add NodeLabels **(c)** Add Ports

(d) Add PortLabels **(e)** Add MinSize

Figure 3.19. The effect of setting size constraints. **(a)** We start with empty size constraints, which means that the node's size as specified in the input graph will not be changed. **(b–d)** Adding size constraints one by one ensures that enough space is reserved for different elements of the node. **(e)** In this example, a minimum size set on the node causes it to be wider than strictly necessary.

Definition 3.10 (Size constraint assignment)**.** Given a graph $G = (V, E)$, size constraints are assigned to nodes through a *size constraint assignment function*

$$\mathrm{sc}\colon V \to \mathcal{P}\left(\mathit{SizeConstraints}\right). \qquad \square$$

Assigning empty size constraints to a node takes away all of the freedom the algorithm has to determine the node's size, thus leaving it untouched.

Clients may want to influence micro layout further. For example, a node may have to be made large enough to properly place its ports and port labels, but it might be just fine for node labels placed above it to exceed its width (Figure 1.2 shows examples of overhanging labels). The following two definitions, illustrated in Figures 3.20 and 3.21, introduce a number of options that influence micro layout.

Definition 3.11 (Size options)**.** *Size options* refine the behavior of size constraints and are defined as follows:

$$\begin{aligned} \mathit{SizeOptions} = \{&\textsc{ClientAreaMinSize}, \textsc{PortsOverhang},\\ &\textsc{LabelsOverhang}, \textsc{EqualPortSpacing},\\ &\textsc{CompactPortLabels}, \textsc{StrictGrid},\\ &\textsc{Asymmetrical}\}. \end{aligned}$$

The size options are to be interpreted as follows:

ClientAreaMinSize	Applies the minimum size not to the node as a whole, but to its client area. The node as a whole will usually end up exceeding the minimum size.
PortsOverhang	Allows ports to extend beyond the node's boundaries if there is not enough space available to place them without violating spacing constraints.
LabelsOverhang	Allows node labels placed outside to exceed the node's dimensions.
EqualPortSpacing	If not set, ports will be placed as near to each other as spacing constraints and size constraints allow. To avoid uneven spacing due to different port or port label sizes, setting this option causes the space between all ports to be increased to the point of being uniform.
CompactPortLabels	If active, preference is given to port label placements that take up less space over consistent decisions as to which side of a port each label is placed.[3]
StrictGrid	By default, inside node labels are placed using the relaxed mode grid. If this option is active, the grid is forced to behave strictly like a table.

[3]Since the submission of this thesis, this mode has been made the default because it always improves graphic association between ports and their labels and allows nodes to be smaller.

ASYMMETRICAL — Allows the outer columns and the outer rows of the inside node label grid to differ in width and height, respectively. If this is the case, labels placed in the inner columns, which would usually be centered horizontally, can end up being placed off center. □

Definition 3.12 (Size options assignment)**.** Given a graph $G = (V, E)$, size options are assigned to nodes through a *size options assignment function*

$$\mathrm{so}\colon V \to \mathcal{P}\left(\mathit{SizeOptions}\right). \qquad \square$$

Clients may also want to control the alignment of ports along their respective port side. This is what the following definition captures.

Definition 3.13 (Port alignments)**.** *Port alignments* control how ports are aligned along their respective port side and are defined as follows:

$$\mathit{PortAlignments} = \{\textsc{Begin}, \textsc{Center}, \textsc{End}, \textsc{Distributed}, \textsc{Justified}\}.$$

The port alignments are to be interpreted as follows:

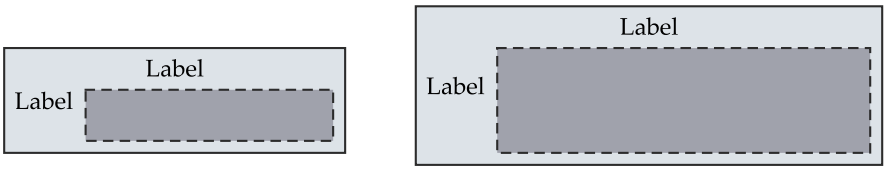
BEGIN — The ports take as little space as possible without violating other constraints and are top-aligned on vertical and left-aligned on horizontal port sides.

CENTER — The ports take as little space as possible without violating other constraints and are centered on their side.

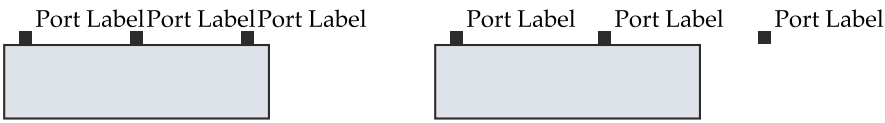
END — The ports take as little space as possible without violating other constraints and are bottom-aligned on vertical and right-aligned on horizontal port sides.

DISTRIBUTED — The ports use the whole space available to them, with the same amount of space between each pair of ports as around the set of ports.

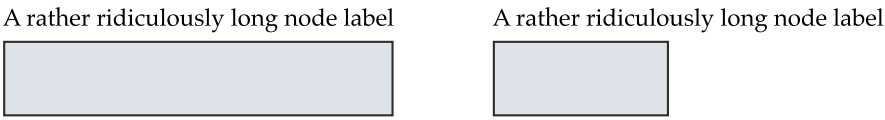
JUSTIFIED — The ports use the whole space available to them, with the same amount of space between each pair of ports, but a fixed amount of space around them. □



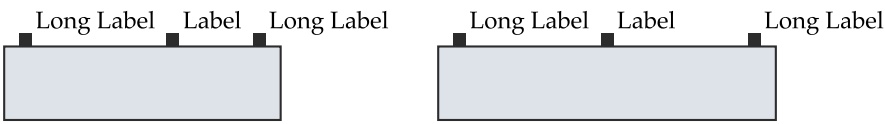
(a) ClientAreaMinSize



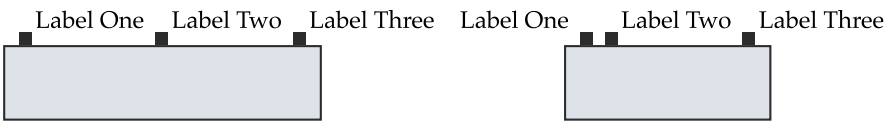
(b) PortsOverhang



(c) LabelsOverhang



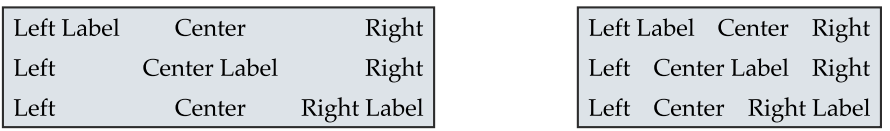
(d) EqualPortSpacing



(e) CompactPortLabels

Figure 3.20. The effect of setting the first five of the seven size options. Each example shows a node with the respective option disabled on the left side, and the same node with the option enabled on the right side. The remaining two options are shown in Figure 3.21.

(a) StrictGrid



(b) Asymmetrical

Figure 3.21. The effect of setting the remaining size options not shown in Figure 3.20. Each example shows a node with the respective option disabled on the left side, and the same node with the option enabled on the right side.

Definition 3.14 (Port alignment assignment)**.** Given a graph $G = (V, E)$, port alignments are assigned to nodes through a *port alignment assignment*

$$\mathrm{pa}\colon V \to \mathit{PortAlignments}. \qquad \square$$

According to this definition, the function assigns a port alignment to the whole node, but that could easily be extended to support different alignments for different port sides. In the interest of a simpler discussion, we will limit ourselves to this definition.

When placing ports at a given side of the node, we usually want to leave a bit of space between the outer ports and the side's boundaries (the Distributed alignment being an exception). We call this the *surrounding port space* and define it separately for horizontal and vertical port sides, as shown in Figure 3.22.

We will return to further explanations of the options introduced in this section once we take a detailed look at the cell system in the next section.

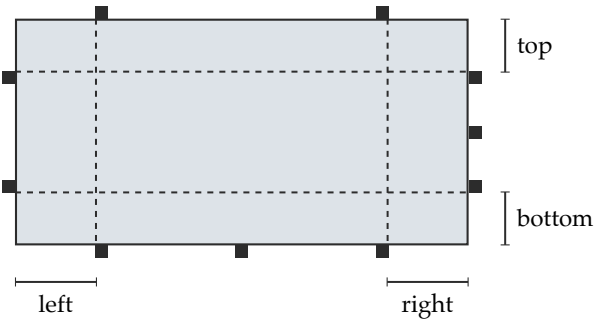


Figure 3.22. The surrounding port space defines how much space to leave between outer ports and the boundaries of the port side they are placed at.

3.2.4 The Cell System

The basic idea of the cell system is to model a node's micro layout in terms of cells placed in containers. The complex task of computing a micro layout is then reduced to configuring the cells and containers properly and letting the cell system compute everything according to then comparatively simple rules.

As the class diagram in Figure 3.23 shows, everything in the cell system is in fact a cell. Some of these can act as containers for other cells. Cells have several interesting properties:

- A *cell rectangle* which describes the cell's coordinates and size. Computing this is one of the major goals of the cell system.

- A *padding* which describes how much space to leave between the borders of the cell's rectangle and its actual content.

- Horizontal and vertical *size contribution flags* which describe whether a cell should contribute to the minimum size of its container. In Section 3.2.5 we will see how these flags are used to implement the different node size constraints.

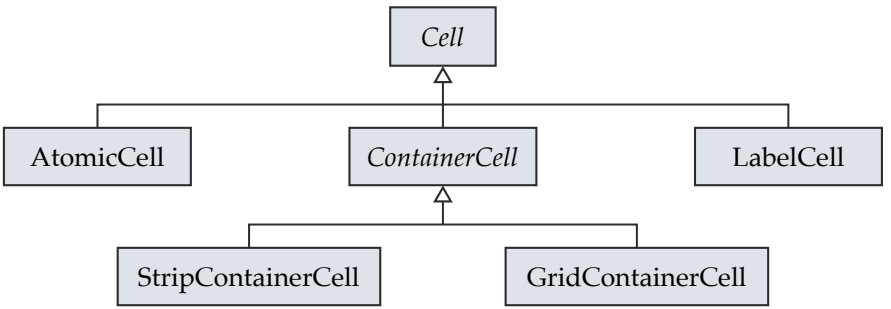


Figure 3.23. Class diagram of the cell system. Everything is a cell, but we distinguish between container cells and non-container cells.

▻ A *minimum size* which describes the minimum amount of space a cell will need to properly layout all of its content. Each type of cell handles computing this size differently.

Let us examine each of the different types of cells before the next section examines how they can be combined to solve the micro layout problem.

Non-Container Cells

Non-container cells are cells that cannot themselves contain other cells.

First up are *atomic cells*, whose only property is the minimum size of its *content area*, that is, the part of its cell rectangle that remains once the padding is subtracted. Atomic cells can be used as placeholders for things that will eventually take up space.

Label cells, the second type of non-container cell, are not just placeholders, but can instead actually contain labels. In that sense they could conceivably be considered containers, but we do not since they cannot contain other *cells*. Their minimum size is defined simply as the minimum amount of space required to place all labels below one another while leaving configurable gaps between them. Once a cell's coordinates and size have been computed, it can place its labels according to configurable horizontal and vertical alignments.

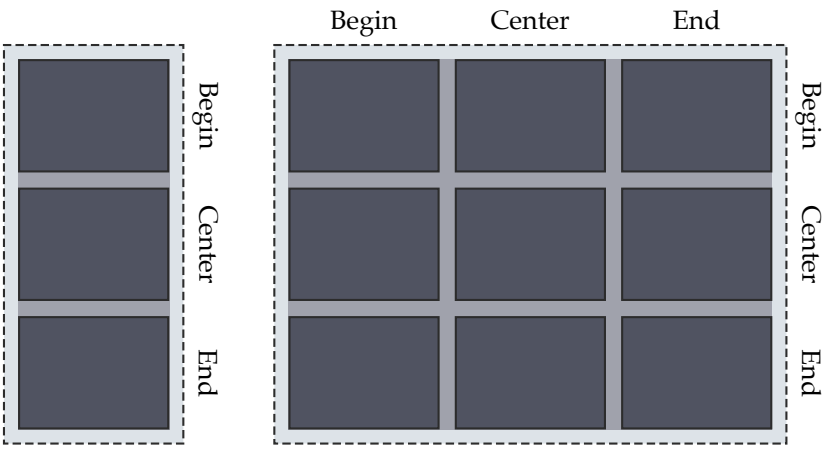
Container Cells

Container cells are containers for up to three other cells along each dimension (three cells suffice to model the label locations shown in Figure 3.14). The child cells may of course be container cells themselves. A container cell's minimum size is a function of the minimum size of those children whose *size contribution flags* are set, while the remaining children are ignored. We will see how this is useful once we discuss how the cell system is actually used in Section 3.2.5.

Similar to label cells, a container cell can lay out its children once the size and position of its cell rectangle are known. Due to the way the algorithm to be described in the next section works, the layout process is split into horizontal and vertical layout. There are two types of container cells.

A *strip container cell* provides space for three children which it lays out along a horizontal or vertical strip, as shown in Figure 3.24a. Assuming a vertical strip, it computes its minimum width by finding the largest minimum width among those children which have their width contribution flag set. Its minimum height is computed as the sum of the minimum heights of those children which have their minimum height contribution flag set, plus a configurable gap between them. An implication is that if the two outer cells have different heights, the cell at the center will not end up actually being in the container's geometrical center. To avoid this situation, the strip container can be put into a symmetrical mode. If engaged, it will not use each outer cell's individual height, but will instead apply the maximum of their heights to both outer cells. The algorithm will engage or disengage that mode based on the Asymmetrical size option.

If a strip container cell ends up being larger than its minimal size, the additional space is awarded to its center cell. If, however, it ends up smaller the question is which cell to subtract the space from. The cell system's answer is: none. Each child of a container cell is always guaranteed to at least meet its minimum size, which changes the question to how its children are laid out if there is not enough space to properly do so. Again assuming a vertical strip, the top cell is always placed at the top border of the container's content area. Correspondingly, the bottom cell is always placed at the bottom border. The remaining space, which may well be



(a) Strip container cell **(b)** Grid container cell

Figure 3.24. Layout of the two different kinds of container cells. Both are surrounded by a configurable padding (light background) and can leave a configurable gap (medium background) between their child cells (dark background). **(a)** The strip container cell, here shown as a vertical strip, provides three slots that house one child cell each. **(b)** A grid container cell can house a matrix of three times three children.

"negative," is what the remaining cell will be centered on. This of course creates overlaps, but will make sense later when we describe how to use the cell system to compute micro layouts.

A *grid container cell* extends the idea of the strip container cell into two dimensions by providing three rows and three columns of cells, but adds another feature: the space reserved for the center cell can have a minimal size set, and the container cell can be configured such that only this minimal size is used as the container's minimal size. We will later use this feature to implement certain combinations of size constraints. The grid can either be treated as a strict tabular structure or as a relaxed grid as explained before in Section 3.2.1.

Using the cell system is a matter of instantiating and populating one of the two types of container cells, configuring size contribution flags and properties for all of the cells, and calling layout methods on the root container. This will end up assigning coordinates and sizes to all cells, to be used for the task at hand—which in the next section will be the computation of a node's micro layout.

3.2.5 A Micro Layout Algorithm

The micro layout algorithm is split into the following seven phases, which we shall go through from start to finish.

1. Cell system setup
2. Padding and minimum client area size
3. Space required to place ports
4. Size contribution flags configuration
5. Node width and horizontal ports
6. Node height and vertical ports
7. Label placement

Phase 1: Cell System Setup

Much like life on Earth, the life of our algorithm starts with the appearance of the first cells. Since we can hardly wait billions of years for that to happen, creating cells is the responsibility of the first phase, resulting in the setup shown in Figure 3.25.

We start by creating the root container, a vertical strip container cell which will represent the node itself. The container's outer areas are populated with simple atomic cells that will later be used to reserve space for inside port labels for NORTH and SOUTH ports. It is important to realize that the atomic cells do not contain labels themselves. They are much rather placeholders used and configured by the algorithm's subsequent phases.

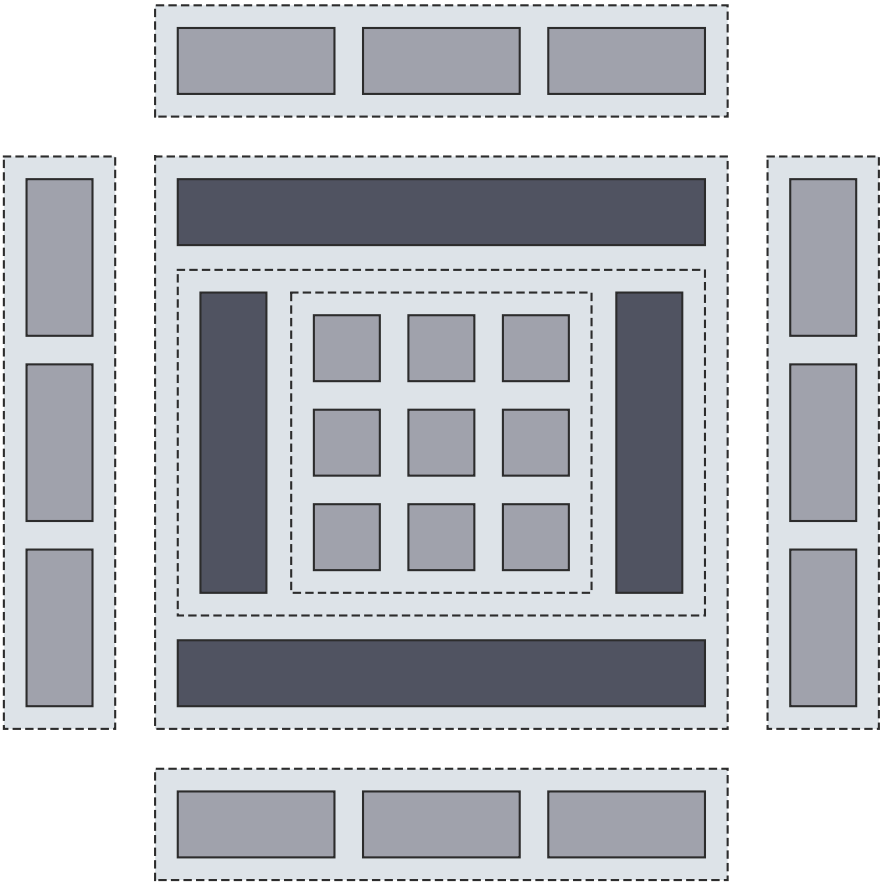


Figure 3.25. Container cells (light background), label cells (medium background), and atomic cells (dark background) are combined to model the node's micro layout. Note that besides the big root container that models the node's insides, there are four containers for label cells that hold outside node labels.

Nested in the container's center spot is a horizontal strip container cell, whose outer areas are, again, populated with atomic cells to reserve space for inside port labels for EAST and WEST ports. At this point, we can already calculate the width of the content areas of these cells as the maximum width of any label that belongs to a port on the cell's respective port side (provided that port labels are placed inside in the first place). This covers all the cells required to model the node itself as well as its inside port labels.

To represent the space used for inside node labels, the algorithm creates a grid container cell and places it at the strip container cell's center area, putting it into strict or relaxed mode as requested by the presence or absence of the STRICTGRID option. The container's padding is set according to the client's request, thus ensuring enough space to surrounding elements (inside port labels or the node's border). Being a grid container cell, it is divided into nine areas according to the inside node label locations we defined in Section 3.2.1. For any such location, the algorithm creates a label cell on demand and populates it with all labels to be placed there. We thus end up with a grid container cell that contains a label cell for each non-empty inside node label location.

What remains to be covered are outside node labels, but they do present a bit of a difficulty. As mentioned in Section 3.2.3, we want to allow node labels placed above the node, for example, to be wider than the node itself. Given this and the fact that there is no need for them to adhere to any kind of grid structure that might govern the node's insides, it becomes obvious that outside node labels need to be handled separately. The algorithm does so by creating horizontal strip container cells for NORTH and SOUTH node labels and vertical strip container cells for EAST and WEST node labels. These containers are not added to the existing cell structure, but are kept separately. Similar to how the grid container cell was populated, label cells are added to the appropriate strip container cell areas to house all the labels to be placed at the corresponding outside node label location.

Phase 2: Padding and Minimum Client Area Size

The second phase has two responsibilities: setting the root container's padding and applying a possible minimum client area size.

The need for a padding stems from the fact that clients can set a node border offset on ports which defines the amount of space to be left between them and their node. An offset of 5, for example, causes the port to be moved away from the node. A bit more problematic are negative offsets, as these will cause a port to be moved *into* its node, possibly overlapping with inside port or node labels. To avoid this situation, the algorithm iterates over all ports and, for each port side, keeps track of the minimum port offset. If that offset is negative, its absolute is used as the side's padding and applied to the root container.

As described in Section 3.2.3, clients can set a minimum node size on the whole node. They can, however, also request the minimum size to be applied only to the node's client area, which in our cell tree corresponds to the grid container cell's center area. This, finally, is the reason for the fact that this particular type of container provides the ability to give its center area a certain minimum size, regardless of the cell it contains (if any).

Phase 3: Space Required to Place Ports

This phase is responsible for two things:

1. Calculating the amount of space required to place ports and their labels.
2. Calculating spacing information about each port to be used later when the time comes to actually place them.

We will walk through this process taking the NORTH port side as an example. Vertical port sides are handled similarly, with an important difference we will come back to at the end. Since the amount of space we need to place ports largely depends on the degree of freedom we have in doing so, the following discussion is structured around the port constraints set for the node.

For the intents and purposes of micro layout, we assume port constraints to be at least FIXED ORDER. If the universe conspires against us by setting them to FIXED SIDE or even only to FREE, the matter can be resolved rather painlessly by assigning port sides and port orders more or less arbitrarily. This should of course be considered a fallback solution only since arbitrary

assignments may not be in a macro layout algorithm's best interests—hence our initial assumption.

With constraints set to FIXED ORDER, we are free to assign arbitrary port positions later as long as the order of ports is not violated. It follows that what we want to compute is the minimum amount of space required to place the ports such that the surrounding port space, a minimum spacing between ports, and, if requested, the minimum amount of space to properly place port labels are met.

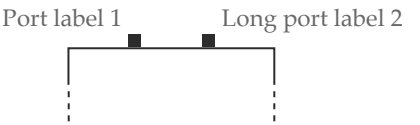
The algorithm begins with a preprocessing step that calculates the margins around each port according to Figure 3.26. These margins need to be free of other ports and their margins, but will not include any port-port spacings. They start off empty and are enlarged depending on the size constraints and size options.

If the size constraints include PORTLABELS, the margins need to include the space necessary to place the port's label. If port labels are to be placed on the outside, the port's right margin is thus enlarged by the port-label spacing and the width of the space required to place the port's labels. If this width differs among ports, this will obviously yield uneven spacings between the ports. If the EQUALPORTSPACING size option is active, we use the maximum label width over all ports instead of the port's own label width, thereby ensuring an even spacing.

There are two exceptions to this rule. First, the rightmost port is handled slightly differently. We allow its label to extend beyond the node's right border and thus do not include it in the port's right margin. And second, the left port may have its label placed on its left instead of its right side. This is true either if there are only two ports, or if the COMPACTPORTLABELS option is active. If so, following a similar argument as with the rightmost port, the leftmost port's label will not be included in its margins.

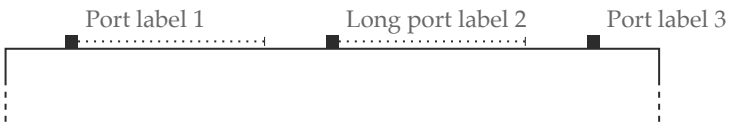
If port labels are to be placed on the inside, they will later be centered below their respective nodes. We thus enlarge the left and right port margins equally, either by the width of a port's labels or, again, by the maximum label width over all ports on that side.

Once this preprocessing step—the calculation of port margins—is complete, the minimum amount of space required to place ports with FIXED ORDER port constraints can be calculated. Assuming there to be $p \geqslant 1$ ports,



(a) Two ports

(b) More than two ports



(c) Equal port spacing

(d) Inside port labels

Figure 3.26. Port margins constitute a port's *virtual size* and include everything that needs to be taken into account by the port placement code. In the diagrams above, the dotted lines indicate the margins calculated for each port. **(a)** With only two ports and port labels outside, no margins are necessary. **(b)** With more than two ports and port labels taken into account, the margins of all but the rightmost port include port labels. **(c)** If ports should be placed an equal distance apart, the margins are wide enough to cover the widest port label. **(d)** If port labels are placed inside, the margins extend into both directions.

the space equals the sum of the widths of all ports, their left and right margins, and $p - 1$ times the port-port spacing. This sum is then applied to the width of the content area of the atomic cell which represents NORTH ports. Including the left and right surrounding port space as well, then, is simply a matter of applying them to the cell's left and right padding.

If port constraints are set to FIXED RATIO, things get more complicated because the position of each port is a function of the node's width. The objective thus is to compute a minimum width that will place the ports far enough apart to respect the spacing requirements. To calculate that width, we start by executing the same preprocessing step we used for FIXED ORDER port constraints before and end up with left and right margins for our ports. To continue, we need the following (admittedly rather simple) observation:

Observation 3.15. Given a line segment of length $l \in \mathbb{R}_{>0}$, two points whose positions along the line segment are defined by two ratios $r_1, r_2 \in \mathbb{R}$ with $0 \leqslant r_1 < r_2 \leqslant 1$, and a minimum distance to be achieved between the points $d \in \mathbb{R}$ with $0 < d \leqslant l$. Then to satisfy the minimum distance, the following must hold:

$$l \geqslant \frac{d}{r_2 - r_1}. \qquad \square$$

We apply that observation to the left surrounding port space and the leftmost port, to each consecutive pair of ports, and to the rightmost port and the right surrounding port space with the values indicated in Figure 3.27. The maximum of the calculated values is the minimum node width required to adhere to all spacing constraints and can be applied to the atomic cell for the given port side as explained before.

If port positions are fixed (that is, if the port constraints are set to FIXED POSITION), the width required to place ports along the NORTH port side is easy to calculate: it evaluates simply to the position of the right border of the rightmost port on that side. As before, we apply that value to the width of the corresponding atomic cell's content area. To ensure that we adhere to any surrounding port space, the cell's right padding is set to the right surrounding port space. Note, however, that since the leftmost port's position is already fixed, any left surrounding port space cannot be applied.

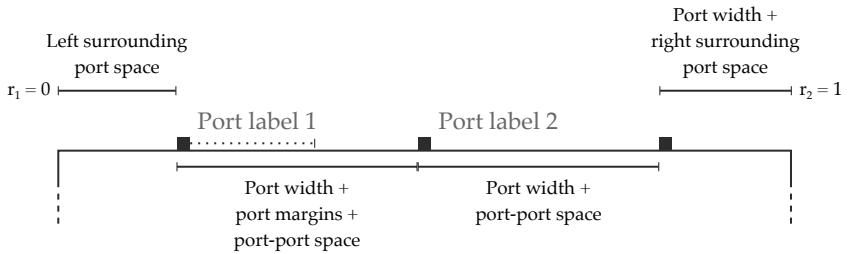


Figure 3.27. The application of Observation 3.15 to ports on the northern port side. Each of the marked distances extend from a left point with ratio r_1 to a right point with ratio r_2 which are usually ratios of ports unless noted otherwise. The text defines the value of each distance's minimum length d. The two center distances show what happens with and without PortLabels size constraints active, respectively.

For ports placed along the vertical port sides, we need to be careful. Referring back to Figure 3.25, we can spot a difference between how atomic cells for horizontal and for vertical port sides are set up: the former span the whole width while the latter are surrounded by other cells above and below. If we are free in our port placement (that is, if the placement is set to neither Fixed Ratio nor Fixed Position), the space occupied by the surrounding cells will be used to place labels of North and South ports, and we can avoid label overlaps by placing the ports only inside the space occupied by the atomic cells setup for them. If the port placement is set to Fixed Ratio or Fixed Position, however, their placement can be anywhere on their respective port sides, without regard to the extent of the vertical port sides. We will revisit this problem in subsequent phases.

Phase 4: Configure Size Contribution Flags

This step is responsible for setting the size contribution flags of all cells, and this is where using the cell system starts to pay off: different size constraints and size options can be applied simply by setting the contribution flags accordingly.

If the size constraints include PORTS, we configure the atomic cells that represent ports along horizontal port sides to contribute their width. Similarly, the atomic cells that represent ports along vertical port sides are configured to contribute their height, but only if port constraints are neither FIXED RATIO nor FIXED POSITION because only then do they accurately describe the space that will later be used to place ports along their respective sides. If this is the case, we also set the horizontal strip container cell to contribute its height to allow the height contributions of the atomic cells to propagate to the root container cell.

If the size constraints include PORTLABELS, we configure the atomic cells that represent ports along horizontal port sides to contribute their height since that will describe the amount of space used to place inside port labels. Note that in this case, their width already includes the space required for port labels. Similarly, we configure the atomic cells that represent ports along vertical port sides to contribute their width. As before, we also set the horizontal strip container cell to contribute its width.

If the size constraints include NODELABELS, we configure both the grid container cell and the horizontal strip container cell to contribute their width and height. We then iterate over all label cells that contain node labels. If a label cell is part of the grid container cell, it is configured to contribute its width and height. If it is part of an outside node label location, it is set to contribute its width and height only if the size options do not include LABELSOVERHANG, because only then does their size influence the node size.

If the size constraints include MINSIZE and the size options include CLIENTAREAMINSIZE, we first ensure that the horizontal strip container cell contributes its width and height. We then check whether the grid container cell already contributes its size. If this is not the case, we need to make sure that its center area will contribute to the size calculations later. We thus configure the grid container cell to contribute only the width and height of its center area.

Phase 5: Node Width and Horizontal Ports

Since now we have an idea of how much horizontal space we would require to properly place all ports and labels, it is time to actually go ahead and set the node's width. It should not come as a surprise by now to learn that how this is done depends on the size constraints. If they are empty, the layout algorithm is not supposed to touch the node's size in any way, so we do not. If the size constraints are not empty, the cell system will have had its size contributions set up in a way that we can simply ask the root container to compute its minimum width. If the size constraints contain NodeLabels and if the size options do not contain LabelsOverhang, we possibly increase that width to the minimum width required for outside node labels above and below the node. If there is a minimum size that applies to the whole node, we make sure not to fall short of that. We then apply the thus calculated width both to the node itself and to the root container and ask the latter to compute a horizontal layout for its children.

Actually placing the ports along the horizontal sides is comparatively simple after the extensive preparations of the third phase. If port constraints are Fixed Position or Fixed Ratio, x coordinates are already fixed or a function of the node's width, respectively. The only thing that needs to be calculated is the y coordinate, taking the port offset into account.

If port constraints are Fixed Order, matters get slightly more difficult. Since the port margins already contain all relevant spacings, all we basically need to do is place the ports such that we respect the port-port spacing between the margins of consecutive ports. This however, is only true in the following cases:

- There is more space available than the port placement requires and the port alignment is not set to Justified or Distributed.
- There is exactly as much space available as the port placement requires.
- There is less space available than the port placement requires and the size options include PortsOverhang.

If one of these cases applies, we place the ports as explained above and align the result in accordance with the active port alignment. If there is more

space available and the port alignment is set to JUSTIFIED or DISTRIBUTED, we insert additional space between each pair of ports such that the placement's resulting width equals the space available to us. If there is less space available and the size options do not include PORTSOVERHANG, we move the ports as close as necessary for the placement to fit into the available space. Of course, that will cause spacing constraints to be violated, but if that is not acceptable to a user they will have to change the configuration accordingly.

The ports are now placed, so it is time to place their labels as well. This is always easy if the size constraints include both PORTS and PORTLABELS and the port constraints are not set to FIXED POSITION: then, enough space is reserved to simply place the labels at their preferred location. If this is not the case, keeping them from overlapping is more of a challenge. We will describe a solution to this problem using inside port labels of NORTH ports as an example (labels of SOUTH ports and outside port labels are handled similarly).

First, observe that the x coordinates of the port labels are already fixed simply because inside port labels are centered below the port they belong to, which already has a position. What we can influence is the y coordinate, and that is the idea of the following algorithm: move labels along the vertical axis until they stop overlapping. We start by defining what exactly "overlapping" means in this context:

Definition 3.16 (Overlaps)**.** The function overlap : $\mathbb{R} \times \mathbb{R}_+ \times \mathbb{R} \times \mathbb{R}_+ \times \mathbb{R}_{\geqslant 0} \mapsto \mathbb{B}$ is defined as follows:

$$\text{overlap}(p_0, s_0, p_1, s_1, min) = \begin{cases} p_0 + s_0 + min \geqslant p_1 & \text{if } p_0 \leqslant p_1 \\ p_1 + s_1 + min \geqslant p_0 & \text{otherwise} \end{cases}. \qquad \square$$

The function takes a position and size for two rectangles along one of the two axes in two-dimensional space as well as a minimum spacing between them and determines whether the spacing is adhered to or not. If two rectangles overlap along the x-axis, we call that a *horizontal overlap*; if they overlap along the y-axis, we call that a *vertical overlap*. We can now formulate our problem as an optimization problem:

Problem 3.17. Given $n \in \mathbb{N}_{>0}$ rectangles $(x_i, w_i, h_i) \in \mathbb{R} \times \mathbb{R}_+ \times \mathbb{R}_+$, $1 \leqslant i \leqslant n$, defined by their x coordinates, widths, and heights. Given further a minimum spacing $s \in \mathbb{R}_{\geqslant 0}$. The objective is to calculate y coordinates $y_1, \ldots, y_n \in \mathbb{R}$ such that the height of the resulting rectangle placement

$$\max_{1 \leqslant i \leqslant n} \{y_i + h_i\} - \min_{1 \leqslant i \leqslant n} \{y_i\}$$

is minimized subject to the following constraints, which ensure that if two rectangles overlap horizontally, they are moved along the vertical axis enough to remove the overlap:

$$\forall_{1 \leqslant i \leqslant n} \forall_{i < j \leqslant n} \left(\text{overlap}(x_i, w_i, x_j, w_j, s) \Rightarrow \neg \, \text{overlap}(y_i, h_i, y_j, h_j, s) \right). \quad \square$$

Let R be the set of rectangles for which to remove overlaps and $C = (R, E)$ be an undirected graph with an edge connecting two rectangles if they overlap horizontally. We call C the *conflict graph*. If one assumes all rectangles to have the same height, the problem becomes to assign each rectangle to one of as few horizontal slots as possible—we effectively want to compute a minimal vertex coloring for C, which is NP-hard [Kar72].

For our use case, we cannot even assume all rectangles to have the same height, so we rely on a simple algorithm split into two stages. The first stage of the algorithm uses a scan line approach to compute the conflict graph. The second stage then iterates over all of the rectangles $r \in R$ ordered by their x coordinates. The current rectangle r starts off being placed at y coordinate 0. The algorithm then iterates over those of the rectangle's neighbors in the conflict graph that have already been placed, in order of ascending y coordinates. If r's current y coordinate would cause a conflict with a neighbor, r is slid down accordingly. This is a classic greedy algorithm: we slide each rectangle down until we find the first space large enough for it to fit into.

Once the labels are placed, the atomic cells for NORTH and SOUTH ports have their height updated to reflect the space used by inside port labels.

Phase 6: Node Height and Vertical Ports

On the surface, phase 6 has the same responsibilities as its predecessor, just in the vertical instead of the horizontal direction. As it turns out, however,

phase 6 is a bit more complex since it has some additional work to do, which we will walk through now.

Recall from Figure 3.25 that the atomic cells that represent East and West ports do not extend to the top and bottom margins of the root container cell. Recall also that the cells have their top and bottom padding set up to reflect the top and bottom surrounding port space. If the atomic cells for North and South ports have a height of zero, the vertical extent of the content areas of the East and West vertical cells will accurately reflect where ports can be placed. As the North and South atomic cells grow in height, however, too much space is left between the ports and the North and South node borders. One way to solve this problem would be to allow cells in the cell system to overlap. That, however, would undermine the system's main advantage, which is the simplicity of how containers compute cell coordinates. We solve it by realizing that by now the height of the North and South cells is known and thus simply adjust the padding of the East and West cells accordingly.

The next step is to compute and apply the node's height. This is very similar to what phase 5 does with the node's width. Once we know the height, however, it is time to take care of a problem generously left to us by the previous phase: since the node's height was not set yet, the y coordinates of South ports had to be calculated by assuming the node's bottom border to be at coordinate zero. We thus take the opportunity to iterate over all South ports and add the node's height to their y coordinates.

With all of this out of the way, East and West ports can now be placed, along with their labels. This works much like it does for North and South ports, except that there is no overlap removal for port labels. The labels of North and South ports had a fixed horizontal position, but could be moved vertically. This worked fine because labels are usually much wider than they are high. Labels of East and West ports, however, have a fixed vertical position and can only be moved horizontally. Doing so to remove overlaps will put rather large distances between them and their ports, which is not acceptable.

Phase 7: Label Placement

By this time, we have set up the positions of all label cells for port labels and inside node labels. What is still missing are outside node labels, since their container cells are separate from the root container. We thus start this phase by setting up the size and position of these containers.

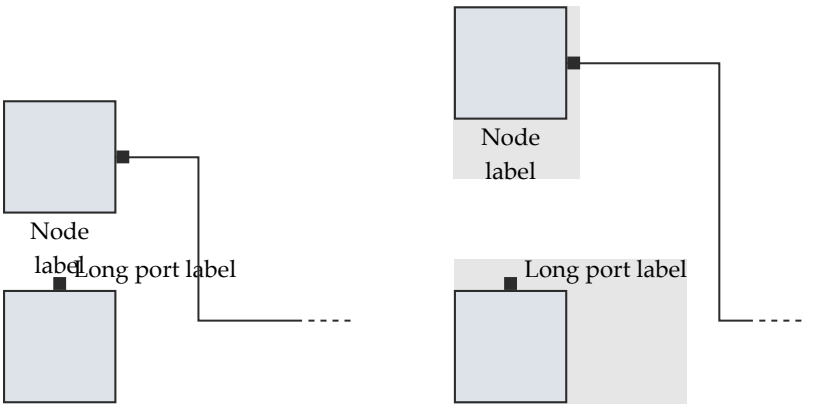
Taking the container for node labels placed above the node as an example, we start by setting its height to the height it has calculated from the label cells inside it and its width to the node's width. Its vertical position is simply a function of its height, since its padding already ensures the correct spacing between labels and the node. Its horizontal position is obtained by centering the container above the node. Finally, we ask the container to position its child cells.

All label cells now have positions assigned to them. To finally bring micro layout to a successful conclusion, we can simply iterate over all label cells and tell them to derive positions for their labels and apply them.

3.2.6 Avoiding Node Overlaps

While some nodes may have all of their labels placed inside, others may have at least some placed outside, and they are the ones that can cause problems. If a layout algorithm ignored these, this would cause violated spacing constraints and overlapping labels, as demonstrated in Figure 3.28a.

To solve this problem, we employ a solution introduced in my Diploma thesis [Sch11]. Instead of placing nodes based on their real size, we rather place them based on a (usually larger) *virtual size*, a concept related to the virtual size of ports we encountered in Figure 3.26. The virtual size of a node of course includes its real size, but also takes *node margins* around the node into account that contain all of the node's components placed around it. As Figure 3.28b shows, this keeps spacings from being violated, labels from being crossed, and everything from overlapping.



(a) Based on real size **(b)** Based on virtual size

Figure 3.28. Margins around a node include its different components, such as ports and different kinds of labels. The bounding box thus formed constitutes a node's virtual size. **(a)** Placing nodes based on their real size may result in violated spacing constraints and overlaps. **(b)** Placing them based on their virtual size yields correct and readable results.

3.2.7 Micro Layout for Hierarchical Nodes

In the previous sections, we have seen how a micro layout algorithm can place the ports and labels of a node. We silently assumed the node to be simple, but why should we discriminate against hierarchical nodes?

Computing the micro layout of a hierarchical node is in some ways different from computing one for a simple node. First, there are nodes and edges inside a hierarchical node which, beside making it hierarchical in the first place, also cause problems for micro layout because not all of the space is available to place things in. And second, a micro layout can be computed for a hierarchical node only once its child graph has already been laid out, which in ELK means that by then its size and its port positions are fixed (as we saw in Section 3.1.8). This implies that while laying out the child graph we must make the node large enough and place ports far enough apart for

micro layout to be able to place things properly. For the remainder of this section, I will suggest ways to do so.

Minimum Size

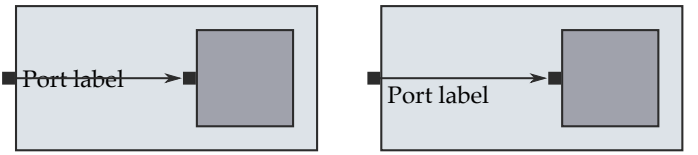
All of the elements that belong to a node can contribute to its minimum size. Many of them are not taken into account while the node's insides are laid out, however. Consider, for example, a hierarchical node that has ten ports with long port labels along the NORTH side, neither of which connected to the node's insides. With size constraints set to include PORTS and PORTLABELS, micro layout would be forced to make the node rather wide. Depending on the node's child graph, however, ELK Layered might end up assigning a much smaller width to the node. This is because while laying out the child graph, the node's ports are not represented since none have any relation to the child graph.

The solution is to run a stripped-down version of micro layout before starting the actual layout process for the child graph to obtain a lower bound on hierarchical node's size. What happens here can be understood as a simulation of what micro layout would do if invoked properly. Respecting the minimum size ensures that even though the node's size will be fixed after the hierarchical layout, micro layout will then have enough space to place everything according to the size constraints and options.

Ports and Port Labels

If port labels are placed inside the node, the node's children have to be placed far enough from the node's borders for them to not overlap with the labels. Recall from Section 3.1.8 that each hierarchical port is represented by a hierarchical port dummy node during layout. Reserving enough space for the port's label is then simply a matter of attaching that label to the dummy node, automatically causing other nodes to be moved far enough away.

Figure 3.29 demonstrates a possible problem with inside port labels we have not discussed thus far: if we placed the port label of a hierarchical port as we do for simple ports, we would cause edge-label crossings with edges connected to elements on the inside. To avoid that, we use a similar solution



(a) Bad port label placement **(b)** Good port label placement

Figure 3.29. If port labels are placed on the inside, hierarchical ports can cause problems. **(a)** Simply placing each label right next to its port can cause overlaps with edges connected to the insides. **(b)** Offsetting port labels slightly solves this problem.

as with outside port labels and offset those inside port labels that would otherwise be crossed by edges. Taking this into account is the responsibility of phases 3, 5, and 6 of the micro layout algorithm.

If port labels are placed outside, there is no need to reserve space inside the node, but we still need to ensure that the ports are spaced far enough apart to be able to place the labels properly if size constraints include PortLabels. To do so, we use the approach of adding the port labels to the hierarchical port dummies again, but with a twist. Let us think about what would happen taking a West hierarchical port as an example. If we simply added the label as is, the hierarchical node's children would be pushed away from its left border for no reason whatsoever, only resulting in superfluous whitespace. To avoid that, we simply pretend the label's width to be zero, thereby avoiding pushing the child nodes away, but still moving other hierarchical port dummies on the same port side far enough down to reserve space for labels.

Node Labels

With port labels taken care of, there is still the problem of having to reserve enough space to place inside node labels. Again, the children of a node have to be moved far enough from its borders to avoid overlapping with any labels. While this is a futile exercise for labels that are to be placed in the node's center, there is yet hope for labels placed at the remaining eight

locations. How much space will be required, however, only becomes known during micro layout and, thus, after the node's children have already been placed.

Since we already invoke the micro layout algorithm to obtain a lower bound on the node's size anyway, the solution of course is to let it compute the padding required for label placement at the node's borders as well.

3.3 Edge Label Placement

In the previous section, we have discussed node and port labels rather extensively, but many visual languages require their edges to be labeled as well. SCCharts are a good case in point: recall from Figure 1.5 that edges model state transitions that can specify both the conditions that trigger them and the side effects they cause (besides changing the active state, obviously). Of course, for a layout algorithm to be usable with such languages it needs to place edge labels properly. In an impressive display of creativity, the problem of placing edge labels is known as the *edge label placement problem*.

The origins of edge label placement stem from labeling cartographic maps. We thus start off our discussions by pondering the applicability of classic "principles and requirements" of map labeling as laid down by Imhof [Imh75]:

Legibility Labels should be easy to read and to discriminate (despite of whatever else is happening visually in the diagram), as well as be easy to find. While legibility, according to Imhof, does depend on "type form, type size, and type color," those are attributes that layout algorithms usually cannot influence. What they can influence are the positions of labels, computing them in a way that users will be able to easily recognize the labels as such.

Clear graphic association For any label there should be no question as to which of the diagram's elements it belongs to.

Minimize disturbances Labels should not overlap other elements. This seems to apply universally at first, but there are map labeling algorithms that allow lines to cross labels, although at a cost that depends on the crossing

angle: lines that run parallel to the type's base line are considered worse than perpendicular crossings [ECM+96].

Assist recognition In maps, labels for larger regions are often laid out such that they follow the region's basic shape. In contrast, labels in diagrams are usually strictly horizontal lines of text, making this point seem less relevant. However, Imhof also makes labels responsible for assisting users in recognizing the differences between objects as well as their importance. Layout algorithms in a way support the former by distinguishing different kinds of labels. The latter falls into the realm of the application since the perception of importance largely depends on the typographical properties of a label.

Type arrangement Through "variation of style and size," type helps communicate differences in classification and importance between objects to users. Again, this is usually nothing that layout algorithms have control over.

Label density Labels should be selected and arranged such that there are neither too many nor too few of them. This is not part of the core layout problem where algorithms faithfully determine positions for all elements handed to them. Instead, it bears relation to the problem of label management that we will examine in detail in Chapter 6.

From a layout algorithm's perspective, the first three principles are the most important as they are the ones that it has actual control over.

There are three basic approaches to placing labels. The first and simplest, *pre-processing*, refers to computing label positions before automatic layout is invoked and is what applications are doing when they compute micro layouts themselves. Since the routes of edges only become known during macro layout, this approach is not applicable to edge label placement.

The second approach, *post-processing*, refers to placing labels after all other diagram elements already have positions assigned to them. The advantage of post-processing is that it can be used with all layout algorithms, regardless of whether or not they support labels, since label placement is executed only after the algorithm has finished. The obvious disadvantage is that there may not be enough space to place labels according to the criteria

examined above. In particular, it may be impossible to place labels without overlaps with other elements. Some applications, such as *OpenStreetMap*[4] or *Google Maps*[5], allow users to zoom in and out while the size of labels stays approximately constant. In this context, filtering approaches can help by showing labels only at zoom levels where there is enough space for them, but finding appropriate ranges of zoom levels is a problem in itself [BDY06].

The third approach to edge label placement is what we shall call the *integrated approach to edge label placement*. In this approach, the layout algorithm not only places edge labels, but also makes sure there is enough space for them while it places nodes and routes edges. The advantage of this approach of course is that all edge labels can be placed according to the principles discussed above because the algorithm still has control over everything they might conflict with. The disadvantage is that diagrams become larger as more space is required to place their labels.

The latter point deserves being elaborated upon. In the introduction we have already seen that some visual languages suffer from a tendency towards long labels. To a certain extent, both the post-processing and the integrated approaches fail in such cases, each in its own way. Post-processing label placement algorithms will find that there is not enough space available to place a long label without overlaps, thereby violating all three of the most important placement principles. If they have the ability to filter labels, they may solve their problem by hiding those that could not be properly placed otherwise. Whether that qualifies as a valid solution depends on the application. Even if it does it needs to be noted that whether or not a label is hidden in this case depends entirely on layout properties and bears no relation to the label's semantical importance. In fact, the diagram may end up displaying short, but unimportant labels while hiding longer but essential ones.

The integrated approach ensures that there is enough space even for long labels to be placed, but may considerably enlarge the diagram in the process.There are ways, however, to improve this situation and we will come back to them when we discuss label management in Chapter 6.

[4] `https://www.openstreetmap.org`

[5] `https://maps.google.com`

During most of the remainder of this section, we will ponder ways of integrating the integrated approach into ELK Layered, from the distinction between different kinds of edge labels to the details of how to actually place them.

3.3.1 Edge Label Locations

There are different ways to think about where along an edge one might place a label. The most liberal way is to not make any demands on the exact location as long as clear graphic association is ensured. Whether this suffices depends on the particular application: in UML class diagrams, for instance, the number of instances of a particular class that take part in an association is denoted by *multiplicities*, placed right next to the association and near the class. Thus, where along an edge a label ends up may be important because placement may imply semantics.

Restricting this freedom a bit, we can understand the space around an edge as a potentially contiguous set of potential label locations. The *graphical feature label placement problem* defined by Kakoulis and Tollis [KT98] does just that and lets an application define a cost function that rates the locations according to preference. Doing so works well for post-processing algorithms where the most desirable label location may be unavailable due to insufficient space and where all potential label locations are known. For integrated algorithms, however, the specification of a cost function can be problematic, at the very least if the labeling space along an edge is contiguous, that is, if a label can be placed anywhere along the edge instead of being constrained to a discrete set of possible locations. Here, the placement of labels influences the length of edges, which is usually not known until the algorithm has finished. Thus, if the algorithm assumes that it places a label at the center of the edge, later phases may cause the label to end up a considerable distance away from the center and with that at a location which may have a much higher cost associated with it. Note that by the same reasoning it is also difficult at best for an integrated algorithm to place labels at a certain "percentage" of an edge's length, which might otherwise be a perfectly valid way to specify a desired edge label location.

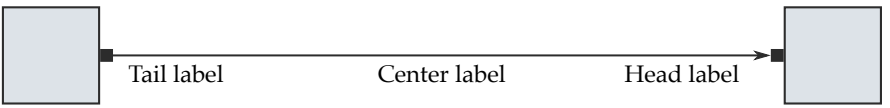


Figure 3.30. We distinguish between three possible edge label locations: one at each end of an edge and one in between.

Fortunately, given the kinds of diagrams we are interested in, this amount of flexibility seems like overkill anyway. The visual languages we have encountered so far are content with at most three possible label locations: *tail labels*, *center labels*, and *head labels*, as shown in Figure 3.30. Since we are dealing with an integrated edge label placement approach, we can guarantee that we will be able to place labels at their desired locations, thus eliminating the need to specify costs for locations "in between."

Before we go on to investigate how to place the different kinds of labels, it is worth noting that we can easily place several labels at any given location simply by stacking them on top of each other, with a configurable spacing separating each from the next. The labels thus stacked can then be treated as one.

3.3.2 Head and Tail Labels

Head and tail labels, or *end labels* when we do not care about the distinction, are placed close to an edge's target or source node, respectively. Attempting to place a label as close to a node as possible mandates for the two to be part of the same layer. If they were not, vertical edge segments could creep in between them, significantly reducing the perception of their relation and blurring the distinction between center and end labels. To keep things legible, we also need to make sure that end labels do not overlap any other node labels and that they are not crossed by vertical edge segments. To understand how this can be done, we need to revisit the node margins that we most recently touched upon in Section 3.2.6.

Placing head and tail labels of edges is done after each node's micro layout has already been computed. Its margins thus include all of its node labels as well as its ports and their labels. Placing end labels outside of the

margins ensures that we do not cause any overlaps with these elements. If we then enlarge the margins to encompass the end labels as well, we keep edges from being routed through that area and cross a label in the process. Note that this can easily cause the width of a layer to increase, causing the amount of white space and the width of the final diagram to increase as well. We will revisit this problem once we discuss center edge label placement.

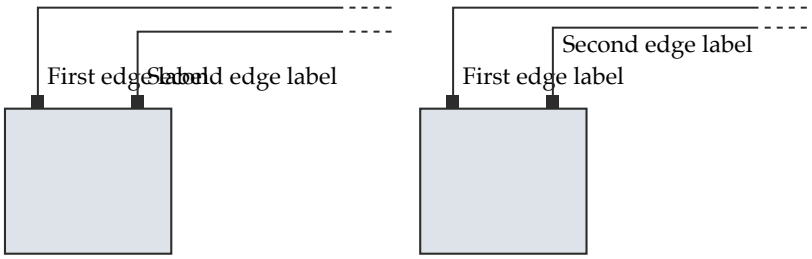
To place an end label, we have to decide which side of its edge it should be placed on. Section 3.3.4 will go to greater lengths to make the same decision for center labels, but for end labels we can refer back to Section 3.2.2 where we had a similar situation with placing port labels outside of their node. In fact, we use the same sides for end labels as would be computed for outside port labels to maximize the perception of which edge an end label belongs to.

For edges that connect to NORTH or SOUTH ports there is a final problem to be solved. Their labels will often be wider than the spacing between any two adjacent ports, giving the labels ample opportunity to overlap, as shown in Figure 3.31a. We have already seen this kind of problem: the same thing could happen with outside port labels on the NORTH and SOUTH sides (see Problem 3.17), and we accordingly use the same solution of moving the labels vertically until we have eradicated all overlaps (see Figure 3.31b).

3.3.3 Center Labels

Contrary to end labels, center edge labels do not need to be placed right next to one of the nodes their edge connects. In fact, doing so might run the risk of establishing a visual connection where none is intended. Center labels should thus not share a layer with any of these nodes but instead be placed in their own layer. The way to do so is to represent center labels as dummy nodes.

Before a layer assignment is computed, an intermediate processor iterates over the graph and looks for center edge labels. For each edge it finds that has at least one of them, it introduces a *label dummy node* that breaks the edge apart. It computes the size of the node such that all center labels fit into it, stacked upon each other with the correct spacings adhered to,



(a) With overlaps **(b)** Without overlaps

Figure 3.31. If the end labels of edges connected to northern or southern ports are too long, they are prone to overlapping. **(a)** Simply placing them without paying regard to their size produces overlaps. **(b)** Offsetting them vertically solves this problem.

plus the spacing to be left between the labels and their edge. At the end of the layout algorithm, once edge routing has finished, all label dummies are removed and the labels they represent are placed based on their dummy's coordinates.

Quite obviously, label dummies need to be inserted before the layer assignment step to ensure that each dummy is assigned to a layer (which might end up existing only because of the label dummy). It does place a burden on the layer assignment algorithm, though: if there is any qualitative difference in where exactly along an edge a label dummy is placed, it seems to be the layer assignment algorithm's responsibility to place it in the optimal spot, as if the layer assignment problem was not already hard enough as it is. Since we do not wish to make it even harder, we make no assumptions as to which layer a label dummy node get assigned to. Instead, we run another intermediate processor which moves label dummies into the layer it considers the best choice.

That choice is obvious if the edge is so short that there is only one layer to choose from. If the edge is longer, however, we need a strategy that defines what the best choice is. We define two basic kinds of strategies: *simple strategies* and *size-aware strategies*. Simple strategies base their decisions only on structural information, while size-aware strategies also take information

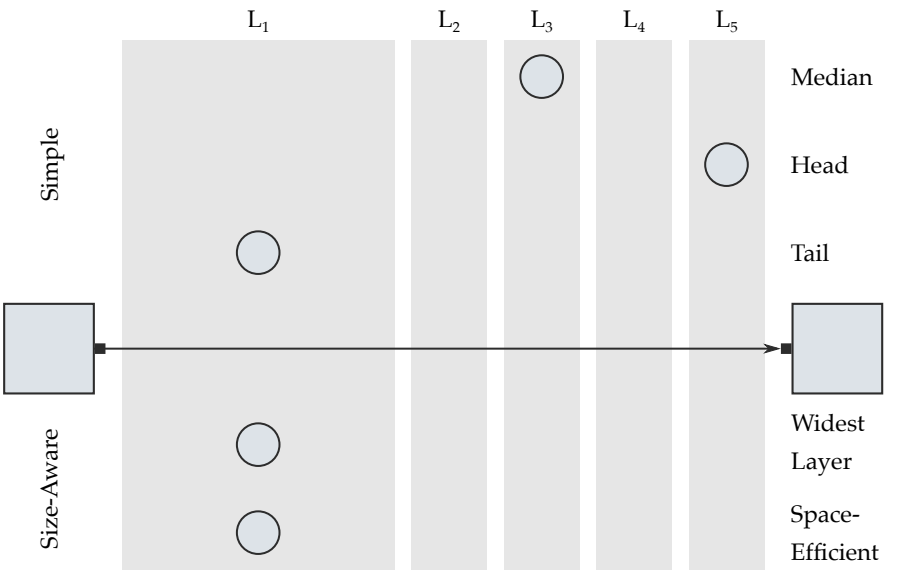


Figure 3.32. Different layer selection strategies would place a label dummy node in different layers spanned by a long edge, in this case layers L_1 through L_5. The *widest layer* and *space-efficient* strategies produce the same assignment in this example. What should be considered the best result depends on the design and the requirements of the visual language.

about node and label sizes into account. For the remainder of this section, we will look at examples of each in turn, all illustrated in Figure 3.32.

Simple Strategies

Possibly the simplest strategy imaginable (apart from refusing to do anything at all) is the *median strategy*, which places a label dummy in the median of all the layers its edge spans. If the width of these layers is approximately uniform, the median strategy will place labels approximately at the center of their edge, staying true to the literal meaning of the term "center edge

label." It can be argued, however, that this may not be the optimal place for all visual languages.

Let us again return to SCCharts to investigate this claim. As explained before, edges represent transitions from a source to a target state that are eligible to be taken based on some condition. That condition is part of the edge label, as is a hint regarding the order in which the conditions are tested if multiple transitions leave a state. If edge labels are placed near the center of their edge, a user might have to search a large area of their SCChart to review the labels of all outgoing transitions in order to understand how the SCChart works.

A second strategy, the *end layer strategy*, targets exactly this kind of situation. It places a label dummy in the layer closest to either the source or the target node of an edge, which begs the question of whether this blurs the distinction between center and end labels. Indeed, end labels could be used, but center labels provide two advantages in this context. First, remember that labels in SCCharts can grow very wide. While center labels can use all of the space their layer provides and share that space with other nodes and labels, end labels are placed in the node margins, excessively enlarging that layer. And second, if many transitions leave a state, it seems easier to place label dummy nodes properly than having to cram end labels around the source state and still keep them legible.

Note that both of these simple strategies may cause very wide label dummies to be placed in layers that otherwise contain only comparatively narrow nodes, thus enlarging them even if other layers may already be wide enough to house the dummy nodes.

Size-Aware Strategies

Such considerations lead to strategies that consider the size of nodes and label dummies, moving them to layers such that the size of the overall layout is minimized. While this seems particularly important for languages whose labels quickly grow wide, it has to be noted that it sacrifices any obvious rule regarding where along an edge to start looking for its label, which may or may not be acceptable. Whether it may even have an adverse

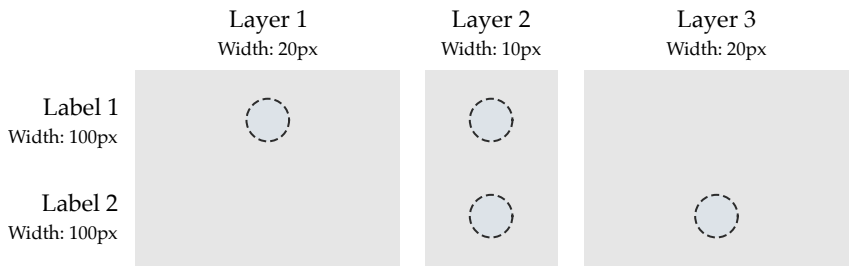


Figure 3.33. In this abstracted example, we have two labels that, due to the way their edges are connected in the original diagram, can be assigned to two out of three layers each (indicated by the dashed circles). The widest layer strategy will assign label 1 to layer 1 and label 2 to layer 3, for a total width of 210 pixels. The optimal solution is to assign both labels to layer 2, for a total width of 140 pixels.

effect on the ability of users to work with diagrams effectively may have to be studied further.

A first naive approach is the *widest layer strategy*, which places a label dummy in the widest of all layers spanned by its edge. For the purposes of this strategy, the width of a layer is defined as the width of its widest non-dummy node. Label dummy nodes in particular do not contribute to a layer's width here since it is not clear whether they are already in their final layer—if they are not, moving them to another layer may change layer sizes, possibly invalidating decisions made earlier.

It is for this exact reason that the approach must be considered too naive. Consider, for example, the situation shown in Figure 3.33. The widest layer strategy would assign each label to the widest layer available, without regard to where the other label would end up. Thus, since layer 2 is the smallest of the available layers, label 1 would be placed in layer 1 and label 2 would be placed in layer 3. The fundamental flaw here is that the widest layer strategy fails to take into account the effect of assigning labels to layer 2 has on that layer's width.

Intuitively it seems that taking the width of other labels into account as well will lead to better solutions, but will make the problem more difficult as well.

Problem 3.18 (MinWidthLabelAssignment). Given a set of layers and labels, their respective widths, and for each label the interval of layers it may be placed in. The width of an assignment is the sum of the width of each of its layers, in turn defined as the maximum of a layer's own width and that of any dummy node assigned to it. The MinWidthLabelAssignment problem seeks an assignment of each label to exactly one of its valid layers such that the assignment's width is minimized.

This problem is easily formulated as a mixed-integer linear program:

Inputs:

$$T = \{1, \ldots, m\} \qquad \text{(textual labels)}$$

$$L = \{1, \ldots, n\} \qquad \text{(layers)}$$

$$wt_T \in \mathbb{R}_{\geqslant 0} \qquad \text{(width of each label)}$$

$$wl_L \in \mathbb{R}_{\geqslant 0} \qquad \text{(width of each layer)}$$

$$V_t = \{lb_t, \ldots, rb_t\} \subseteq L \qquad \text{(interval of valid layers for } t \in T\text{)}$$

Output:

$$(a_{t,l}) \in \{0,1\}^{m \times n} \qquad \text{(assignment of labels to layers)}$$

Minimize:

$$\sum_{l \in L} \max \left\{ wl_l, \max_{t \in T} \left\{ wt_t \cdot a_{t,l} \right\} \right\} \qquad \text{(total width)}$$

For all $t \in T$ subject to:

$$\sum_{l \in L} a_{t,l} = 1 \qquad \text{(assign to exactly one layer)}$$

$$\sum_{l \in V_t} a_{t,l} = 1 \qquad \text{(assign to valid layer)}$$

□

Note that this formulation of the problem is deliberately close to our application in that it explicitly models the width of the layers. It could

be stated more simply if instead of modeling layers explicitly we would add a dummy label for each layer with that layer's width that can only be assigned to that layer.

If all labels and layers have the same unit width, the problem is equivalent to the *hitting interval problem*, where, in our terminology, we seek the minimum number of layers such that each label is "hit" at least once [KBD+17]. This problem can be solved in polynomial time by looking at it through the lense of an *interval graph*: the graph's nodes are the labels, and two nodes are connected by an edge if they share a layer they can both be assigned to. All labels that can be assigned to a given layer form a clique in the interval graph, so finding the minimum number of cliques gives us the minimum number of layers for all labels to be assigned. While usually an NP-hard optimization problem, such a minimum clique cover can be computed in polynomial time on interval graphs [Gol04].

The decision version of the MinWidthLabelAssignment problem asks whether there exists an assignment that does not exceed a given total width. This problem is obviously in NP, since given an assignment, we can compute its width in polynomial time. It is is not at all clear, however, whether it is also NP-hard. It bears resemblance to the *set cover problem*, which, given a finite universe U of elements and a family of subsets of that universe, asks whether it is possible to find up to a given number of subsets whose union yields the universe. One might try reducing from set cover to MinWidthLabelAssignment: the universe would be the set of labels and the subsets would be the layers, each represented by the set of labels that can be assigned to it. This reduction, however, is a doomed enterprise due to the fact that MinWidthLabelAssignment requires labels to be assignable only to intervals of layers instead of to arbitrary subsets.

A similar problem arises when trying to reduce from the *restricted assignment problem*, which assigns jobs with run times to machines subject to constraints as to which job may be assigned to which machines. Again, the intervals pose a problem, as does the fact that MinWidthLabelAssignment only takes the maximum width of nodes into account while restricted assignment optimizes based on the total run time of the jobs assigned to each machine. In conclusion, while MinWidthLabelAssignment feels like an NP-complete problem, it is not yet clear whether it is in fact one.

Open Problem 3.19. Is MinWidthLabelAssignment NP-complete? □

The *space-efficient layer strategy* is a heuristic that aims to approximate an optimal solution to the MinWidthLabelAssignment problem. It starts by assigning all labels that have a layer large enough to house them or that have only a single layer to choose from. What is left are labels that can be assigned to multiple layers, all of which too small. Those labels are processed in descending order of their width. For each label, the heuristic calculates the *potential width* of each layer it can be assigned to, defined as the maximum of the layer's current width and the width of all labels that might still be assigned to it (except for the label that is currently being processed). The label is assigned to the layer that has the largest potential width.

This being a greedy strategy it seems likely that there will be cases in which it fails to find the optimum. Sure enough, Figure 3.34 shows a case in which that happens. Label 3 is processed first since it can only be assigned to layer 3. Label 1 is processed next. Without label 2 it would be assigned to layer 3, but label 2 causes it to be diverted to layer 2. Label 2 itself, then, is also assigned to layer 2 because of label 1. The optimal assignment, however, would assign label 1 to layer 3 and label 2 to layer 1. The problem is caused by the fact that when the algorithm processes label 1, the potential size of layer 2 is calculated without regard to what label 2 should be assigned to.

If the goal is not to reduce the width of a diagram, but to place labels at the center of their edge, it would seem that this calls for another size-aware strategy. As we will see in Section 3.4.2, however, the median strategy already produces results that are good enough.

The implementation of size-aware strategies in ELK Layered offers a number of surprises. A naive approach would place the intermediate processor that implements them between the layer assignment and crossing minimization phases. What would work perfectly well for simple strategies poses a problem for the size-aware ones: at that point in the algorithm, only the real size of each non-dummy node is known. For the virtual size to become available, which will actually determine the width of its layer, we have to wait for micro layout to be run, which is not until after crossing minimization. This seems problematic, since in general nodes cannot be

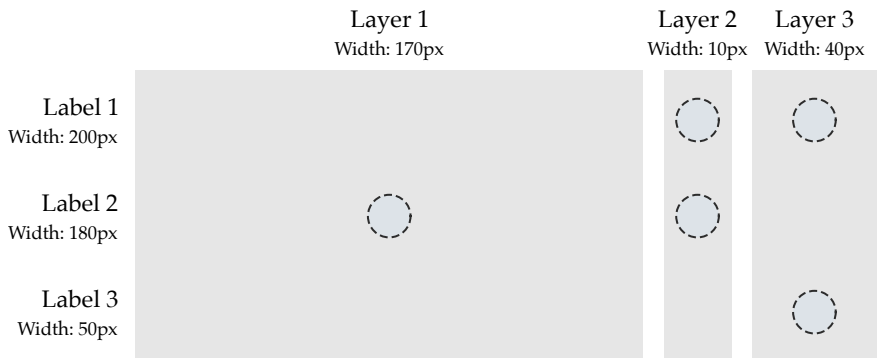


Figure 3.34. An example where the space-efficient heuristic produces a non-optimal assignment with a width of 420 pixels (label 1 → layer 2, label 2 → layer 2, label 3 → layer 3). The optimal assignment is 390 pixels wide (label 1 → layer 3, label 2 → layer 1, label 3 → layer 3).

moved to other layers anymore after the order of nodes in each layer has been determined. In this case, however, all the layer selection strategies do is to swap label dummy nodes with long edge dummy nodes. Since both have exactly one incoming and one outgoing edge, we can safely interchange them without changing the number of crossings. It is thus safe to delay label layer selection until after the virtual size of all nodes is known.

3.3.4 Label Side Selection

The subject of our discussions in the previous section was where to place center edge labels along the horizontal axis. In this section, we will be concerned with their placement along the vertical axis. A center edge label can be placed above, below, or even on the edge it belongs to. Implementation-wise, this decision influences where we place the ports of the label's dummy node, as Figure 3.35 shows—but how do we make this decision? Let us work our way through different strategies and examine their advantages and disadvantages.

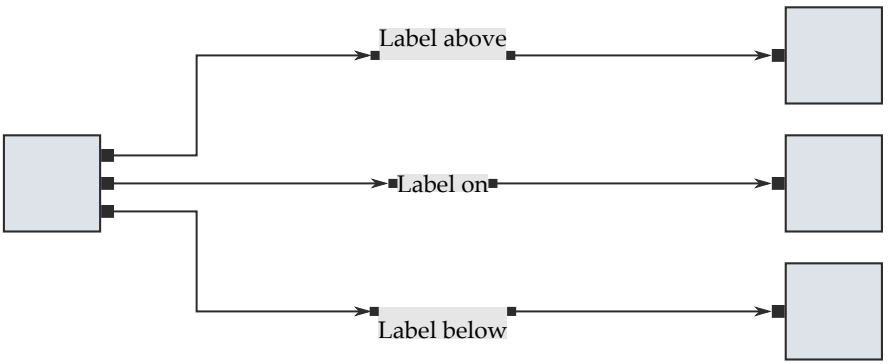


Figure 3.35. Whether an edge label will end up above, below, or on its edge is a function of where its dummy node's ports are placed. Note that the size of the dummy nodes includes the space to be left between the label and its edge, except in the on-edge case.

End Label Strategy

We have already discussed that deriving label sides for end labels can easily be done the same way that we derive label sides for port labels. A first strategy thus might be to simply use these label sides for center labels as well, perhaps in an attempt to keep all the labels of a given edge on the same side. This strategy has two problems, however.

First, as a diagram gets larger, chances are that more and more center labels move a considerable distance away from the end labels (depending on the layer selection strategy, of course). It is not clear why the placement of an end label should have any influence on the placement of a center label which may not even be in the end label's proximity.

More importantly, however, it is easy to construct examples where the end labels are placed on different sides, as in Figure 3.36. It thus seems that we are forced to continue our quest for good label side selection strategies.

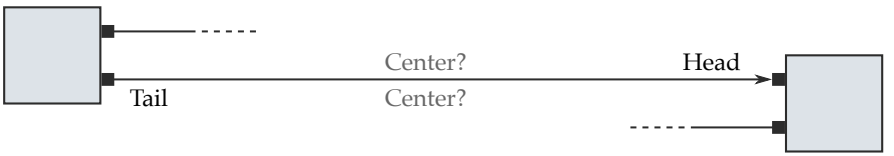


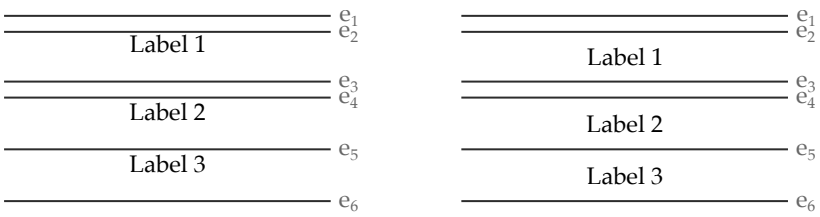
Figure 3.36. While the tail label is placed below its edge, the head label is placed above. If the end label side was to be used for the center label as well, it would not be clear which side to choose.

Same-Side Strategy

An obvious strategy is the *same-side strategy*, which places all labels either above or below their edge. The simplest strategy to implement, it may also be the easiest for users to understand due to its consistency, which can make it work even when other aspects of a layout do their best to sabotage it.

Consider the example in Figure 3.37a. It is clear which edge each label belongs to due to the fact that it is placed nearer to its edge than to any other. Donald Norman would call this "knowledge in the world" [Nor88] in that the diagram can stand on its own and does not require further information to be deciphered. Note that this is not a statement about how easy this kind of placement is for users, only about whether it is possible for them to make sense of it at all. Now consider Figure 3.37b. Here, the associations between labels and edges are ambiguous due to unfortunate spacings. Knowing that labels are always placed below their edge resolves any ambiguity and makes the same-side strategy work even in such circumstances. Norman, however, claims that such additional information required to understand the world—what he calls "knowledge in the head"—should be avoided whenever possible. Although the same-side strategy works even with unfortunate spacings, the preferred way is to support it with properly chosen spacings to make graphic association as clear as possible. Still, a convention regarding the side labels are always placed at may be part of a visual language.

It might seem odd that only some of the edges in the example are labeled. However, this is a common situation since edges usually span different sets of layers and thus have their labels placed in different layers.



(a) With clear spacings **(b)** With ambiguous spacings

Figure 3.37. Placing all labels above or below their edge yields maximum consistency. **(a)** With proper spacings between labels and related or unrelated edges, a label placement is unambiguous regardless of the label side selection strategy. **(b)** With unfortunate spacings, the same-side strategy still yields unambiguous results if users are aware that this strategy is used.

Directional Strategy

While the same-side strategy works well in terms of clear graphic association, it does not encode additional information, such as the direction an edge is heading towards. Since a label may be far removed from the end points of its edge, any clue as to the edge's direction may help a user navigate the diagram. The *directional strategy* aims to do just that by always placing a label to the left or to the right of an edge (looking towards its head).

Figure 3.38 shows an example of this strategy in action. Knowing that labels are always placed to the left of an edge lets us deduce that e_2 is headed rightwards while e_4 and e_5 are going off to the left. If spacings are chosen well, this additional piece of knowledge is not required for clear graphic association, but offers additional information to advanced users of a visual language who know about the convention.

If spacings are chosen badly, the directional strategy ceases to work. Referring back to Figure 3.37b, it would for example not be clear whether *Label 1* belongs to e_2 (which would then be headed leftwards) or to e_3 (which would then go to the right).

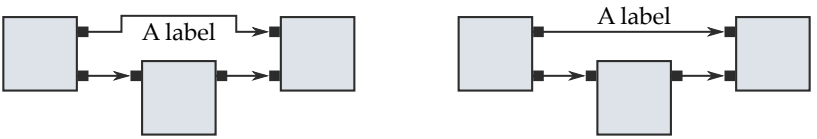
Figure 3.38. Directional label side selection lets users deduce where a labeled edge is heading without having to look for its end points. This is only true, of course, if users know about how edge directions are encoded in the placement of their labels.

Of course, this strategy requires knowledge in the head for deriving edge directions. We will pick up on this problem later on in Section 3.3.5 when we discuss other ways of indicating edge direction.

Augmented Same-Side Strategy

Arguably one of the things most frustrating to users is hitting the layout button and being served a result with deficiencies that they immediately see how to resolve. Schelten calls graphs that contain such abominations Obviously Non-optimal (O-No) [Sch16a], an abbreviation which seems particularly appropriate. Here, we use the term not for complete graphs, but for the deficiencies themselves which, in accordance with the *Tao of Programming*'s "Law of Least Astonishment" [Jam86, §4.1], we seek to avoid. While the same-side and directional strategies already seem like decent methods for selecting label sides (particularly with sensible spacings), in practice they can produce O-Nos, ranging in severity anywhere from "not too bad" to "I'll do it myself next time!"

Based on observing examples of O-Nos, I will derive a set of rules with which to augment the same-side strategy to arrive at what we unimaginatively call the *augmented same-side strategy*. All labels not matched by one of the rules will be assigned a default label side. The goals are both clear graphic association and improving drawings with regard to aesthetic criteria, but when in doubt we will choose to be conservative in that clear graphic association will never be sacrificed for improved aesthetics.



(a) Without Rule 1 **(b)** With Rule 1

Figure 3.39. Simple label side selection methods can easily produce edges that are longer and have more bend points than necessary. **(a)** The label causes its edge to be routed around it. **(b)** Changing the label sides reduces the edge's length and the number of its bend points.

Before we start examining examples, however, we should take a look at the general algorithm that implements the augmented same-side strategy. Algorithm 3.1 is executed for every layer in the graph. It iterates over the layer's nodes from top to bottom and looks for consecutive runs of long-edge and label dummy nodes (called `dummyGroup` in the code). Once one such run ends because another type of node is found or because the layer ends, it is processed by calling `process()` in lines 13 and 19. The definition of that function is deviously missing in the pseudo code because it is this function which implements the set of rules we are about to discuss. The algorithm also does a bit of bookkeeping to keep track of the number of actual label dummy nodes in the current run as well as of whether the current run contains the layer's topmost or bottommost node. This information will be used in our rules, so let us start looking at the examples from which we will derive them.

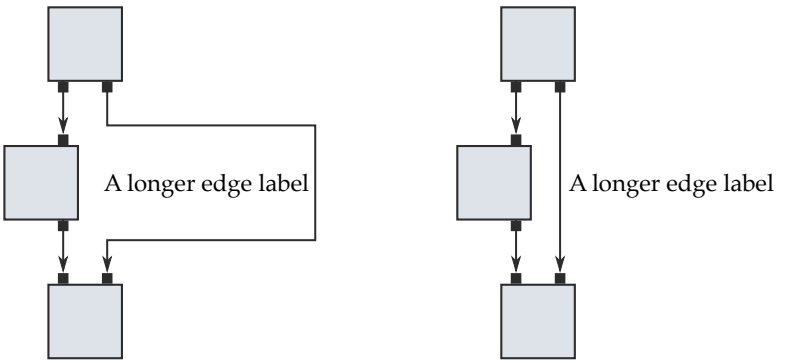
The first rule concerns the length of edges and, ideally, the number of bend points. Consider the example in Figure 3.39a. What we have here is a label of a layer's topmost edge and a label side selection algorithm which thought it a good idea to place the label below the edge. This of course causes the edge to have to make a detour around the label, increasing both its length and the number of its bend points. Simply placing the label above the edge, as in Figure 3.39b, improves edge routing while retaining graphic association, thus leading to our first rule.

```
Input: layer, a list of nodes in the layer to be processed
Output: Assignment of label side to each label dummy in layer
1  dummyGroup ← empty list
2  labelCount ← 0
3  topInLayer ← true
4  bottomInLayer ← false
5  foreach node in layer do
6      if isLabelDummy(node) then
7          Add node to dummyGroup
8          labelCount += 1
9      else if isLongEdgeDummy(node) then
10         Add node to dummyGroup
11     else
12         if labelCount > 0 then
13             process(dummyGroup, topInLayer, bottomInLayer)
14             dummyGroup ← empty list
15             labelCount ← 0
16         topInLayer ← false
17 if labelCount > 0 then
18     bottomInLayer ← true
19     process(dummyGroup, topInLayer, bottomInLayer)
```

Algorithm 3.1. The *augmented same-side strategy* is run for every layer of a graph. It basically looks for consecutive runs of long-edge and label dummy nodes and processes them.

> **Rule 1**
> If `labelCount` $= 1$ and either `topGroup` is true and the first node in `dummyGroup` is a label dummy or `bottomGroup` is true and the last node in `dummyGroup` is a label dummy, configure the corresponding label to be above or below its edge, respectively.

Note that we are being conservative here: if more than the topmost edge has a label in the layer, we do not apply this rule in order to avoid any confusion (one of the subsequent rules may still apply, though).



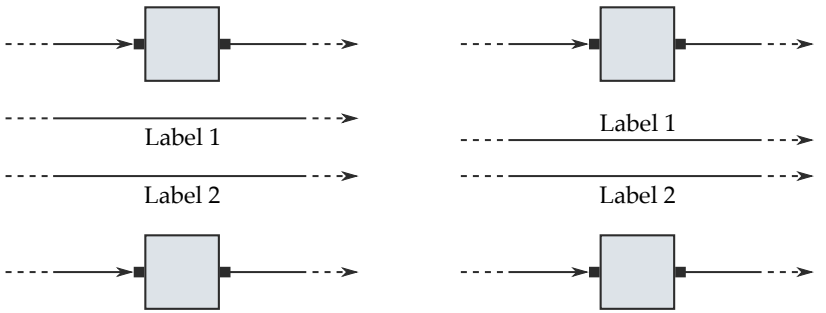
(a) Without Rule 1 **(b)** With Rule 1

Figure 3.40. A graph similar to the one in Figure 3.39, but with a vertical layout direction. **(a)** Without applying Rule 1, the problem becomes even worse since the label's width has more of an impact. **(b)** Applying Rule 1 again improves the result.

The usefulness of Rule 1 becomes more apparent for vertical layouts, as in Figure 3.40. Here, the width of the label has much more of an impact on the layout than its height did before.

The second example concerns cases where a run of dummy nodes consists of exactly two nodes, as in Figure 3.41. The same-side strategy will place one of the labels between the edges, while the directional strategy may even end up placing both labels there. While this may already be clear enough, especially if the label-side selection strategy is known and spacings are chosen sensibly, we can still improve graphic association by placing none of the labels between the edges. To repeat a point made during our discussion of port labels in Section 3.2.2, placing the labels around the edges ensures that there is not even a slight possibility of mistaking the association of one of the labels. This leads us to our second rule.

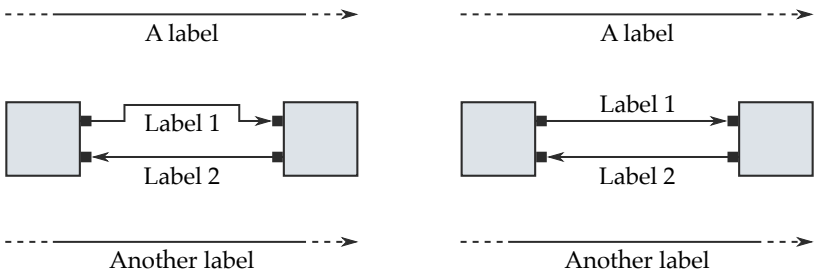
> **Rule 2**
> If $|\texttt{dummyGroup}| = 2$, configure the group's first and second dummy node for above and below placement, respectively, if it is a label dummy node.



(a) Without Rule 2 **(b)** With Rule 2

Figure 3.41. Example of a diagram where only two edges run between two regular nodes. **(a)** The same-side strategy will put one of the two labels between the edges. The directional strategy may even end up putting both labels there. **(b)** Placing the labels around the edges improves graphic association.

The final example highlights a construct that frequently appears in SCCharts: two states connected by two transitions, one in each direction, as shown in Figure 3.42. We call this a *tight loop*, although being a loop is not really a requirement—one can easily imagine situations in data-flow languages where one actor sends two signals to the next actor instead of just one, effectively producing the same situation. This of course causes the same problems we already discussed for the previous rule, but worse. If two nodes are connected by two edges, those edges will usually be spaced rather tightly if they are not labeled. Introducing a label into the space between them may not only introduce the possibility of ambiguity, but also requires them to be moved apart, introducing bend points and elongating them (Figure 3.42a). Placing the labels around the edges improves the situation (Figure 3.42b), leading to the final rule.



(a) Without Rule 3 **(b)** With Rule 3

Figure 3.42. Edges of a tight loop tend to be close to one another, which causes longer edges and bend points if labels are placed between them. This is a generalization of Rule 2 to tight loops, which are possibly surrounded by other edges instead of regular nodes.

> **Rule 3**
> When encountering exactly two dummy nodes that belong to edges that connect the same two nodes, configure the upper one's label to be placed above and the lower one's label to be placed below the edge.

This rule seems contradict our basic goal of being conservative in our label side decisions. After all, there may be more label dummy nodes in the current run, which would so far cause us to fall back to the default label side for all of them. However, in our experience such tight loops are often placed at a bit of a distance from surrounding elements, thus preserving graphic association. Of course, the optimum would be to take distances to other edges into account. That, however, is not possible because label side selection needs to happen before nodes are placed and edges are routed.

On-Edge Strategy

We have thus far focussed on what might be called *traditional* or *next-to-edge* label placement: the assumption that an edge label needs to be placed next to its edge, which the vast majority of graphical modeling tools follow. This

makes perfect sense in that the principles of legibility and minimization of disturbances seem to call for labels to not overlap their edges. There is a case to be made, however, for placing a label *on* its edge.

One of our main concerns over the past few sections was to achieve clear graphic association between labels and the edges they label. This had to be a concern of ours because labels were placed next to their respective edge: the perception of their association improved as they got closer to each other and further away from unrelated elements, an example of the principle of *proximity* in what Gestalt psychology calls *perceptual grouping* [WEK+12]. Graphic designers will use the same principle to group elements or distinguish them from other elements. Given properly configured spacings, this is not too hard to achieve, but will cause a drawing to grow in size. As spacings shrink, drawings do get smaller, but the danger of ambiguous graphic association grows.

Wong et al. [WMP+05] eliminate the need for perceptual grouping by going so far as to replace the edge itself with its label, gradually changing the font size from tail to head to indicate edge direction. We will not follow their proposal, due to several reasons. First, for the approach to work without repeating a label's text or introducing either distortion or vastly different font sizes, the length of an edge would have to be a function of the text it is labeled with—a prerequisite quite obviously not compatible with the layered approach. Second, we allow edges that share a common end point to be drawn as hyperedges, which essentially lets them share parts of the routes they take through the diagram. This would have a decidedly negative impact on the legibility of labels. And finally, the orthogonal edge routing style (or any routing style that employs bend points, for that matter) would not exactly improve label legibility, either.

Figure 3.43 shows an example of what on-edge label placement can look like. Here, the ports of a label dummy node would be placed at the center of their port side, causing the edge to go right through the label in the final drawing. Also, the spacing usually left between the label and its edge would be omitted when computing the label dummy's height.

For on-edge label placement to work, the drawing framework has to be aware of it. A naive implementation might end up drawing an edge after having drawn its label, which of course would not work (unless crossed-

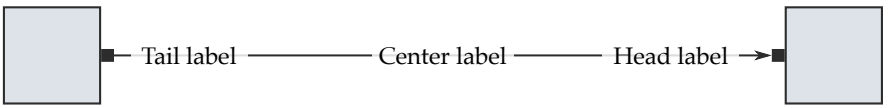


Figure 3.43. Instead of placing edge labels next to the edges they label, they can also be placed *on* their edges. This only works if the label is drawn such that it hides the edge underneath, or at least "dims" it significantly.

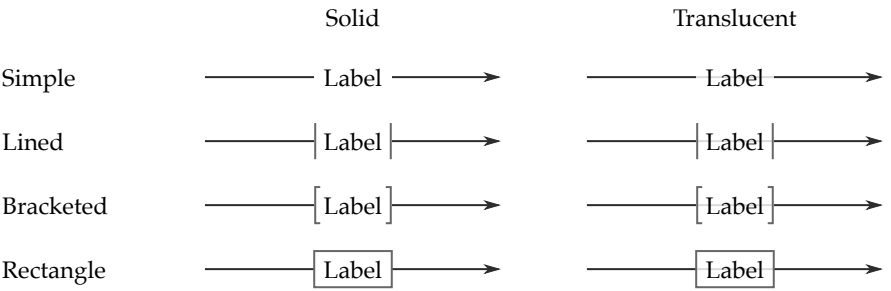


Figure 3.44. Four examples of decorating labels for them to be placed on their respective edge.

through labels were the intended goal, that is). But even if edges are always drawn before their labels, the graphical representation of the latter has to be designed with on-edge label placement in mind. Labels must have either a solid background or at least cause the background to be sufficiently faded for the edge not to interfere with the text. This requirement is easy to meet and many designs for on-edge labels are possible, which may even be used to reflect different edge semantics. Figure 3.44 shows four simple examples of on-edge label representations, each with both a solid and a slightly translucent background. Castelló et al. [CMT01] indeed use a simple solid design when drawing statecharts, but do not discuss their motivation for doing so.

What we have gained with on-edge label placement is optimal graphic association between edges and their labels. If the layout direction is horizontal, we have also reduced the diagram's height slightly because spacings are not necessary anymore, and the space between labels and unrelated

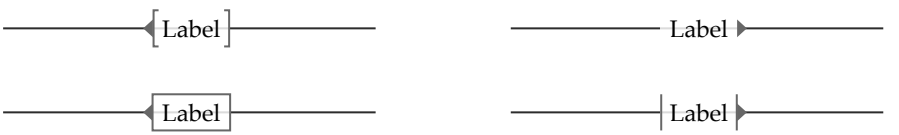


Figure 3.45. Labels can be decorated with arrows to point towards where the edge is heading.

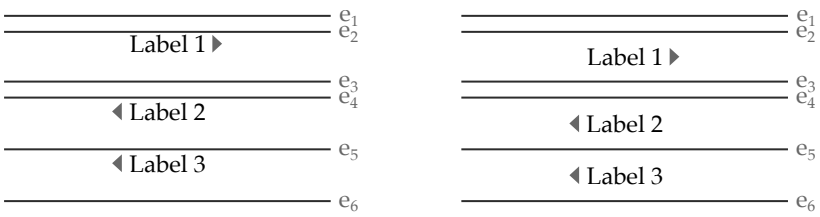
edges can be much smaller than it could be otherwise. On the negative side, it may be the case that interrupting the line that represents an edge may cause users to have a harder time following the edge through the diagram. This hypothesis, however, needs to be verified, and we will indeed come back to this point in Section 3.4.3.

3.3.5 Directional Decorators

The directional label side selection strategy had the advantage of giving users information about the direction of an edge without them having to find and look at the edge's end points. However, it suffered from two problems: ambiguous graphic association unless spacings are chosen well, and the requirement for knowledge in the head for its proper interpretation. We will close the section on edge label placement by looking at another way of providing directional information that is compatible with all label placement strategies.

If the label side is unavailable as a means to communicate additional information, what remains is the possibility to communicate through the label's design. Figure 3.45 shows examples of on-edge labels decorated with an arrow which points towards the edge's head.

Compared to the directional strategy, this way of indicating edge direction has the advantage of representing knowledge in the world, not in the head. It does not require knowledge about any label side conventions that would otherwise have to be learned. Such decorations consequently work with any label side selection strategy, thus allowing the same-side strategy to communicate the same amount of information as the directional strategy while being slightly clearer in terms of graphic association (Figure 3.46).



(a) With clear spacings **(b)** With ambiguous spacings

Figure 3.46. The same edge label placement as in Figure 3.37, but with directional decorators added to the labels. **(a)** With arrows, the same-side label selection strategy can clearly communicate edge direction. **(b)** With badly chosen spacings, the directional strategy would cease to work due to ambiguous graphic association. The same-side strategy still works (provided that the user knows about the strategy being used) and can communicate the same amount of information when augmented with directional decorators.

An interesting problem concerns the implementation of directional decorators. Which direction the arrow should point to is subject to the diagram's layout, which implies that the viewing framework needs to support changes to the visualization after automatic layout has run. KLighD provides *style modifiers* to do just that, resulting in the following sequence of steps to implement direction decorators for KLighD-based visualizations:

1. While generating the view model, add all four possible arrows to each label's rendering.
2. Execute automatic layout.
3. Using a style modifier, set each arrow's visibility to `true` or `false` subject to the edge's direction as computed by the layout algorithm.

3.4 Evaluation

The micro layout algorithm described in Section 3.2 was developed to address the **P-PLACEMENT** principle for node and port labels. Our main

goal was to cover those use cases that we regularly see in applications by providing a certain amount of flexibility through layout options. Other than to report that we have since successfully used it for a number of different visual languages it seems hard to evaluate it quantitatively. The situation is different for the edge label placement strategies discussed in Section 3.3, which is what the rest of this section is about.

Our approach to placing center edge labels in Section 3.3 consisted of strategies to solve two main problems: which layer to place a label in and which side of its edge to place it on. What remains is to compare the strategies. We begin with two evaluations that pit them against each other based purely on a selection of aesthetics criteria. If a strategy fares well there, however, that does not necessarily mean that its supposed effectiveness will survive first contact with users. This is why we complement the quantitative evaluations with both a user study and a survey among students using SCCharts.

3.4.1 Layout Impact of Side Selection Strategies

The choice of a side selection strategy will influence different aesthetics of the drawing. The aim of the first evaluation is to get an idea of the extent of that influence. I wanted to answer the following questions:

1. Does the on-edge strategy result in smaller drawings?

   It requires less edge-edge spacing due to the obvious graphic association between labels and the edges they belong to. This suggests an influence on the height of horizontal and on the width of vertical drawings. I did not expect the differences to be very large, though. Note that this question directly relates to the **P-SIZE** principle.

2. Does the augmented side selection strategy yield shorter edges?

   It does attempt to reduce obvious O-Nos where a label's placement causes its edge to take a detour, leading to the hypothesis that it would indeed produce shorter edges than its simple counterparts, although again only slightly. However, since labels are wider than they are high, the effect would be more pronounced in vertical layouts.

3. Does the augmented side selection strategy increase the number of straight edges?

   The way the augmented strategy chooses label sides suggests that it might, but again the increase must be expected to be small.

To help in the quest for answers, I obtained 315 SCCharts produced by students. Some were created as part of their homework assignments in courses on synchronous languages and real-time systems that used SCCharts as an example of a visual synchronous language. Others were derived from a large SCChart that controls an elaborate model railway, produced by students during a recent practical. In all cases the students used a textual language to describe the SCChart which was then rendered as a diagram. Since SCCharts usually consist of several levels of hierarchy that are laid out by different layout algorithms, I extracted simple graphs (only a single level of hierarchy) and removed diagrams that had no edge labels or less than three nodes. I thus ended up with 641 diagrams averaging 6.47 nodes (for a total of 4,147 nodes), 9.37 edges (6,005 total), and 8.93 edge labels (5,726 total). Section A.1 provides several samples of the diagrams used for this experiment.

Before being analyzed the diagrams were laid out by ELK Layered. Each diagram was laid out twice for each side selection strategy, once with a horizontal layout direction (left-to-right) and once with a vertical layout direction (top-to-bottom). We used the same settings throughout, except for the on-edge strategy where we reduced the edge-edge spacing from 10 to 5 pixels due to clearer graphic association.

First Question The first question was about the impact of the on-edge side selection strategy on a horizontal drawing's height and a vertical drawing's width. Table 3.2 shows the mean and the standard deviation of the height and width of our drawings subject to the chosen label side selection strategy. The standard deviation indicates that the size of the drawings varies widely, but the mean size is smallest for the on-edge strategy.

If we shift our focus away from absolutes towards relatives, as in Table 3.3, the results get much more obvious. Here the size of each drawing made using the on-edge strategy is expressed in terms of the size produced

Table 3.2. Height of horizontal drawings and width of vertical drawings (in pixels) subject to the selected label side selection algorithm. Height and width are described in terms of the Mean (MN) and the Standard Deviation (SD).

	Height		Width	
	MN	SD	MN	SD
Down	168.56	208.16	1,739.86	3,160.52
Up	169.57	207.20	1,748.76	3,274.06
Dir. Down	170.43	208.02	1,712.57	3,271.44
Dir. Up	170.63	207.65	1,732.86	3,127.06
Aug. Down	169.15	207.04	1,730.64	3,134.30
Aug. Up	169.12	206.70	1,746.82	3,301.33
On Edge	147.53	175.02	1,593.43	3,077.38

by the other strategies and is, on average, consistently smaller. A look at a plot of the relative data in Figure 3.47 however reveals that this is not always the case. In fact, the different strategies will yield slightly different node placements, which in turn can lead to bigger diagrams. Due to the fact that labels are wider than they are high, this effect is more pronounced in vertical layouts. As Table 3.4 shows, however, the number of cases where this happened in our set of diagrams is very small.

In summary we can conclude that using the on-edge label side selection strategy will yield smaller diagrams than the other strategies in the vast majority of cases, thus rendering it a contribution that conforms to the **P-SIZE** principle.

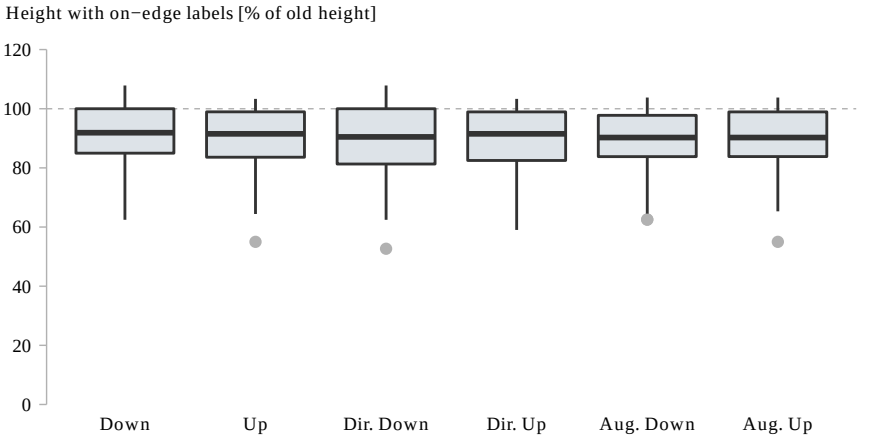
Second Question The second question was concerned with whether the augmented side selection strategies resulted in shorter edges compared to their simple counterparts. Table 3.5 shows the differences in edge length when switching from the always down and always up strategies to the augmented down and augmented up strategies, respectively. While the differences are negligible in horizontal layouts, they are indeed much more

Table 3.3. Height of horizontal drawings and width of vertical drawings (in pixels) obtained using the on-edge side selection strategy expressed as a fraction of the size produced by the other strategies.

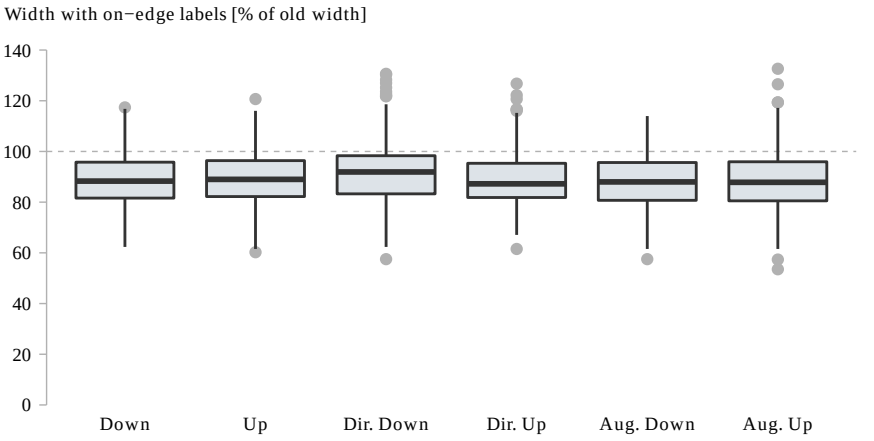
	Relative Height		Relative Width	
	MN	SD	MN	SD
Down	0.907	0.091	0.883	0.093
Up	0.897	0.093	0.887	0.093
Dir. Down	0.891	0.099	0.915	0.113
Dir. Up	0.892	0.097	0.880	0.096
Aug. Down	0.894	0.089	0.876	0.096
Aug. Up	0.896	0.090	0.880	0.104

Table 3.4. The percentage of cases where switching from a given side selection strategy to the on-edge strategy led to smaller, unchanged, or larger drawings (regarding height in horizontal and width in vertical layouts). All values are given in percent of all 641 analyzed drawings.

Strategy	Height			Width		
	Smaller	Unchanged	Larger	Smaller	Unchanged	Larger
Down	73.5	25.0	1.6	92.5	2.3	5.1
Up	75.8	23.4	0.8	92.4	2.5	5.1
Dir. Down	74.4	24.8	0.8	80.7	2.3	17.0
Dir. Up	76.3	23.4	0.3	92.4	2.3	5.3
Aug. Down	77.7	21.5	0.8	92.5	2.2	5.3
Aug. Up	76.3	22.8	0.9	92.4	2.3	5.3



(a) Horizontal drawings



(b) Vertical drawings

Figure 3.47. Box plots of the data underlying Table 3.3. Lower values are better.

Table 3.5. Change in edge lengths (in pixels) when switching from a simple side selection strategy to its augmented counterpart. Negative change means shorter edges in the augmented case. Our data points for the *mean length* column are the mean lengths of all edges in a diagram.

Direction	Strategy	Average Length		Maximum Length	
		MN	SD	MN	SD
Horizontal	Down	−4.73	13.47	−14.31	82.64
	Up	−5.06	12.05	−9.51	52.56
Vertical	Down	−75.05	157.74	−221.89	549.19
	Up	−76.30	124.14	−228.32	461.88

pronounced in vertical layouts due to the fact that labels can be rather long, forcing considerable detours onto their edges. However, the standard deviations are rather large as well.

Table 3.6 shows the percentages of drawings where switching to an augmented strategy made edges shorter, left them unchanged, or made them longer (both the mean length of edges in a drawing as well as its longest edge). Here the pattern becomes clearer: switching to an augmented strategy improves mean edge length in about 60% of vertical drawings. More importantly, it turns out to be a bad idea only in less than about 10% of our examples.

I thus feel it is safe to say that the augmented strategies will indeed often lead to shorter edges, although the impact is much larger for vertical than for horizontal layouts.

Third Question Encouraged by the answer to the second question, it is time to answer the question of whether the augmented strategies reduce the number of bend points as well. As it turns out we can answer this question rather easily if we look at the sum of bend points across all drawings by label side selection algorithm, as shown in Table 3.7. The different strategies produce very similar numbers of bend points, including the augmented strategies, which do not even constitute the lower end of the spectrum.

Table 3.6. The percentage of cases where switching from a simple side selection strategy to its corresponding augmented side selection strategy led to shorter, unchanged, or longer mean and maximum edge lengths in a drawing. All values are given in percent of all 641 analyzed drawings.

Direction	Strategy	Mean Length			Maximum Length		
		Shorter	Unchanged	Longer	Shorter	Unchanged	Longer
Horizontal	Down	47.3	37.1	15.6	45.9	43.4	10.8
	Up	54.4	38.4	7.2	41.0	48.5	10.5
Vertical	Down	61.0	31.7	7.3	53.4	35.7	10.9
	Up	59.3	34.9	5.8	53.2	37.6	9.2

Even with the sums staying approximately the same, that does not mean, of course, that this is true for each diagram as well. However, a closer look at each diagram confirmed that indeed the number of bend points did not change much there either. We must thus answer this final question with no, the augmented strategies do not reduce the number of bend points significantly.

3.4.2 Layout Impact of Layer Selection Strategies

The impact of choosing a layer selection strategy on the aesthetics of the resulting layouts may not be the first criterion to base that choice on. Choosing the head or tail layer strategies, for example, can simply be based on the usability requirements of a given visual language. SCCharts are a good case in point: it is probably more helpful to display transition conditions near the source state than to worry about what that may do to the aspect ratio of the resulting drawing. Still, what an aesthetic evaluation can give us is an understanding of the effectiveness of the size-aware layer selection strategies. It is with this in mind that we want to answer the following questions:

1. How successful are the size-aware strategies in reducing the width of drawings? The answer to this question relates to the **P-SIZE** principle.

Table 3.7. The number of bend points generated across all analyzed drawings by label side selection algorithm

Strategy	Bend Points	
	Horizontal	Vertical
Down	2,014	1,900
Up	1,967	2,065
Dir. Down	1,989	2,031
Dir. Up	2,060	2,060
Aug. Down	2,055	2,013
Aug. Up	2,041	2,076
On Edge	2,064	1,989

2. How near to an edge's physical center does the median strategy place labels?

I based this evaluation on the same diagrams as the previous one, but removed diagrams where there was no choice regarding which layer to place labels in since the layer selection strategies would not make a difference there. I thus ended up with 366 diagrams totaling 2,758 nodes (with an average of 7.54), 4,525 edges (12.36), and 4,357 edge labels (11.9).

First Question To answer the first question, we laid out each diagram with all layer selection strategies plus with an implementation of the optimal strategy that uses IBM's CPLEX software to solve the linear program introduced in Problem 3.18. We then measured the widths of the resulting drawings. Figure 3.48 plots the results while Table 3.8 has more details. There is a lot of variation in the data, but it appears safe to draw several conclusions.

First and unsurprisingly, the median strategy does not fare well in regard to diagram size. In fact, it is the worst performer among the strategies. The head and tail layer strategies perform better, which might be due to their

Table 3.8. Width of drawings produced by different label layer selection strategies.

	MN	SD	Min	Max
Median	1,663	2,186	228	16,513
Tail	1,600	1,801	228	11,968
Head	1,516	1,824	228	11,873
Widest	1,535	1,812	228	12,863
Space-Efficient	1,452	1,757	228	11,782
Optimal	1,429	1,760	228	11,782

tendency of "gathering" edge labels around nodes instead of spreading them out through the diagram, as the median strategy does.

Second, the size-aware strategies perform better than the simple strategies, with the exception of the rather primitive widest layer strategy. This does not come as a shock: we already suspected that calculating layer sizes based only on non-dummy nodes may not be the best idea. This shortcoming is the reason for the space-efficient strategy's existence, so it seems satisfying that it in fact outperforms the widest layer strategy. Moreover, it yields only slightly worse results than the optimal strategy, sporting a mean that is 200 pixels less than that of the median strategy. I thus feel it safe to conclude that the space-efficient strategy conforms to the **P-SIZE** principle.

Second Question To answer the second question on how successful the median strategy is at placing labels at the center of their edge, we looked at the horizontal coordinate span of each edge and measured where along that span the center points of its labels ended up. Values of 0.0 and 1.0 would indicate that a label's center point lies on the edge's left and right end points, respectively. A value of 0.5 would be ideal.

Figure 3.49 shows a box plot of the results. The median strategy produces a mean of 0.493 (SD of 0.085), which seems surprisingly good. As mentioned before in Section 3.3.3, investigating special size-aware strategies does not seem necessary in light of this result.

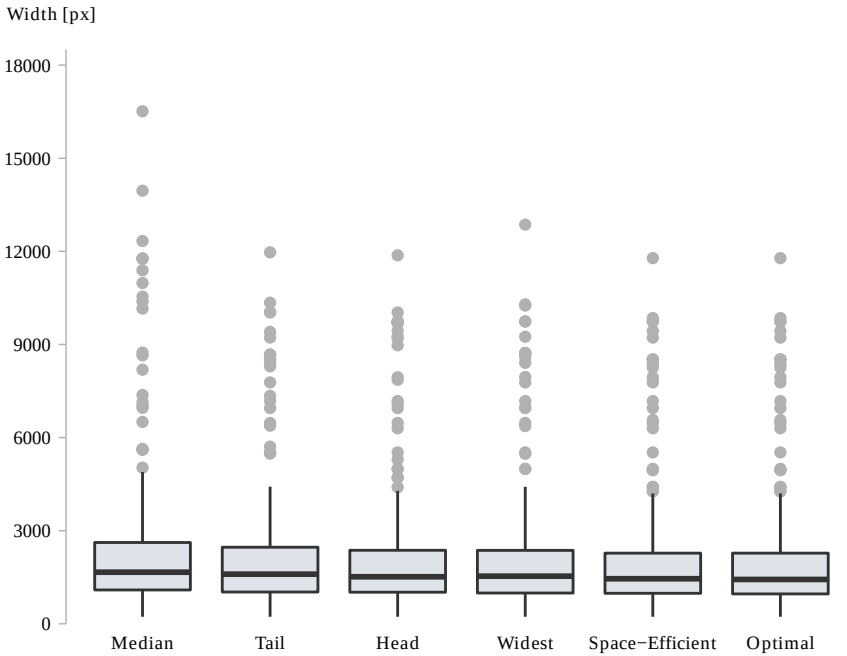


Figure 3.48. Width of drawings produced by different label layer selection strategies. Lower values are better.

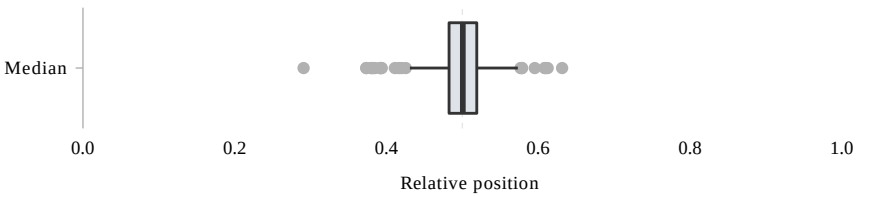


Figure 3.49. Relative positions of the center point of labels along the horizontal span of the edge they label. A value of 0.5 would indicate a label placed at the center of its edge.

3.4.3 A Controlled Experiment

Having a positive impact on the number of bend points or on the size of diagrams is one thing—effective communication quite another. Three questions appear in need to be answered:

1. How do the label placement strategies compare regarding graphic association?
2. Which strategy is better at indicating edge direction, traditional label placement with direction-dependent label side selection or on-edge label placement with directional decorators?
3. Does on-edge label placement have a negative impact on the ability of users to follow edges through a diagram compared to traditional label placement?

First Question The first question seems easy to answer if we only care about the distinction between on-edge and traditional label placement: graphic association can hardly get any clearer than with placing each label directly on the edge it belongs to. With properly chosen spacings, it also seems unlikely that the different label side selection strategies differ considerably in terms of graphic association, although placing labels consistently above or below their edges might have a slight edge ("pun irresistible," to quote Luciano Floridi [Flo13]) over directional placement strategies in dense layouts.

Second Question As for the second question, it seems obvious that adding explicit directional decorators makes edge direction more apparent than following direction-dependent label side selection conventions, again going back to the distinction between knowledge in the world as opposed to knowledge in the head [Nor88]. The question, however, remains how much of a difference there really is.

Third Question It is the third question that seems most important: the objective advantages of on-edge label placement, namely clear graphic

association and reduced space, would be of little value if it meant that users were not able to effectively follow edges through their diagrams anymore. We hypothesized that this might be the case due to the visual interruptions caused by on-edge labels as opposed to traditional label placement.

We designed a controlled experiment to help answer the latter two questions; since the answer to the first question seems rather obvious at least for on-edge versus traditional label placement, I did not include it in the experiment due to time considerations. The experiment consisted of three parts: one for the second question, one for the third question, and a concluding interview where participants were asked to rank different label placement strategies according to personal preference. The first two parts were performed using an application developed specifically for that purpose. Since the tasks to be solved during the experiment were not too time-consuming, we used a within-participants design throughout, that is, every participant was subjected to all of the conditions.

The collected data and any scripts used to conduct the subsequent analysis are available online.[6] The software used to conduct the experiment as well as all of the data necessary to repeat the experiment can be obtained by contacting the author.

We start by describing the experiment's overall procedure and then describe and analyze each part separately.

Procedure 48 participants between 18 and 33 years (averaging 23.35) were recruited for the experiment, 9 of them female. All participants studied computer science either as a major or as a minor (36 undergraduate students, 11 graduate students, and one PhD student). They were recruited via e-mail to the institute's central students mailing list, but none were in any way involved in the research that was the subject of this experiment. Participation was completely voluntary, but we compensated participants by paying them 5 Euros. 10 participants had prior experience with SCCharts. All experiments were conducted by a single experimenter and in the same room. A slight background hum was audible from an adjacent server room, but no participant reported this to have been a distraction, even if asked

[6] https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-1802-data.zip

directly. Most experiments took place between 1pm and 5:30pm, except for two that were conducted at 10am and 11am.

Participants were sat down in front of a computer screen with a resolution of 1920x1200 pixels and had a keyboard as well as a mouse available. However, only a screen to collect demographic data made use of the mouse; the main experiment was controlled purely through the keyboard.

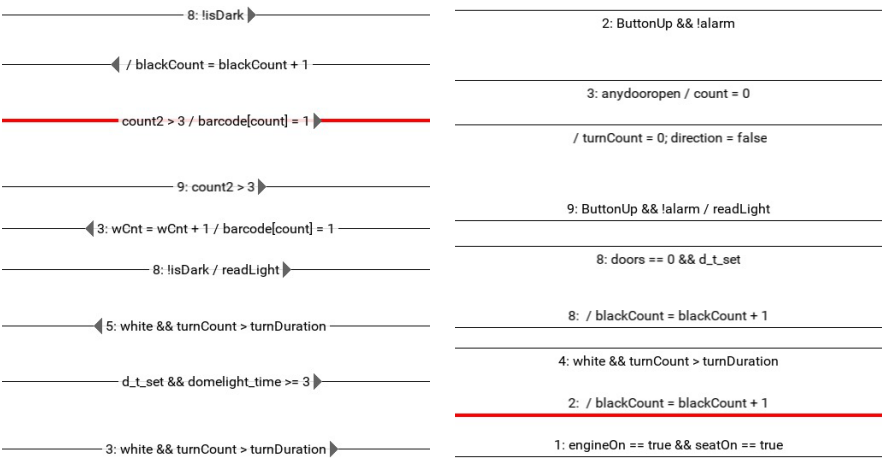
Participants were first asked to read a document that explained the experiment and the data to be collected, and then signed a declaration of consent. The experimenter then briefly explained the structure of the experiment before they were directed to follow the instructions on the computer screen, which led them through the first two parts of the experiment. A full run through the experiment would take a participant about 30 minutes to complete.

Thanks to five pilot runs, no significant problems arose during the experiments. One session was slightly disturbed by someone knocking and opening the door, but the participant did not feel that this was a significant problem—an assessment confirmed by scanning their data for anomalies.

Effectiveness of Directional Label Placement

The first part of the experiment was meant to help answer the question of whether on-edge label placement with directional decorators (OED) or next-to-edge label placement with direction-dependent label side selection (NED) is more effective at conveying edge direction. These two strategies thus served as the two conditions. Our hypothesis was that explicit directional decorators would be significantly more effective than implicitly encoding direction through placement side, in terms of both reaction time and error rate.

Experimental Method The experimental objects were designed to only show parts of edges to simulate a situation where users would have to rely on edge labels alone to infer where an edge is headed, as shown in Figure 3.50. This was to simulate typical use cases the placement strategies were designed for: focusing on a particular part of the diagram that does not include the end points of an edge. Each experimental object was generated



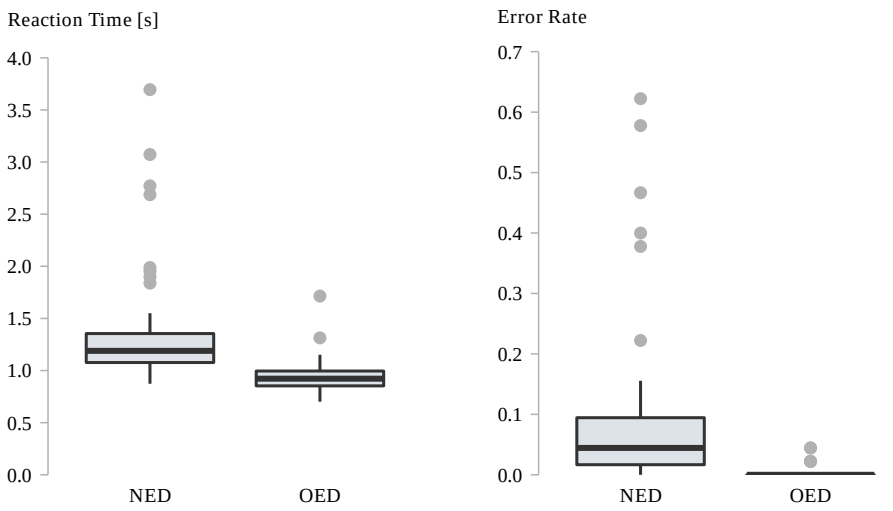
(a) Condition 1: OED **(b)** Condition 2: NED

Figure 3.50. Experimental stimuli for the two conditions of the experiment's first part. Participants were asked to infer the direction of the highlighted edge as quickly as possible. The correct answer would be "right" in both examples.

randomly, with 5 to 10 edges placed at a distance of 20 to 70 pixels, decorated with labels randomly generated based on labels found in SCCharts. One of the edges was highlighted in red and drawn as a thicker line to account for color perception deficiencies.

The task was to infer the direction of the highlighted edge and indicate it by pressing the left or right control key. The control keys were chosen over the arrow keys to reduce the probability of accidentally pressing a wrong key due to close proximity. We did not impose a time limit, but instructed participants to solve the task as quickly as possible.

The software first introduced participants to both edge label placement strategies and included five practice trials per condition to practice the task to be laid upon them, with visual feedback indicating whether or not a particular answer was correct. At this time they also had the opportunity to ask the experimenter for clarification on anything they did not understand.



(a) Reaction time

(b) Error rate

Figure 3.51. Box plots of the mean reaction times and error rates of each participant according to condition for the first part of the experiment.

Participants then performed three blocks of 30 trials each (15 per condition, shown in randomized order) and we measured the reaction time and recorded the correctness of each answer.

Data and Analysis Figure 3.51 shows box plots of the mean reaction time and error rate of each participant according to condition. Neither of the two sets of data could be considered to follow a normal distribution. Combined with the facts that the experiment used a within-participants design and exactly two conditions, we were led to use a Wilcoxon signed rank test with the standard significance level of 0.05.

We found significant differences ($p < 0.01$) between the conditions, with the OED condition taking less time and producing fewer errors than the NED condition.

Conclusions The results support our hypothesis: the OED condition outperforms the NED condition in both metrics. The error rate in particular also supports Norman's preference for information in the world: the error rate was generally higher for NED than it was for OED, and some participants seem to have gotten disoriented as to which edge direction a given label placement side indicates. As one participant put it, "This completely confused me, I always had to think about how it worked."

Of course, the task forced participants to focus only on edge direction, which in real-world scenarios is only part of what users do when reading and working with diagrams. I believe it to be likely that focusing on this task may actually have improved results for the NED condition in this experiment since there were no distractions that may have removed the (arbitrarily chosen) semantics of NED from working memory. Even so, 30 out of the 48 participants explicitly stated in the concluding interview that inferring edge directions with the help of NED was confusing, hard to remember, or required more thought.

Perceptual Impact of On-Edge Label Placement

The second part was designed to answer the question of whether on-edge label placement has a negative impact on the ability of users to follow edges through a diagram. Our hypothesis was that it might indeed, but we hoped that the effect would be small.

In addition to the two conditions already present in part one (OED and NED), part two added always-down next-to-edge label placement with same-side label side selection (NE) as a third condition. Since that strategy does not encode directional information, any significant performance difference (or lack thereof) would allow us to draw conclusions about the extent to which participants used those information to solve the task to be described next.

Experimental Method This time, the experimental objects were diagrams that fit completely on a single screen. In the interest of realism, we based the diagrams on nine excerpts from SCCharts developed by students as part of their homework during lectures throughout the past semesters, but modified

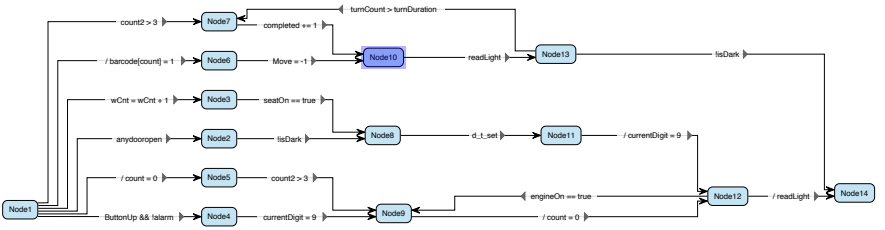


Figure 3.52. Experimental stimulus of the experiment's second part, in this case with on-edge labels with directional decorators. All objects were laid out using a left-to-right layout direction and drawn using orthogonal edges. Participants were asked to count the number of different nodes reachable from the highlighted start node (*Node10*) by traversing exactly two edges. The answer in this case would be two (*Node7* and *Node14*).

them to fit onto a single screen. The diagrams averaged 16 nodes and 22.8 edges and all but one had edge crossings. We obtained nine additional objects by reversing all edges of the original excerpts and switching node names around, resulting in diagrams that looked significantly different.

Given a start node, the task was to count how many different nodes could be reached from the start node by traversing exactly two edges (such *path-readability tasks* have been used in similar experiments [XRP+12; HW09]). This required participants to follow edges through the diagram, allowing us to draw conclusions about our research question. Again, we did not impose a time limit, but asked participants to solve the task as quickly as possible. Each of the 18 experimental objects had two possible start nodes defined, yielding a total of 36 different stimuli. Figure 3.52 shows an example of a typical stimulus.

Each trial began by drawing the experimental object on screen. The start node was highlighted one second later to avoid the need for participants to spend time searching the diagram for it. We measured reaction time (starting with highlighting the start node) and recorded the correctness of each answer.

The software first introduced participants to the task that would be their charge and allowed for three practice trials, with visual feedback indicating

which nodes are reachable in exactly two steps from the highlighted node. At this time they also had the opportunity to ask the experimenter for clarification on anything they did not understand.

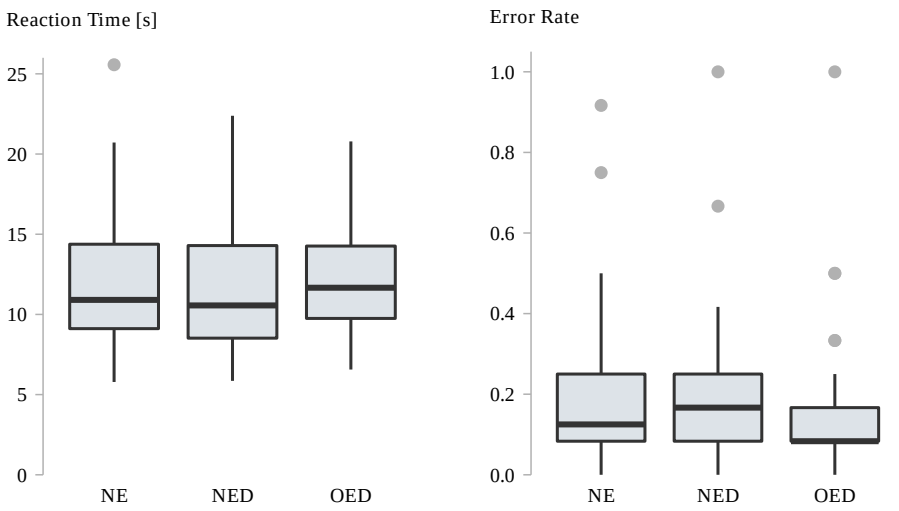
Participants then performed three blocks of 12 trials, one block for each condition. To average out any learning effects, the order of blocks was assigned based on a Latin square and each of the six possible permutations of the blocks appeared the same number of times across the 48 participants. Each of the 18 experimental objects appeared at most once in the same block to reduce the likelihood of participants recognizing a diagram and reacting more quickly than they normally would.

Data and Analysis Figure 3.53 shows box plots of the mean reaction time and error rate of each participant according to condition. It seems obvious that there are no significant performance differences between the three conditions in either of the two data sets—an observation confirmed by a Friedman test ($p = 0.33$ and $p = 0.09$ for the error rates and reaction times, respectively).

A single participant was responsible for the worst error rate in each condition. It is unclear why the error rate was this high since the participant mentioned nothing during the concluding interview that might explain where the problem was. However, even if this outlier was removed, the data would not become any more significant.

Conclusions The lack of significant differences in the results does not allow us to claim that either of the three conditions is better or worse regarding a user's ability to follow edges through a diagram. This might still an interesting observation, however: given 48 participants, any large performance differences between the three conditions may have been detectable. Still, we cannot draw conclusions either way.

A possible threat to the experiment's validity is that, depending on the stimulus, participants spent quite different times on solving their task. While we believe that the way we assigned stimuli made results of different participants comparable, future experiments should control for this.



(a) Reaction time

(b) Error rate

Figure 3.53. Box plots of the mean reaction times and error rates of each participant according to condition for the second part of the experiment.

Concluding Interview

The concluding interview was meant to shed light on which label placement strategies users preferred, and why. We also wanted to give participants an opportunity to give feedback about the experiment itself.

In addition to the side selection strategies they already knew from the first two parts (OED, NED, and NE), the interview added on-edge label placement (OE) (without directional decorators).

Experimental Method Participants were presented with the graph shown in Figure 3.52, drawn with the four side selection strategies, which they were asked to order from best (1) to worst (4). I asked them to try to assign each strategy a distinct preference rank, but accepted if they could not settle on an order between two or more strategies.

I made an audio recording of each interview—to which no participant objected—and asked them to explain their thought process while ranking the four strategies. The audio recordings were later transcribed by the experimenter to be coded into several categories. The coding was done separately by the experimenter and a colleague who was otherwise not involved in the experiment. Divergent opinions on how statements should be coded were resolved by going through the transcription together and discussing arguments for and against a particular coding.

Data and Analysis Out of 48 preference rankings we collected, only 6 had at least two strategies that shared the same ranking. Table 3.9 shows the 42 submissions that assigned unique ranks to each strategy, and Table 3.10 shows the remaining six.

For the purpose of this analysis, we consider *consistent rankings* to be rankings that follow consistent preferences regarding the underlying techniques. One such consistent ranking could be to always prefer next-to-edge placement to on-edge placement and to break ties by preferring directional decorators; another could be to prefer directional decorators and to break ties by preferring on-edge placement to next-to-edge placement. A first interesting observation is that consistent rankings are the minority (18 out of 42). Among the remaining 24 inconsistent rankings, 18 rank OED highest and 13 of these rank NED lowest, possibly due to problems when inferring edge directions with the latter strategy.

Figure 3.54 shows the sums of the ranks assigned to each side selection strategy, which already indicate that the OED condition may have an advantage over the others. The OE condition also seems to have been ranked slightly better than both NE and NED, but the difference is far from being as pronounced. Figure 3.55 shows histograms of ranks assigned to each strategy. The histogram for the OED condition seems to support the former statement. The second statement, however, is not obvious.

Since graphics do not confirm the existence of statically significant ranking differences, we performed a Friedman test on the data and found that significant differences do in fact exist ($p < 0.01$). We proceeded with a pairwise sign test and found that the only significant differences in preference rankings involved the OED condition, which was ranked significantly better

Table 3.9. Preference assignments with distinct ranks submitted by participants. The top half of the table contains rankings that follow consistent preferences regarding the underlying techniques when ranking strategies.

Rank 1	Rank 2	Rank 3	Rank 4	Participants
NE	NED	OE	OED	2
NE	OE	NED	OED	0
NED	NE	OED	OE	2
NED	OED	NE	OE	0
OE	NE	OED	NED	0
OE	OED	NE	NED	4
OED	NED	OE	NE	4
OED	OE	NED	NE	6
NE	NED	OED	OE	0
NE	OE	OED	NED	1
NE	OED	NED	OE	0
NE	OED	OE	NED	2
NED	NE	OE	OED	2
NED	OE	NE	OED	0
NED	OE	OED	NE	0
NED	OED	OE	NE	1
OE	NE	NED	OED	0
OE	NED	NE	OED	0
OE	NED	OED	NE	0
OE	OED	NED	NE	0
OED	NE	NED	OE	0
OED	NE	OE	NED	3
OED	NED	NE	OE	5
OED	OE	NE	NED	10
Consistent				18
Inconsistent				24

Table 3.10. Preference assignments with shared ranks submitted by participants. Each assignment was submitted only by a single participant.

Rank 1	Rank 2	Rank 3
NE	OE, OED	NED
NED, OED	OE	NE
OE	NE, NED	OED
OED	NE, NED	OE
OED	NE, NED, OE	
OED	OE	NE, NED

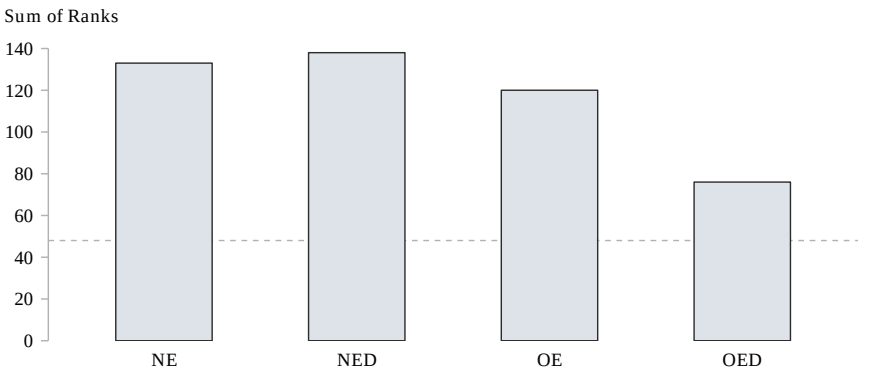


Figure 3.54. Sums of the ranks assigned by participants to each label side selection strategy from 1 (best) to 4 (worst). Lower bars indicate overall higher preference. The dashed line marks the best score a condition could have reached (48).

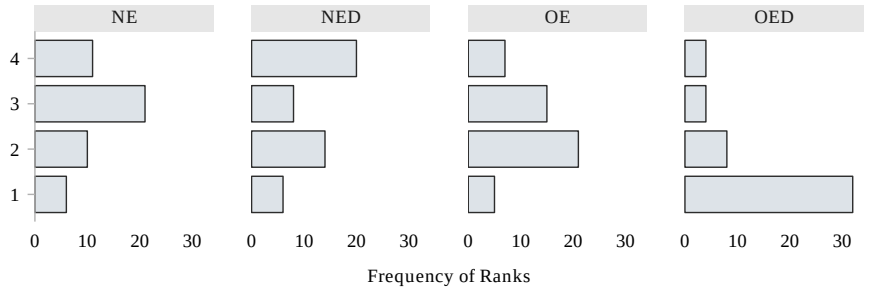


Figure 3.55. Histograms of the occurrences of each ranking for each label side selection strategy, from 1 (best) to 4 (worst).

than all other conditions (always with $p < 0.01$). The OE condition did not perform significantly better than either NE or NED.

Regarding the interviews, Table 3.11 shows a summary of how often the most important statement categories as determined while coding the interviews occurred for each condition. For the most part, the results are not surprising. 24 participants mentioned clear graphic association as an advantage of on-edge label placement, while 21 and 11 participants mentioned unclear graphic association as a disadvantage of NED and NE, respectively. When it comes to the ease of tracing edges through a diagram, the numbers are reversed: 13 mention traceability as an advantage of the next-to-edge strategies, while 16 mention it as a disadvantage of the on-edge strategies.

11 called NED confusing and inconsistent, and 27 said that it was hard to infer edge directions with. However, 10 mentioned that it might just be a matter of getting used to this strategy. 5 participants stated explicitly that this strategy requires explanation since encoding edge direction through placement side would not be obvious to them.

33 found OED to be helpful for inferring edge directions and 17 generally found the additional directional information to be helpful, although another 10 did not. 7 opined that the usefulness of this strategy improves as edges get longer and graphs get larger.

13 participants stated that the best strategy to use would depend on the use case.

Table 3.11. Statements from interviews coded to several categories, subject to label side selection strategy. Answers for some categories are counted towards two strategies instead of just one.

Category	NE	NED	OE	OED	NE, NED	OE, OED
Graphic association is clear	3	1				24
Graphic association is unclear	11	21				0
Tracing edges is easy					13	5
Tracing edges is hard					0	16
Looks clear or consistent	11	2	3	7		
Looks confusing or inconsistent	1	11	0	3		
Direction easy to infer		3		33		
Direction hard to infer		27		0		
Additional information helpful		11		17		
Additional information unhelpful		4		10		

As for feedback regarding the experiment itself, there were very few complaints. Three participants thought some of the graphs were scaled down too much in order for them to fit on the screen, one found the task in the second part to be hard, one would have appreciated more practice tasks, and one would have liked to see their results at the end.

Conclusions There is little question that, overall, participants largely preferred the OED strategy, for different reasons. First, participants found it easy to infer an edge's direction with OED, something which they found hard with NED (although interestingly one participant thought he was quicker inferring directions with NED, an assessment proven wrong by the data). Second, clear graphic association was mentioned quite often. As one participant said about NED, "I would have problems associating labels with edges if there were too many edges in one place. It's a clarity thing." However, some complained about visual clutter and redundant information in the diagram. This might well be due to the fact that the value of additional directional markers increases with diagram size—the diagrams in the exper-

iment, however, were designed to be small enough to fit on a single screen to keep users from having to scroll through them. It might be valuable to make the appearance of directional decorators depend on the length of an edge. As one participant said, "For small diagrams I think I would prefer OE. If it is a large diagram with long edges, I would prefer OED. Especially in small diagrams with many edges, where everything is close by, it gets a little confusing."

While it is understandable that many participants found graphic association to be a weakness of NED, it is interesting that 11 thought the same about NE. This might be due to a bad impression of NED carrying over to the (superficially similar) NE strategy. It might however also be due to the fact that NE becomes clear only once users are aware that labels are always placed below or above their edges.

A possible threat to validity is that the first part of the experiment may have somewhat primed participants to prefer the OED condition for its obvious advantages in inferring edge directions. However, we believe that effect to be limited since 50% of participants explicitly mentioned clear graphic association as an advantage of the OE and OED strategies, while 50% also explicitly mentioned that NED suffered from unclear graphic association—an observation that the first task would probably not have primed for as much.

3.4.4 Two Surveys

To get an idea of how well the different label side selection strategies perform in practice, we conducted two additional surveys among the students of our lecture on the "Design of Embedded Real-Time Systems." They used SCCharts extensively in their homework assignments, where they had five label side selection strategies at their fingertips: the four that already featured in the controlled experiment (NE, NED, OE, and OED) as well as next-to-edge label placement with augmented same-side label side selection (NEA). We conducted the first survey during and the second survey at the end of the semester.

First Survey

The first survey took place during the eighth set of homework assignments. Already having been introduced to SCCharts, students were asked to use the language to program a *Lego Mindstorms* robot to read bar codes printed on the surface it was driving around on (Figure 3.56 shows a typical solution produced by students). They had a good three weeks available to complete the assignment, which 16 students did, divided into five teams of two and two teams of three students each. Completing the survey earned them points for the assignment set.

The survey consisted of two tasks:

1. Rate the label side settings from best to worst and explain your rating.
2. Provide any thoughts and comments you might have on the label side settings.

The sums of the ratings given to the different strategies are shown in Figure 3.57. As in the controlled experiment, OED was rated most favorably, although not by as large a margin. Both on-edge label placement strategies received mixed reviews. On the negative side, one team found on-edge labels to reduce their ability to follow transitions through their SCCharts and to read the actual text. Three teams, on the other hand, explicitly praised the clear graphic association on-edge labels achieve, as well as that they allow for more compact diagrams.

As in the controlled experiment, three teams described NED as a negative influence on readability or as being downright confusing. Interestingly, though, NE received the worst ratings overall. Opinions were varied, from two teams appreciating its graphic association, to one that thought it was downright "boring" and "inefficient." The latter team explained that "with many transitions close by, one has to look for the outermost ones to be able to correctly associate labels with transitions." This is an interesting point: even placing labels consistently above or below edges may not always be enough to establish clear graphic association without additional information in the head, it seems.

The NEA strategy, which was not part of the controlled experiment, did receive praise. One team stumbled upon cases where rules two or three

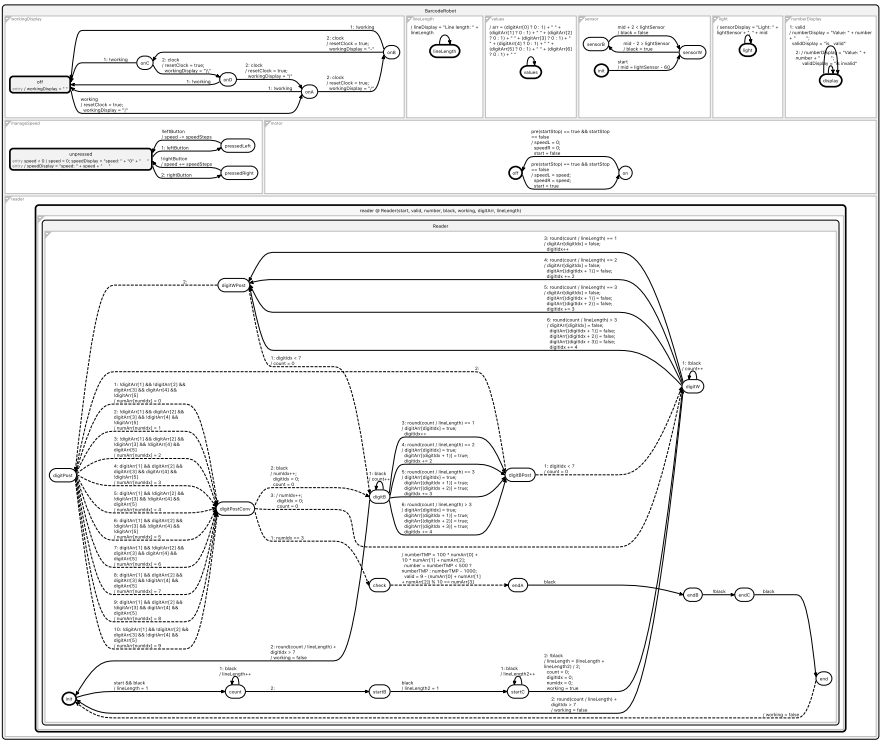


Figure 3.56. A typical SCChart students would produce to make a *Lego Mindstorms* robot read bar codes. While not exactly legible at this scale, it does serve to illustrate the size of the models produced by students while they responded to the first survey. To increase scaling at least somewhat, label management was applied to the transition labels, neatly foreshadowing Chapter 6. Reprinted with permission.

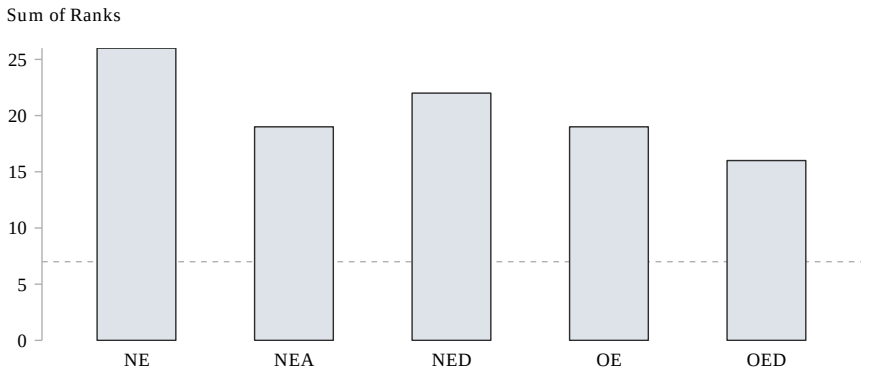


Figure 3.57. Sum of the ranks assigned by students to the five label side selection strategies during the semester. Lower bars indicate overall higher preference. The dashed line marks the best score a strategy could have reached (seven).

applied and liked how the strategy handled them. Another team stated that it helped them "distinguish transitions while working with a lot of labels in small space."

Regarding the second assignment, several teams provided interesting insights. One said that on-edge label placement with directional decorators "would be awesome if the arrow was a bit more visible" since they used that placement strategy primarily in situations where they wanted to get an overview of their SCChart. Another team joined in, remarking that they "like the arrowed edge mode so an option to darken the arrow or enlarge it would be nice."

Second Survey

The second survey took place at the end of the semester, as part of a more general SCCharts survey. Students were asked to answer the following question: "Since their introduction, which label side selection strategies did you end up using regularly?" Multiple selections were possible.

The results, shown in Figure 3.58, are consistent with the controlled experiment in that the directional label side selection strategy does not fare

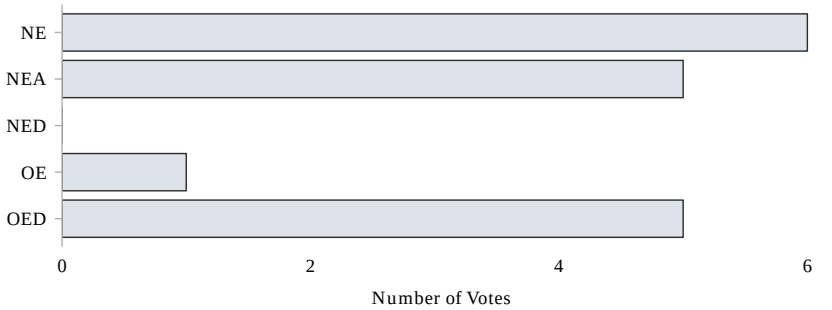


Figure 3.58. The label side selection strategies used by the eleven students remaining at the end of the semester. Multiple answers were possible. Longer bars are better.

well with users. There do not seem to be differences between the other strategies, except for on-edge label placement, which was largely preferred in its variant with directional decorators displayed.

Asked for further comments, which five students provided, one admitted that "as I was already used to the consistent side variant, I stuck with that most of the time." Two explicitly highlighted on-edge labels, one noting that "the on-edge functionality is incredibly useful in SCCharts with increased complexity" and the other simply stating that he thought "on edge with arrows is a great idea."

Given the small sample size, in either survey, we can in no way assume that we can generalize any results. The surveys do, however, affirm the results from the study in that users generally seem to prefer on-edge labels. The second survey might add that the augmented same-side strategy can be regarded a valuable alternative to traditional same-side placement.

3.4.5 Discussion

We have discussed and evaluated a number of approaches that relate to the **P-PLACEMENT** principle. Both the controlled experiment and the surveys suggest that the NED label side selection strategy is considerably less successful than anticipated. Both graphic association and the inference of edge directions are inferior to the OED strategy's performance. In spite of

the praise of a minority of participants, it appears safe to advise against using NED in real-world applications.

Among the remaining strategies, recommendations are less obvious. Next-to-edge label placement is still what users are accustomed to the most. The NEA strategy might be the way to go here since it aims to fix certain unfortunate placements, but NE has the advantage of always maintaining consistency—a quality that garnered praise, but also disapproval from participants. Regardless of which strategy one settles for, spacings always need to be chosen carefully so as not to threaten clear graphic association.

This is not a problem for the on-edge strategies. Overall, these seem to have performed rather well and would be recommended were it not for the challenges they pose regarding the visual design of labels. This is a critical point to get right indeed: the feedback collected from participants supports the hypothesis that on-edge labels can disrupt or appear to disrupt the "flow" of edges (even with participants that complained about disruption, we did not find that they performed significantly worse than with the other placement strategies). Whatever the case, the designer's aim must be to minimize any such effects—whether real or perceived—as much as possible.

Closely related are the arrows shown by the OED strategy. Some participants found them to clutter up parts of the diagram dominated by short edges. There are different possible solutions to this problem. Most obviously, the design of label arrows could be changed to be blend more into the background until they are explicitly focussed. Another would be to hide arrows, either as the length of their edge falls below a threshold value or as their edge's real arrow head becomes visible on screen. A more dramatic solution might be to get rid of arrows at the end points of edges, letting the label arrows replace them entirely. All of these solutions have their particular drawbacks, either in terms of confusion they might cause or helpful information not being displayed where users need it.

More experiments seem to be called for to investigate the effectiveness of on-edge label placements further. First, it seems plausible that a single-line label is easier to follow than a multi-line label due to its visual semblance to a line. Whether this is indeed the case and, if so, how large the effect is would have to be established empirically. The second point concerns the design of on-edge labels, of which we saw examples in Figure 3.44.

Delimiting labels with lines or boxes establishes further visual barriers which may degrade the ability of users to follow edges through diagrams. Again, the existence as well as the extent of any such effects need to be established empirically. Finally, all such potential drawbacks may be more or less pronounced in vertical layouts since the area of edge interruption is smaller. This, too, warrants additional investigations.

This chapter has focussed on handling text when laying out flow-based diagrams. In the next chapter, we will turn to UML sequence diagrams.

Chapter 4

Sequence Diagrams

The previous chapter was all about using the well-known layered approach to lay out flow-based diagrams, of which Ptolemy II block diagrams and SCCharts are examples. This chapter finally hones in on our third and final language: UML sequence diagrams. We will describe ELK Sequence, my layout algorithm for sequence diagrams, which—besides constituting an additional use case for subsequent chapters—knows how to integrate labels and comments into the layout process and aims to produce smaller diagrams, thus following the **P-Placement** *and* **P-Size** *principles.*

While we refer to the official standard for a complete definition of sequence diagrams [Obj17, Section 17.8], Section 1.1.3 already provided an overview of some of the constructs available in sequence diagrams. Sequence diagrams offer more, such as *duration constraints* and *duration observations*, but these seem to be most prevalent ones. Since our main motivation for introducing automatic layout to sequence diagrams is to have another demonstrator for the techniques to be introduced in Part II, we will leave the less frequently used elements out of our discussions for the remainder of this chapter. Their inclusion into the layout algorithm about to be described is for the most part obvious.

Our journey through sequence diagrams will start with how they can be represented as graphs, continue with our layout algorithm for sequence diagrams, and finish with its application in our sequence diagram editor based on a custom DSL called KieSL, before we evaluate the layout algorithm through an aesthetics-based evaluation.

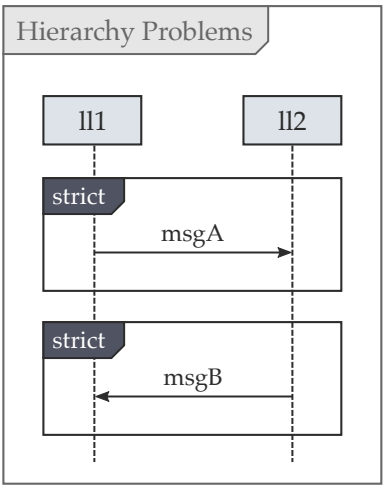


Figure 4.1. Sequence diagrams that include combined fragments are not traditional hierarchical graphs as we understand them in this thesis.

4.1 Sequence Diagrams as Graphs

Sequence diagrams are not graphs as we defined them, although they may seem to be at first glance. If messages are sent between lifelines, then why should we not be able to understand them as edges that connect nodes?

The problem arises once we introduce combined fragments. Consider the small sequence diagram in Figure 4.1. It seems simple enough: two lifelines communicate by exchanging two messages, each part of a separate fragment. It is the latter circumstance, however, that we cannot easily express using a purely hierarchical graph data structure. There is clearly some sort of hierarchy involved, but we cannot find a proper inclusion tree.

The usual approach to solve this kind of problem would be to devise a class hierarchy that can properly describe sequence diagrams and be done with it. There is one problem, however: making a layout algorithm available through ELK's infrastructure entails using ELK's interfaces, which includes accepting ELK graphs as the agreed-upon way to specify layout

Table 4.1. This table shows which ELK graph elements are used to represent the elements of a sequence diagram.

ELK Graph Element	Sequence Diagram Element
Node	Sequence diagram Surrounding interaction Lifeline Execution specification Combined fragment State invariant Destruction occurrence specification Source of found message Target of lost message Comment
Edge	Message
Label	Message text Comment text

problems. We thus need to think about how to describe a sequence diagram with a graph structure that was not designed for this kind of diagram. The answer lies in the ELK graph's properties mechanism: if we cannot express all relations between diagram elements through the graph's structure, we can resort to properties to establish them instead.

Before we do so, however, we need to define which ELK graph element is used to represent which sequence diagram element. Table 4.1 shows an overview of the element mappings. A sequence diagram itself is represented by an ELK graph's root node, which in turn contains a single child node that represents the interaction. That, then, contains further child nodes that represent virtually everything the interaction contains. Messages are represented by edges that always connect lifelines.

That last point might seem curious since not all messages connect directly to a lifeline—some are incident to execution specifications instead. In fact, we just stumbled upon a first example of a relation that is not expressed directly through the graph's structure, but through properties:

each message uses them to indicate which execution specifications it is incident to.

To be able to do so, we require each graph element to have an *element identifier* set on it, which is nothing more than a unique integer. The identifier can then be referenced by other elements to establish relations between them. Returning to our example of messages and execution specifications, each message can now specify which executions it is incident to through lists of identifiers, one each for its source and for its target lifeline. All in all, the following relations are specified using properties:

Parent lifelines. Execution specifications and destruction occurrence specifications always belong to a specific lifeline, their *parent lifeline*.

Fragments. The list of fragments a given message belongs to.

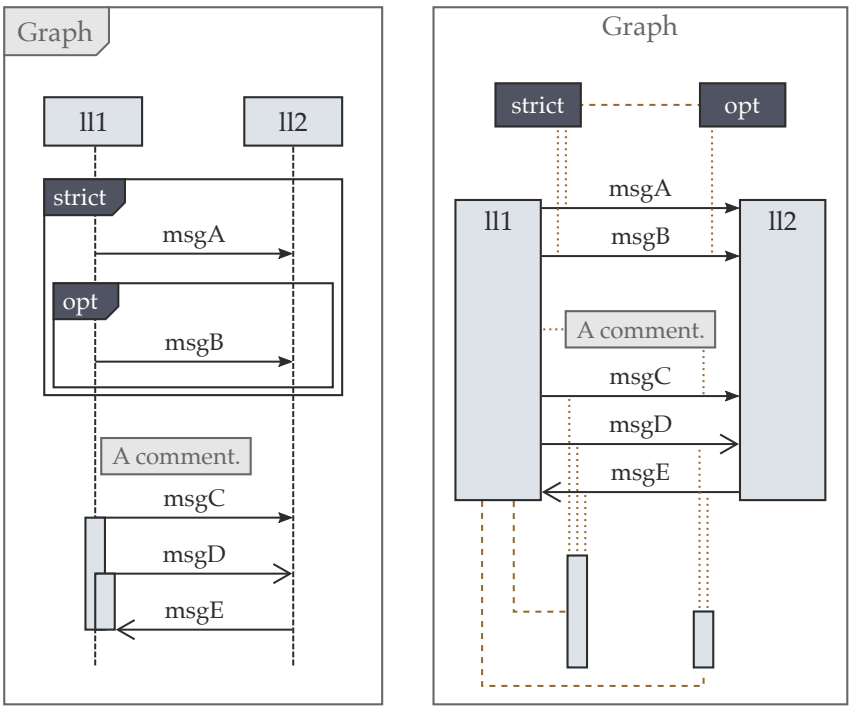
Parent fragment. Since fragments can be nested, this relation establishes hierarchies between them.

Source and target executions. Establish which execution specifications are active at the end points of a message, both for its source and for its target lifeline.

Attached elements. Comments can relate to certain diagram elements. This will usually be one element, two elements (say, a message and one of the lifelines it connects), or no element at all.

Figure 4.2 shows a sequence diagram and its corresponding ELK graph representation, including the relations just introduced.

Note that this set of relations is by no means the only way of adding the necessary information to the graph. For example, instead of specifying the parent lifeline of executions and destruction occurrences, we could do it the other way around and have them be referenced by their parent lifeline instead. While this would admittedly not make much of a difference, another way of modeling their relations would: since these are strict parent-child relationships, we could use the graph's natural hierarchy to express them by adding execution nodes to the list of children of lifeline nodes. We do not go with this option for two reasons. First, putting all elements



(a) Sequence diagram **(b)** ELK graph

Figure 4.2. A simple sequence diagram and its structural representation as an ELK graph. Dotted lines hint at reference relationships between elements while striped lines hint at parent-child relationships. All of these relations are established through properties.

on the same level of hierarchy places them in the same coordinate system, which makes interpreting layout results easier. And second, introducing hierarchy would require clients to configure their graphs such that the graph layout engine hands the whole hierarchy over to the sequence diagram layout algorithm instead of executing layout separately for each level of hierarchy—an additional configuration step we wanted to avoid, especially since executing layout separately does not make any sense.

Being able to express sequence diagrams as ELK graphs indicates the graph's flexibility as a data structure, which is mainly due to the properties mechanism. In fact, we have used properties in other projects to express *higraphs* [Har88], another kind of graph whose inclusion tree is not actually a tree, but an acyclic directed graph.

4.2 Laying Out Sequence Diagrams

Due to the rigorous visual structure of sequence diagrams it may seem that they do not make for a particularly interesting layout problem. After all, the order of messages is fixed within each lifeline, and the lifelines themselves are simply placed next to each other along a horizontal line. I believe, however, that this is a misconception. The order of lifelines is free and determines the number of crossings between messages and lifelines. Also, the exact coordinates of message end points, while constrained, is not fixed and can impact readability. ELK Sequence aims to capitalize on these degrees of freedom.

When it comes to layout adjustment versus layout creation, it seems hard to cleanly classify the algorithm as being of the one or the other variety. The order of messages at each lifeline needs to be specified in the input graph and cannot be changed in any way by the algorithm since that would change the diagram's semantics (we tend not to give layout algorithms this kind of power). The order of lifelines, however, can be changed, and it is here that the algorithm can operate according to either category: it can either keep the order of lifelines stable, or compute a (possibly) new order subject to certain aesthetic criteria. Hence one could classify ELK Sequence

as being a layout adjustment algorithm in the diagram's vertical dimension, and both in the horizontal dimension.

In accordance with the vertical distance constraint defined by Poranen et al. [PMN03], ELK Sequence divides the diagram into horizontal *communication lines*, a configurable amount of space apart, and restricts messages to run along these lines only. Referring back to Chapter 3, the horizontal lines are reminiscent of layers which messages are assigned to. The analogy is quite fitting, and ELK Sequence actually draws upon parts of the implementation of ELK Layered, as we will see. This also allows messages to share a communication line, in contrast to other sequence diagram tools—at least ones based on automatic layout—which usually assign each message to its own communication line, ordered by appearance in the original diagram (or a textual description thereof). This of course results in taller drawings, whereas allowing messages to share communication lines gives us more freedom to optimize the sequence diagram's height. We refer to this as *vertical compaction*, which ELK Sequence allows to be switched on or off.

That minimizing the height of sequence diagrams is a worthy goal becomes apparent when looking at real-world diagrams. In fact, IBM have gone to the trouble of establishing best practices in designing sequence diagrams to be used by "authors of technical material" [BMM04] for that express reason, among others. They note that "sequence diagrams, even those that depict relatively simple processes, can quickly become overly large, unwieldy, and impractical for technical documentation." In due course we will see, however, that vertical compaction does complicate parts of the algorithm considerably.

ELK Sequence makes use of the same algorithmic framework as ELK Layered (see Section 3.1.6). It is structured into five phases, each with clearly defined responsibilities:

1. Lifeline ordering
2. Space allocation
3. Cycle breaking
4. Communication line assignment

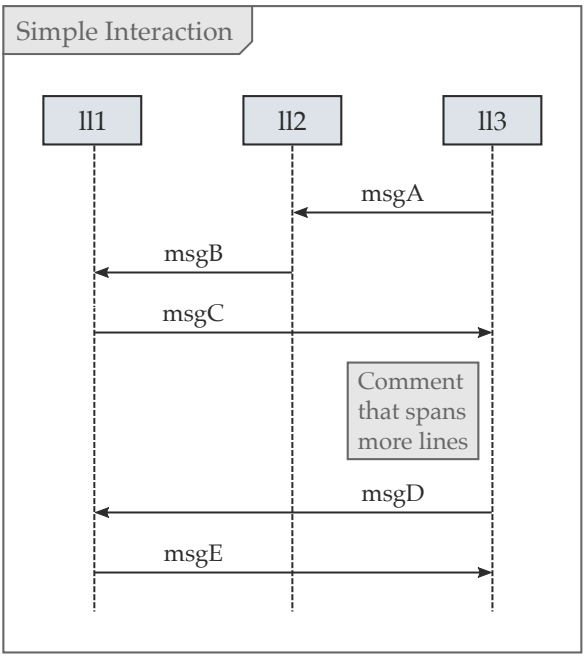


Figure 4.3. This simple sequence diagram will be our main example for explaining how the algorithm works.

5. Coordinate assignment

The set of phases is surrounded by transformations that import an ELK graph into the *sequence graph* data structure used by the algorithm, and that export the layout results back to the ELK graph.

The following sections will describe each of the algorithm's phases. Many of the descriptions will be based on the simple sequence diagram shown in Figure 4.3, except for when we need to look at exceptional cases.

4.2.1 Phase 1: Lifeline Ordering

The algorithm starts out by tackling what it has the most control over: determining the order of lifelines. ELK Sequence provides three different ordering algorithms: *interactive order*, *communication line order*, and *short message order*. Before we delve into the rationale behind and a description of each strategy, there is one thing we need to look at first.

The latter two strategies require further information about dependencies between messages, which is why the first phase begins by calculating an *element ordering graph*. Therein, each message is represented by a node and an edge runs from node x to node y if the following conditions are met:

1. There is a lifeline both messages are incident to.
2. The message represented by x immediately precedes the message represented by y at that lifeline.

We call this an *element ordering constraint*, and it must be adhered to in the final layout by assigning the involved messages to different communication lines.

If vertical compaction is not enabled, further constraints are added to the graph to ensure that each messages is assigned its own communication line. We order the messages by their vertical coordinates and introduce a constraint between each pair of adjacent messages, unless one already exists. Note that the coordinates are used only to establish an order among the messages and thus need not be based on any kind of existing drawing of the sequence diagram. For a diagram generated from a textual description, it may for example be enough to simply use the line number a message was defined in as its vertical coordinate.

The element ordering graph is kept along with the algorithm's main data structure, the sequence graph.

Interactive Order

The simplest of the three ordering algorithms, the *interactive order* algorithm, simply retains the lifeline order from an input graph. The idea here is to order the lifelines exactly as the user specified them. If the sequence

diagram is specified through a textual language, this order conforms to the user's mental map. This algorithm is what other sequence diagram editors implement as well.

Similar to the sketch-driven phase implementations of ELK Layered, the interactive order lets ELK Sequence operate as a layout adjustment algorithm. The result corresponds to our input graph in Figure 4.3.

Communication Line Order

The idea of this algorithm is to order lifelines in a way that makes following the communication through the diagram as easy as possible. To achieve that goal, it chooses one of the topmost messages since those can be considered the interaction's starting points. That message's source lifeline is assigned the leftmost slot in the diagram. The algorithm then traces the message exchange through the involved lifelines, assigning them to consecutive slots as it first encounters them. The result is shown in Figure 4.4.

The drawings produced by this algorithm adhere to the starting object constraint defined by Poranen et al. [PMN03] which requires the lifeline that initiates the interaction to be the leftmost one. Their criterion of slidability should not be confused with our goal here. While layouts with good slidability lend themselves well to being browsed through from top to bottom with occasional horizontal shifts, the layouts produced by this algorithm will generally tend to be browsed through from left to right.

Algorithm 4.1 shows the algorithm in more detail. The main loop runs until all lifelines are placed. Each iteration starts with a call to the `findUppermostMessage(...)` function (line 3). Among the outgoing messages of all unprocessed lifelines, it computes the subset of messages whose corresponding nodes have no predecessor in the element ordering graph. From this subset it returns the message whose source lifeline has the biggest difference of outgoing message count minus incoming messages count, the idea being to have lifelines with lots of outgoing messages move to the left of the diagram. If there is no outgoing message left among the unprocessed lifelines, we add the remaining lifelines to our result in an arbitrary order (lines 4 to 6).

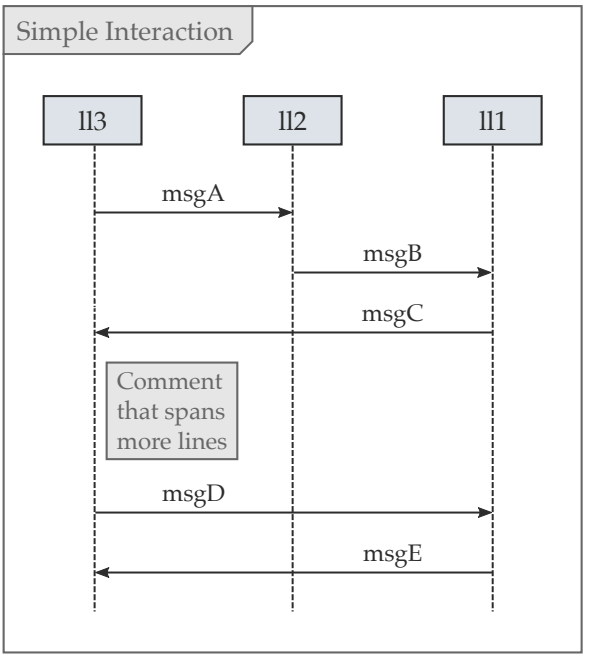


Figure 4.4. The lifeline order as computed by the communication line order algorithm. Compared to Figure 4.3, *ll0* and *ll2* have switched places.

If we did find an uppermost message, we add its source lifeline to the end of our list (lines 8 and 9). If the message's target lifeline has not been placed yet, we do so (lines 11 to 13) and then proceed to find the uppermost message that connects it to a lifeline that has not been placed yet by calling the `findUppermostOutgoingMessage(...)` function (line 14).

To take our input graph as an example, we start with all three lifelines unprocessed. The algorithm finds that *msgA* is the uppermost message and thus adds *ll3* to the as yet empty list of lifelines. It then proceeds to add the message's target, *ll2*, to the list as well and looks for its uppermost outgoing message that is headed for an unprocessed lifeline. It finds *msgB* and adds *ll1* to the list, at which point there are no unprocessed lifelines left.

```
Input: lifelines, the set of lifelines to be ordered
Output: Ordered list of lifelines
1  result ← empty list
2  while lifelines ≠ ∅ do
3      msg ← findUppermostMessage(lifelines)
4      if msg is undefined then
5          Add lifelines to result in arbitrary order
6          lifelines ← ∅
7      else
8          Remove source(msg) from lifelines
9          Add source(msg) to result
10         repeat
11             if target(msg) ∈ lifelines then
12                 Remove target(msg) from lifelines
13                 Add target(msg) to result
14             msg ← findUppermostOutgoingMessage(target(msg), lifelines)
15         until msg is undefined
16 return result
```

Algorithm 4.1. The communication line lifeline ordering algorithm.

Short Message Order

None of the preceding algorithms minimizes the length of messages or the crossings they cause. The culprits in our example diagram are the three bottommost messages, which in both layouts so far span the whole diagram. Besides the message length itself, longer messages quite obviously also produce more crossings with lifelines, which may harm readability. The third and final algorithm thus aims to optimize for message length, as Figure 4.5 shows. Doing so, however, turns out to be a hard problem, as already pointed out by Poranen et al. [PMN03].

Let L be a set of $n \in \mathbb{N}$ lifelines and let $m(l_1, l_2)$ be the number of messages from l_1 to l_2 or vice versa for $l_1 \neq l_2 \in L$ (zero for $l_1 = l_2$). What we are looking for in our quest to order the lifelines is a bijective assignment $\mathcal{L} : L \to \{1, \ldots, n\}$ (we reuse a symbol here that we previously used to

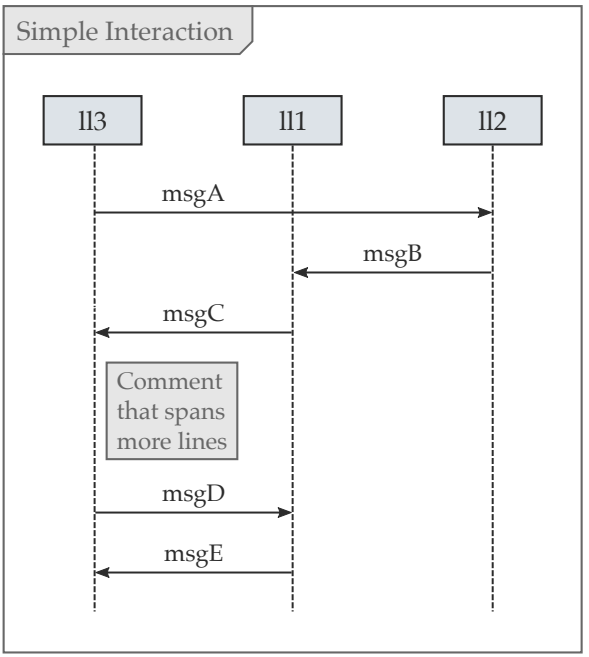


Figure 4.5. The lifeline order as computed by the short message order algorithm. Compared to the two preceding results, which produced a total message length of 8 each, this algorithm gets away with 6.

describe layer assignments). The length of a message that connects l_1 and l_2 is exactly $|\mathcal{L}(l_1) - \mathcal{L}(l_2)|$, and the number of crossings it produces is $|\mathcal{L}(l_1) - \mathcal{L}(l_2)| - 1$ (unless it is a self message, in which case it does not produce any crossings).

Poranen et al. list two different goals one might want to optimize for [PMN03]:

▻ The first would be to minimize the weighted sum of the length of all messages:

$$\sum_{\{l_1, l_2\} \subseteq L} m(l_1, l_2) \cdot |\mathcal{L}(l_1) - \mathcal{L}(l_2)|.$$

This is a weighted version of the *linear arrangement problem*, which is NP-complete already in the unweighted case [GJ79].

▷ The second would be to minimize the length of the longest message. This is equivalent to the *bandwidth minimization problem*, shown to be NP-complete by Papadimitriou [Pap76].

We accept a few long edges to gain mostly short edges and thus optimize for the first goal. Our implementation is based on an algorithm proposed by McAllister [McA99], which, like many algorithms for bandwidth minimization, is split into two phases:

1. Selecting a start lifeline. This will end up being the leftmost lifeline in the diagram.

2. Iteratively selecting lifelines for the remaining $n - 1$ slots.

We make two adjustments to McAllister's algorithm to adapt it to the requirements of sequence diagram layout. First, we select the start lifeline as we did with the communication line order algorithm to have the leftmost lifeline be the one that starts the interaction. And second, we introduce an option to increase the weight of messages that are contained in combined fragments. The algorithm then tries harder to keep such messages short, resulting in smaller fragments.

4.2.2 Phase 2: Space Allocation

The algorithm will soon use the element ordering graph to assign messages to communication lines such that no two messages with direct or transitive ordering constraints end up on the same line. If the algorithm only considered messages, however, only the space between adjacent communication lines would be available for placing other diagram elements, which will usually not suffice.

To solve such problems, Phase 2 introduces additional nodes to the element ordering graph to represent comments and the header area of each combined fragment. We may end up creating more than one node for a diagram element. Figure 4.6 shows the extended element ordering

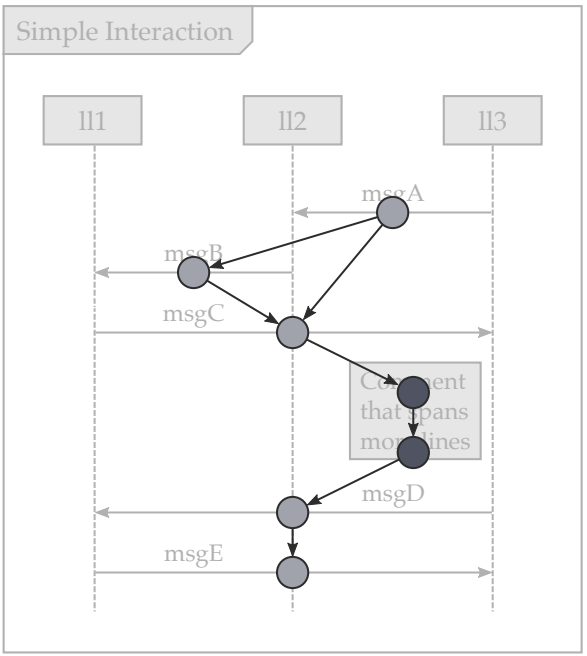


Figure 4.6. The element ordering graph the second phase produces for our example diagram. Note how the successor constraint from *msgC* to *msgD* was broken to reserve space for the comment.

graph created for our example diagram from Figure 4.3. The comment is so tall that it requires two communication lines of space, which is why it is represented by two nodes in the element ordering graph.

Unless vertical compaction is switched off, there is yet another problem to be solved. Consider the sequence diagram in Figure 4.7. Four lifelines are connect by a series of six messages, of which the first three are part of a combined fragment. Without further provisions the layout algorithm may end up moving *msgD* into the combined fragment, and with good reason: so far, the element ordering constraints for *msgD* only require that it be placed below both *msgA* and *msgB*, since those are the messages it shares

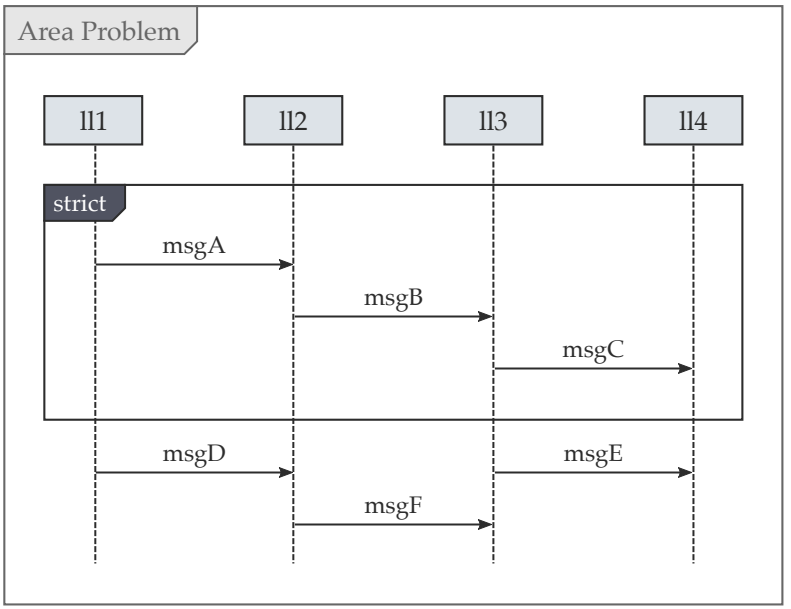


Figure 4.7. Without further precautions, the ordering constraints placed on *msgD* (*msgA* to *msgD* and *msgB* to *msgD*) would allow it to creep into the combined fragment *strict*, sharing a communication line with *msgC*.

lifelines with. That does not, however, stop it from sharing a communication line with *msgC*, effectively allowing it to creep into a fragment it has no business being in.

Before we solve this problem it helps to distinguish two kinds of nodes in the element ordering graph. The *lowermost nodes* of a fragment are nodes that represent those of its messages that have no successors in the ordering graph that represent messages of the same fragment. In the example, the node that represents *msgC* is a lowermost node since its only successor in the graph—the node that represents *msgE*—is not contained in the fragment. However large the fragment becomes, one of its lowermost nodes will always mark its bottom boundary.

The second concept we need is that of *fragment successor nodes*: nodes that represent messages not part of a given fragment, but that have at least one predecessor which is. In the example, both *msgD* and *msgE* are fragment successor nodes, but *msgF* is not.

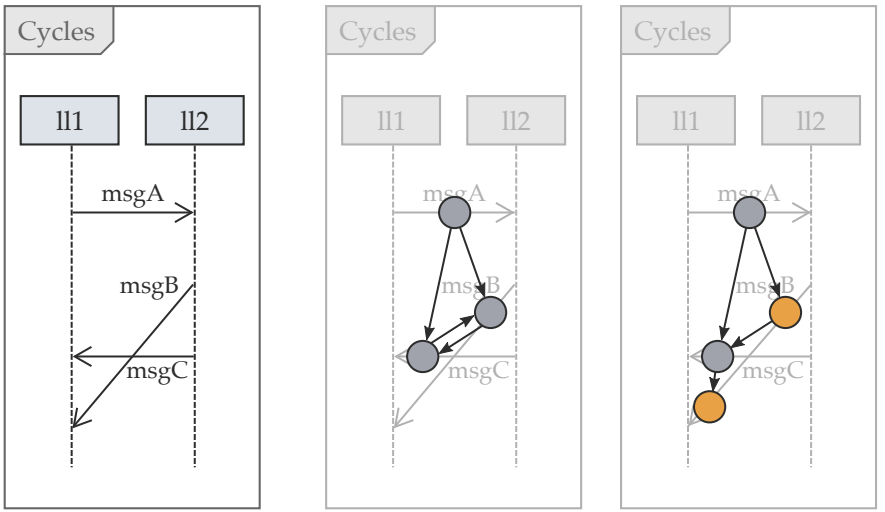
To finally solve our problem, we introduce additional ordering constraints from all lowermost nodes to all fragment successor nodes. In other words, we force the algorithm to keep the messages that follow the fragment from creeping past those messages that will determine the fragment's lower boundary later.

Similar problems arise above a fragment. First, the *fragment predecessor nodes* must be kept from creeping past the fragment's upper boundary. And second, the fragment's *uppermost nodes* must not enter the fragment's header area. Both problems are solved by introducing additional successor constraints from the predecessor nodes to the dummy node that represents the fragment's header area, and from said dummy node to the fragment's uppermost nodes.

4.2.3 Phase 3: Cycle Breaking

The element ordering graph created by the preceding phase captures message ordering constraints that need to be met once messages are assigned to communication lines. For this to be possible, the graph needs to be acyclic. At first, this does not seem to be a problem, but there are valid sequence diagrams that violate this requirement, such as the one in Figure 4.8a, which results in the element ordering graph shown in Figure 4.8b. While it certainly seems debatable whether such a diagram makes sense, the fact is that such states can arise while users are in the process of building a sequence diagram, and thus need to be supported by the layout algorithm.

ELK Layered handles cyclic graphs by reversing edges until there are no cycles left (see Section 3.1.1). In the final diagram, the reversed edges will run against the prevalent layout direction. This solution does not work for ELK Sequence: since the edges represent message ordering constraints that need to be adhered to, they cannot simply be reversed. Instead, ELK Sequence splits one of the nodes on the cycle, as shown in Figure 4.8c.



(a) Sequence diagram **(b)** Cyclic graph **(c)** Acyclic graph.

Figure 4.8. The algorithm needs to be able to handle diagrams with cyclic element ordering graphs. **(a)** A sequence diagram with non-horizontal messages, inspired by the UML specification [Obj17, Figure 17.3]. **(b)** Without processing, *msgB* causes the graph to be cyclic. **(c)** The node that represents *msgB* is split into two nodes to remove the cycle, each representing one of the message's end points.

Since most messages will end up being drawn horizontally, each node in the element ordering graph usually represents both the start and the end point of its corresponding message. Its incident edges consequently represent ordering constraints involving both the source or the target lifelines. This is the cause for cycles in the first place. Splitting a node makes the ordering constraints independent from one another, making it impossible for cycles to appear.

4.2.4 Phase 4: Communication Line Assignment

With the first three phases completed, we now have an acyclic element ordering graph that we can use to assign the graph's nodes to communication lines. To do so, we use the network simplex algorithm, one of the algorithms available to ELK Layered to compute a layer assignment for a layered graph (see Section 3.1.2). The algorithm aims to minimize the length of edges, effectively producing a result that tends to keep messages close together at each lifeline.

Similar to Phase 2, the fourth phase also has to solve an additional problem if vertical compaction is switched on. Phase 2 introduced constraints that keep messages from entering fragments off limits to them, but it was only able to catch those messages that it knew for a fact would have to precede or follow a fragment due to constraints involving the fragment's messages. It could not, however, prevent completely unrelated messages from wreaking havoc with the diagram's semantics.

Consider, for example, the sequence diagram in Figure 4.9. It has five lifelines of which *ll2* and *ll3* have no relation at all to the others, particularly not to anything contained in the combined fragment. Figure 4.10a shows the original assignment of the element ordering graph's nodes to communication lines as produced by Phase 4, including the nodes added for the headers of the two combined fragments, *strict* and *opt* (shown in a darker color). While *msgE* does not cause a problem since it is part of a successor constraint involving *msgD*, Phase 2 completely missed the fact that *msgF* and *msgG* must not be moved into the *strict* fragment. Phase 4 thus produces an assignment that would lead to an overlap of the two fragments (see Figure 4.10b) unless we take additional steps to prevent that from happening.

Note how even if Phase 2 had noticed the problems, it could not have determined which additional constraints to add: should *msgF* and *msgG* be placed above or below the *strict* fragment? Such decisions can only be made once messages have been assigned to communication lines and the nature of possible overlaps can be detected.

We deal with this problem by post-processing the communication line assignment as follows. For each combined fragment, we compute the lifelines

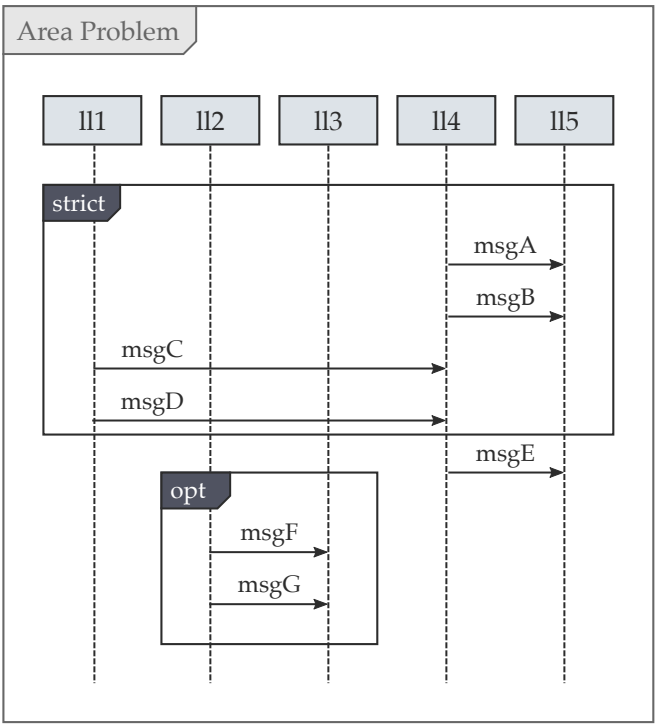
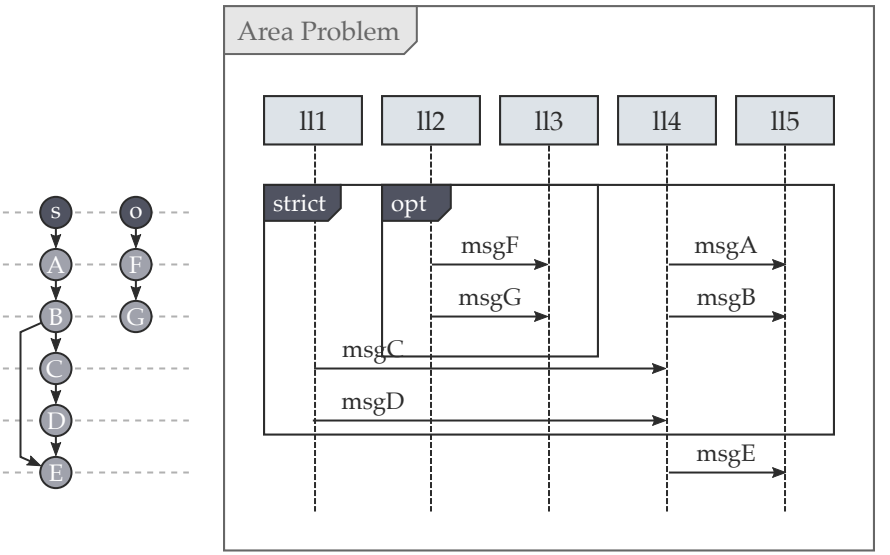


Figure 4.9. A sequence diagram with five lifelines, of which two (*ll2* and *ll3*) bear no relation to the others.

it spans by looking for the leftmost and for the rightmost lifeline incident to one of the fragment's messages. The big fragment in Figure 4.9, for example, spans all five lifelines while the smaller fragment only spans two. We then iterate over all communication lines, keeping a list of fragments for each lifeline that are currently open (or *active*) there. For each communication line, we perform four steps of computation.



(a) Assignment **(b)** Wrong drawing

Figure 4.10. The naive communication line assignment produced by Phase 4 for the diagram shown in Figure 4.9. Blindly following the assignment leads to a wrong drawing.

Step 1

We iterate over the communication line's nodes, looking for nodes that represent the header of a combined fragment. Let x be such a node and f_x be the fragment it represents. The fragment can begin at the current communication line if there is no other fragment f_y for which all of the following conditions are true:

- f_y has already been marked as being active.
- The sets of lifelines spanned by f_x and f_y have a non-empty intersection.
- f_x is not contained in f_y and f_y is not contained in f_x (otherwise, it would be perfectly fine and even necessary for them to overlap).

If we find no such fragment f_y, f_x can begin at the current communication line and is thus marked as being active at all lifelines it spans.

Taking the ordering graph from Figure 4.10a as an example, we would first encounter one of the nodes that represent either the *strict* or the *opt* fragment; let us assume that we deal with the *strict* fragment first, which causes us to mark that fragment as being active at all lifelines. For the node the represents the *opt* fragment, we would detect a conflict with the larger one at lifelines *ll2* and *ll3* and thus refrain from activating it.

Step 2

We iterate over the communication line's nodes again, building an initially empty list of nodes that will have to be moved to the next communication line because they are in conflict with active fragments. If a node represents the header of a fragment that has not been marked active by the previous step, it is added to our list. If a node represents a message that would cross an active fragment it is not part of, that too is added to our list.

In our example graph, the *opt* fragment's header node would be the only one added to the list while processing the first communication line.

Step 3

Each node in the list produced by the previous step is moved to the next communication line. If it has successors in the element ordering graph, that may of course invalidate the communication line assignment by placing two nodes with ordering constraints in the same communication line. We thus allow the movement to propagate through the following communication lines to restore the assignment's validity.

In our example, we would move the *opt* fragment's header node to the second communication line. Since it now shares a communication line with the node representing *msgF*, we would move that to the third line, and move *msgG* to the fourth line for a similar reason.

Step 4

Finally, we look for active fragments that need to be deactivated. A fragment can cease to be active once we have encountered all of its bottommost nodes.

In the example, the first time this happens is when we encounter the node that represents *msgD* on communication line five. This is when we mark the *strict* fragment as not being active anymore, thereby allowing the *opt* fragment to become active once we process the next communication line.

The result of applying Phase 4 to our example graph produces a result exactly like the one shown in Figure 4.6.

4.2.5 Phase 5: Coordinate Assignment

With a lifeline order having been decided upon and communication lines having been computed, the fifth and final phase is in charge of computing actual coordinates for all elements of the sequence diagram. This is rather straightforward, particularly in comparison to some implementations of the preceding phases. The result for our example diagram is exactly what we already saw in Figure 4.3.

The bulk of the coordinate assignment process can be divided into two stages: the *vertical sweep* and the *horizontal sweep*.

The vertical sweep iterates over the communication lines computed during the preceding phase and is only interested in those of the nodes of the element ordering graph that represent messages. At first glance it seems that we can simply assign all of the line's messages a y coordinate that is a configurable amount of space below the preceding line's messages. In doing so, however, we might end up with overlaps between messages with a non-empty intersection of the sets of lifelines they span: the element ordering graph only captures vertical successor relationships, not conflicts due to overlaps. The solution is to introduce the possibility for a communication line to split into several lines to avoid message overlaps, but trying to get away with as few lines as possible. This is a problem similar to Problem 3.17, and we solve it with a similar greedy algorithm.

The horizontal sweep primarily iterates over the lifelines and assigns x positions to them. This is a straightforward process, yet one needs to be sure to reserve enough space for lost and found messages, execution specifications, message labels, and comments attached to messages or lifelines. The x positions of the lifelines can not always be directly applied to their incident messages due to the possible presence of executions. After placing these, x coordinates of incident messages need to be adjusted.

At this point we know how far to the right the lifelines extend. Any non-specific comments can now be placed in a column to the right of the rightmost lifeline. Also, coordinates of combined fragments can be set according to the messages they contain.

Note that in placing message labels and comments, the fifth phase addresses the **P-PLACEMENT** principle..

4.3 The KieSL Language

To test the algorithm as well as the concepts to be introduced in Chapter 6 we have developed the sequence diagram editor shown back in Figure 1.9. It makes use of the same technologies as the SCCharts editor: an Xtext-based language called KieSL along with the text editor that comes with it, and a KLighD-based transient visualization that builds upon ELK Sequence to compute layouts [Jah15].

The design of KieSL was guided by the following principles that distinguish it from existing languages:

- Lifelines should be explicitly declared near the top of the document. Other textual sequence diagram editors can create lifelines on the fly as they are first referenced by messages, but that can lead to problems. First, properly readable lifeline labels rarely lend themselves well to double as succinct identifiers. And second, it seems sensible to start a sequence diagram by thinking about the participants relevant to the interaction.

- Message types should be explicitly named. Other tools use different symbols to distinguish different types of messages. WebSequenceDiagrams, for example, uses `->`, `->>`, and `->*` to denote synchronous, asynchronous,

```
interaction "Graph"

lifeline "ll1" as ll1
lifeline "ll2" as ll2

fragment strict {
    ll1 sync "msgA" to ll2

    fragment opt {
        ll1 sync "msgB" to ll2
    }
}

ll1 sync "msgC" to ll2
    sourceStartExec
    sourceNote "A comment."
ll1 async "msgD" to ll2
    sourceStartExec
ll2 response "msgE" to ll1
    targetEndExec all
```

Figure 4.11. A textual representation of the sequence diagram shown in Figure 4.2a.

and create messages, respectively. Making sense of the textual representation requires knowing about these conventions, which KieSL aims to avoid.

A KieSL document starts with an optional declaration of the interaction itself, which, if omitted, makes the interaction's title and surrounding rectangle disappear in the drawing. Next, lifelines can be declared and named, followed by all elements the sequence diagram should contain.

Instead of supplying the full KieSL grammar, Figure 4.11 shows a textual representation of the sequence diagram from Figure 4.2a. The example shows many of the language's concepts and features.

The KLighD synthesis responsible for translating KieSL models into KLighD graphs with rendering information simply creates the graph structure ex-

pected by ELK Sequence. It presents users with several options to customize the visualization:

- A choice between different rendering styles, some more suitable for printing and others for viewing diagrams on screen. The synthesis is designed in a way to make adding further rendering styles easy.

- Direct access to some of the algorithm's options, such as the lifeline sorting or label placement strategies.

4.4 Evaluation

I evaluated the layout algorithm with an aesthetics-based experiment. To do so, however, there was the seemingly mundane problem to be solved of finding sequence diagrams as close to real-world examples as possible. Saving the day were Regina Helbig and her colleagues, who published not just a paper on mining GitHub repositories for UML models [HQC+16], but also the data they collected.[1] I was able to find 50 sequence diagrams among the first 5,000 entries of their comprehensive list of UML files, which I recreated using KieSL and then used for the evaluation. The sequence diagrams averaged 6.06 lifelines (for a total of 303) and 16.54 messages (827).

The collected data and any scripts used to conduct the subsequent analysis are available online.[2] Samples of the diagrams used for the evaluation are shown in Section A.2.

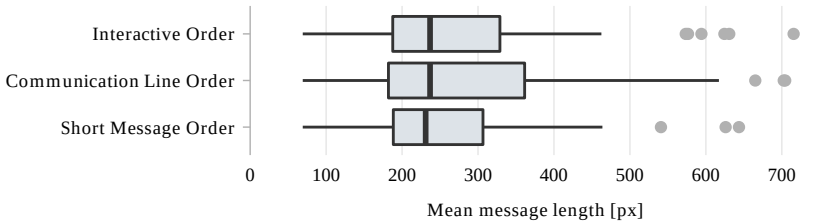
I wanted to answer the following questions:

1. Which effects do the lifeline ordering strategies have on the length of messages and their crossings with lifelines?

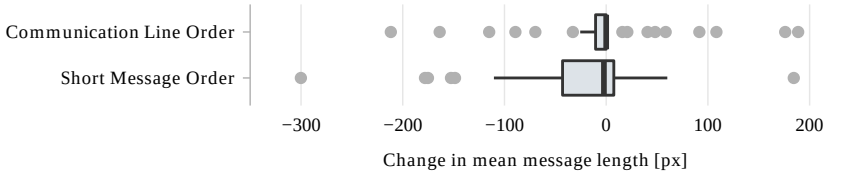
2. Which effect does vertical compaction have on the height of sequence diagrams? The answer to this question affects the **P-SIZE** principle.

[1] http://oss.models-db.com

[2] https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-1804-data.zip



(a) Mean message length (lower values are better)



(b) Change in mean message length compared to interactive order

Figure 4.12. The mean message length produced by different lifeline ordering strategies for each sequence diagram, both overall and expressed as the change compared to the interactive lifeline order.

4.4.1 Lifeline Ordering Strategies

Figure 4.12a shows the mean message length in each diagram produced by the three lifeline ordering strategies. The results look rather similar, so Figure 4.12b shows the effect of the non-interactive strategies on message length compared to the original ordering as determined by the diagram's designer. As expected, short message order will usually decrease the mean edge length in a diagram, which is true for 50% of all diagrams in the test set, as Table 4.2 shows.

Figure 4.13a shows the number of crossings between messages and lifelines produced by the three strategies in each diagram. The way we counted crossings, neither the source nor the target lifeline of a message contributed to the total. In this metric, short message order has a considerably more pronounced effect, a conclusion confirmed by Figure 4.13b,

Table 4.2. Percentage of diagrams whose message length or number of message-lifeline crossings have decreased, remained unchanged, or have increased subject to the lifeline ordering algorithm compared to the interactive lifeline order.

	Communication line order	Short message order
Mean message length		
Decreased	32	50
Unchanged	50	24
Increased	18	26
Message-lifeline crossings		
Decreased	32	50
Unchanged	50	24
Increased	18	26

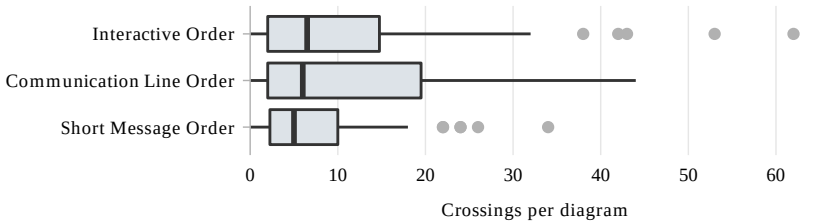
which shows the number of crossings relative to the original ordering. As Table 4.2 shows, short message ordering reduces the number of crossings in half of all diagrams.

4.4.2 Vertical Compaction

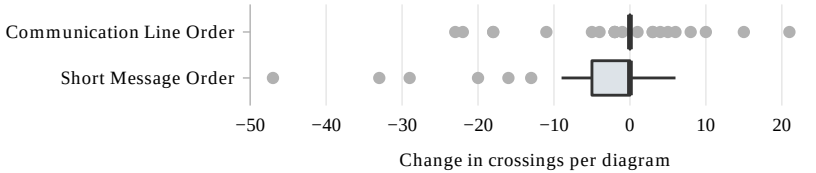
Let us turn to the second question: the influence of vertical compaction on diagram height. Figure 4.14a shows the height of diagrams with and without vertical compaction. Both results are virtually identical, suggesting that vertical compaction does not have much of an effect. Indeed, closer inspection revealed that it affected only 18% of diagrams. If we limit our evaluation on those, we get the data shown in Figure 4.14b. If vertical compaction has a chance to do something, its influence is noticeable, as Figure 4.14c shows.

4.4.3 Discussion

It is hard to give final advice on how to configure ELK Sequence. As is often the case, the question of what is the "best" configuration depends on



(a) Message-lifeline crossings (lower values are better)

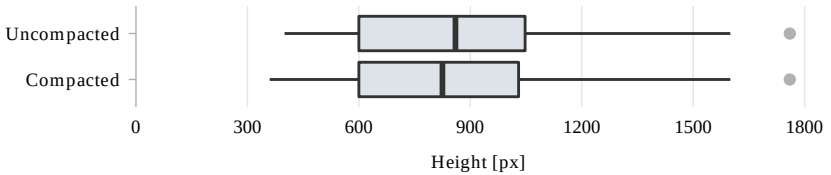


(b) Change in message-lifeline crossings compared to interactive order

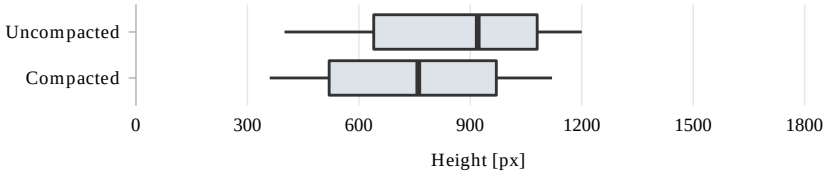
Figure 4.13. The sum of message-lifeline crossings produced by different lifeline ordering strategies for each sequence diagram, both as absolute values and as the change compared to the interactive lifeline order.

the situation. If one associates certain semantics with the order of lifelines, going for the short message order strategy might not be a good idea. The communication line strategy aims at finding a helpful order and will probably be the best choice if one does not yet have an idea of what the final diagram should look like, as is perhaps the case while the user is still in the process of creating it. Since users may be irritated if the order of lifelines changes while editing, a sensible approach may be to keep the order they chose until they request it to be optimized.

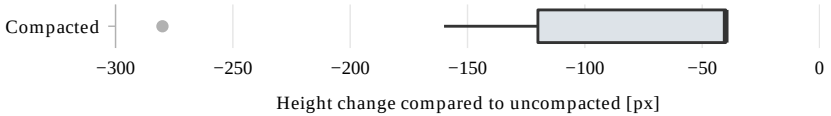
Vertical compaction seems less ambivalent. While this may be different for other diagrams, it never had a negative effect on the diagrams in our test set and proved a valid approach that conforms to the **P-SIZE** principle. This suggests that leaving it turned on will usually not be harmful.



(a) All diagrams (lower values are better)



(b) Affected diagrams only (lower values are better)



(c) Height change in affected diagrams

Figure 4.14. The height of drawings with and without vertical compaction, both for all diagrams and limited to those diagrams affected by compaction.

In the first part of this thesis, we have concerned ourselves with the challenges text poses for automatic layout algorithms. Starting with the next chapter, we will look at opportunities that arise from the availability of automatic layout, which, of course, will give rise to a whole new set of problems to be solved.

Part II

Reaping the Rewards

Layout Pragmatics

Chapter 5

Comment Attachment

In this chapter, we will have a go at the **P-NOTATION** *principle by finding ways for layout algorithms to preserve certain aspects of secondary notation. In particular, we will concentrate on comments whose relations to diagram elements are expressed implicitly through placement and content. To preserve these cues during automatic layout, we need to find ways to recognize them and make them explicit.*

The availability of automatic layout algorithms gives us opportunities for new ways of interacting with diagrams. One example is the KIELER Ptolemy Browser that we already encountered in Section 1.1.1: instead of having to open a separate window for each sub-model, the browser allows users to browse them in-place, enlarging hierarchical nodes enough to display their child graphs. This is only possible through automatic layout, but Ptolemy II models contain a feature that keeps us from simply running ELK Layered and be done with it.

Just like textual programming languages, Ptolemy II allows developers to add comments to what they produce. Originally intended to help people understand code, comments have since become the subject of occasionally intense debates among developers as to when they are useful, how they should be written, and even whether they should be necessary in the first place. We do not intend to concern ourselves with such questions here. Instead, we shall look at how comments make things more difficult for those bent on using automatic layout with Ptolemy II models.

Our first point of contact with Ptolemy II models was Figure 1.2, back in Chapter 1. As we already noted back then, the authors of the model shown there went to great lengths to properly explain how things work, and it is absolutely clear to us which comment relates to which node largely due to their placement. Running ELK Layered on that diagram, however, yields the

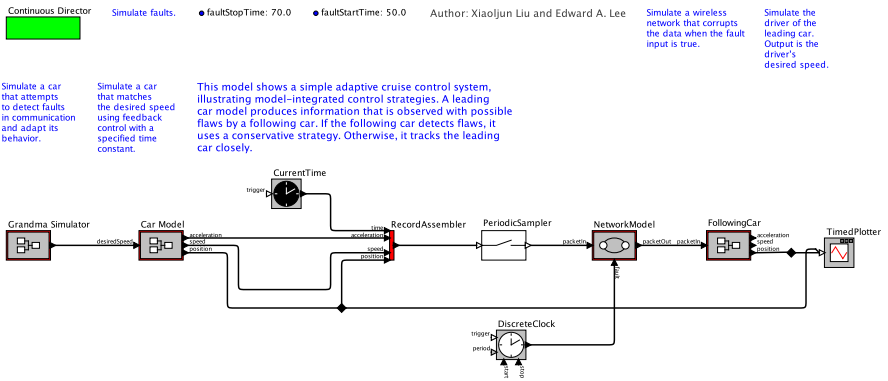


Figure 5.1. The Ptolemy II block diagram from Figure 1.2 laid out without paying special attention to comments. Note how it is not clear anymore which comment relates to which node.

result shown in Figure 5.1. It is not at all clear anymore which comment relates to which node because ELK Layered has not preserved the spatial cues which made the relations obvious in the original diagram. This is because spatial cues are implicit: while they may appear very clear to users, they are not an explicit part of the model, but an emerging property of the placement. As a consequence, ELK Layered is not aware of such information. While a layout adjustment algorithm might have been successful in preserving the spatial cues without actually being aware of their existence, a layout creation algorithm such as ELK Layered has virtually no chance of doing so. Since our goal is to use ELK Layered with Ptolemy II models, we will have to make implicit relations between comments and whatever they relate to, if anything, explicit in order to have the algorithm take them into account. We will call the task of doing so the *comment attachment problem* since our goal is to explicitly *attach* comments to whatever diagram element they relate to (we will give a more precise definition later).

Getting problems like this right can be a deciding factor for users to keep or to stop using automatic layout. Who, after all, would use a feature that did its best to mess up the placement of comments? Indeed, we know

of a developer of commercial software who hides comments completely instead of running the risk of having them end up in the wrong places.

Looking beyond the immediate problem, if we can find a way to infer the relations between comments and nodes, we can start exploiting those relations. Tan et al. [TMT+12], for example, show how to automatically find inconsistencies between Javadoc comments and code, something that necessarily relies on knowing what element a comment relates to, which is easy in the Java world. Going back to our browsing scenario, comment attachment could be used to only ever show the subset of comments relevant to the current selection of nodes, either in the diagram or in a separate information view; we will revisit this in Section 6.4.1.

For the rest of this chapter, we will investigate a bit more how comments relate to secondary notation before diving into different types of comments and how we might go about solving the comment attachment problem.

5.1 Comments and Secondary Notation

Green and Petre [GP96] define secondary notation as using "layout, colour, other cues to convey extra meaning, above and beyond the 'official' semantics of the language."

In visual languages used through drag and drop editors, layout seems to be an especially powerful tool. An obvious example is laying out car braking subsystems such that they correspond to the positions of the wheels they control. Layout can make or break a diagram, and novice users of visual languages are particularly prone to producing diagrams that violate unwritten rules advanced users employ to produce and understand diagrams [Pet95]. In our context, this immediately raises the question of how automatic layout algorithms fare in this regard. While they are usually designed to optimize certain aesthetic criteria, little research has gone into the usability of the results [PCA02].

If a visual language provides comments, they do not seem to fit the definition of secondary notation—after all, they *are* part of the language's syntax. Where they do fit into secondary notation is when it comes to judging which diagram element they relate to. Some languages support

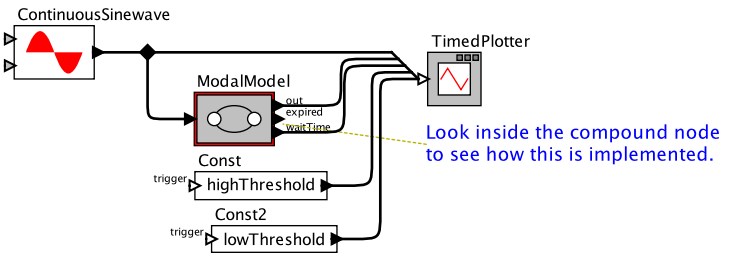


Figure 5.2. A Ptolemy II model with a comment that is explicitly attached to the actor it relates to. Even though it is closer to and better aligned with the *TimedPlotter*, it is perfectly clear that it in fact relates to the *ModalModel*.

explicit attachments, usually in the form of lines that explicitly connect comments to what they describe. Explicit attachments are unambiguous to users and are part of a model's definition, which makes them easier to handle for layout creation algorithms. The comment in Figure 5.2 is an example of what explicit attachments look like in Ptolemy II, which introduced the feature some time ago. Many existing Ptolemy II models do not make use of them, however, either because they were created before explicit attachments were available or because their authors simply did not care to use them.

In the absence of explicit attachments users have to rely on implicit cues to understand what element a comment relates to. This is where secondary notation enters both the proverbial and the literal picture: placement, formatting, and the comment's text may all contribute to a user's understanding of what the comment describes. We will examine possible implicit cues more closely in Section 5.3. Implicit cues are by no means always unambiguous. Bad placement decisions may suggest misleading relations that even the comment's text may not be able to compensate for. Petre's distinction between novices and professionals suggests that the two groups may differ in the effectiveness of their comment placement.

Note that the distinction between explicit and implicit attachments translates well to textual languages. Taking Java as an example, it is always clear which semantic element a Javadoc comment relates to if that comment

Table 5.1. We distinguish different types of comments. Specific comments have an attachment while non-specific ones do not. This chapter concentrates on specific node comments and non-specific comments.

Comments	
Specific	Non-specific
Node	Title
Port	Author
Edge	General

follows specifications. In that sense, Javadoc comments may well be viewed as always being explicitly attached. Java's other comments have no such explicit attachment, but it is usually very clear which part of the code they relate to. This may to be due to the fact that Java code is read top-to-bottom and left-to-right, implying that a comment will always be interpreted to relate to whatever it precedes vertically or succeeds horizontally. The ambiguity in visual languages may stem from the fact that there is no such natural reading order. For any given comment, the element it relates to could conceivably be found in pretty much any direction.

For the purposes of this discussion, we will divide comments into two basic types (Table 5.1). *Specific comments* relate to a small number of diagram elements, usually one, and can be further distinguished according to the types of those elements. *Non-specific comments* do not relate to specific elements. We distinguish between *title comments*, which add a title to a diagram much like to a book chapter, *author comments*, which, like `@author` tags in Javadoc comments, add information about who produced a diagram, and *general comments*, which add other kinds of information to the diagram.

5.2 The Comment Attachment Pipeline

Starting with this section, we will discuss how to infer implicit comment attachments and thus make them explicit for layout algorithms to make use of. We will limit ourselves to non-specific comments and node comments, since ELK does not support links between comments and ports or comments

and edges yet. This is left for future work, but many of the ideas discussed henceforth should be applicable there as well.

Before we go any further, however, we should define the comment attachment problem.

Definition 5.1 (Comment attachment problem)**.** Let $G = (V, E)$ be a graph and C be a finite set of comments. We call a partial function $\text{att} : C \to V$ that defines which node a comment relates to a *comment attachment function* and write $\text{att}(c) = \perp$ if it is not defined for a comment $c \in C$. The problem of finding such a function is called the *comment attachment problem*. □

The quality of a comment attachment function computed for a given diagram must be judged by how closely it matches a user's understanding. It is not immediately obvious how best to compute functions that fare well in this regard. A first clue, however, comes from our classification of comments into different types. Non-specific comments do not relate to any specific diagram element. For any such comment c, it should thus hold that $\text{att}(c) = \perp$. For all other comments, it may not always be clear which element they relate to, so it seems justified to try and find heuristics that are good predictors as to how likely it is for a given node and comment to be related.

These thoughts lead us to the pipeline shown in Figure 5.3. Each comment is first put through any number of *attachment filters* that determine whether the comment is likely to be non-specific. If so, it is immediately discarded and thus left unattached. Otherwise, it is put through any number of *attachment matchers*, together with all of the diagram's nodes. For each node, the matchers compute a probability for the two to be related according to different heuristics. The results are then fed into an *attachment decider*, which looks at the probabilities and either attaches the comment to a node or not.

The pipeline takes care of making implicit attachments explicit, but what if a diagram already contains explicit attachments, as some Ptolemy II models do? In that case, it seems like a good idea to disable comment attachment. After all, if a developer already went to the trouble of defining explicit attachments for at least one comment, chances are that leaving other comments unattached was a deliberate decision.

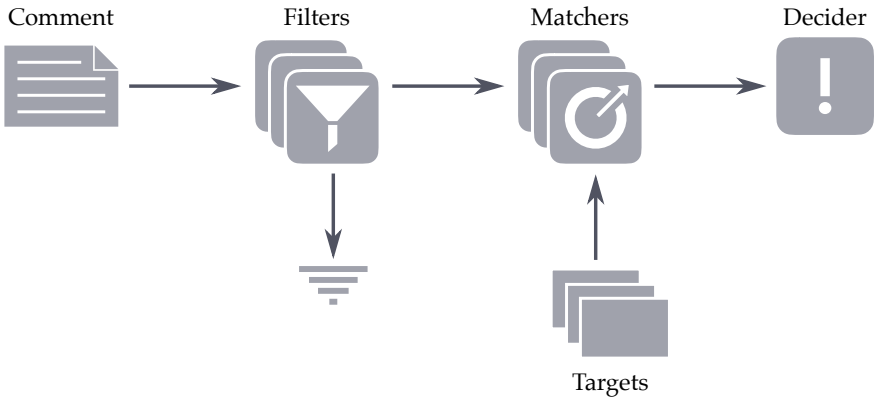


Figure 5.3. The comment attachment pipeline that comments go through. All of the labeled components can be customized by clients.

5.3 Measuring Attachment

The success of the comment attachment pipeline relies on good filtering and matching heuristics. A promising approach to find such heuristics is analyzing how developers use comments, which requires us to find a data set to analyze.

Ptolemy II ships with a set of demo models intended to showcase different models of computation, actors, and development techniques. 348 of them, created by 40 different developers, served as the main data set for the analyses. All of these models are mostly based on data flow diagrams, but some contain small submodels that are state machines. However, since the heuristics to be discussed are not tied to any particular type of diagram or layout style, they should be applicable to state machines as well. Overall, the models contained 7,447 nodes, with a mean of 21.4 nodes per model, and 1,078 comments (3.1 per model), of which 182 (about 17%) are specific comments; we will see how that last number was obtained in a moment. Each model in the data set meets all of the following criteria:

▻ The KIELER Ptolemy Browser is able to open and display it properly.

- ▻ It contains at least one comment.
- ▻ It contains no explicit attachments.
- ▻ It is clear to a human what, if anything, each comment relates to.
- ▻ All comments are either non-specific or specific to exactly one node. Note that this excludes port and edge comments, which some of the Ptolemy II models did contain, but can currently not be processed in ELK.

To analyze the usage of comments, we went through each model and manually defined a reference attachment function att_{ref} by figuring out which node, if any, each comment relates to. We then used that function to analyze the success of the heuristics below in describing comment usage.

The ideas for the heuristics themselves stem from several backgrounds. First up is our experience from looking at diagrams. Second, some heuristics, such as proximity, are rooted in Gestalt psychology [Wer23]. The alignment and font size heuristics are based on established principles in graphic design. In the following, we will introduce the basic idea of each heuristic, analyze our data set, define the heuristic based on the analysis, and evaluate how well it performs.

5.3.1 Filters

Font Size Filter

The aim of this filter is to recognize title comments to prevent them from being attached. Text documents usually start with a title set in type larger than that used for the rest of the text. Our hypothesis was that developers may follow a similar pattern, such as in Figure 5.4.

57 out of 348 demo models contain a title comment. Title comments always appear at the uppermost hierarchy level, and are never the only comment there, except for a single case. In 45 of the 57 cases, the title comment has the largest font size out of all comments, and its font size exceeds the default font size used for comments in Ptolemy II. There is one

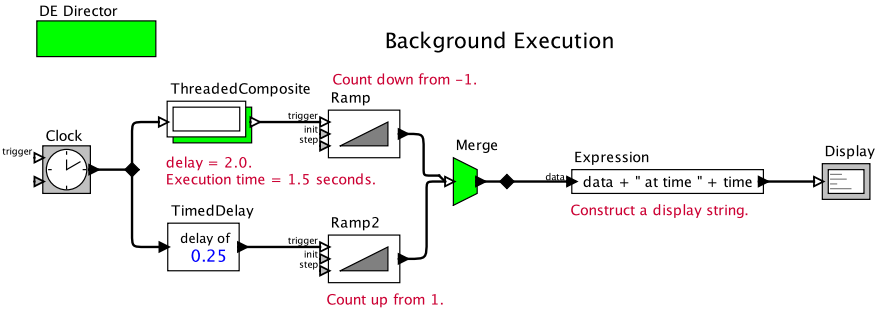


Figure 5.4. A diagram with a title comment, *Background Execution*. Also note how the bounding box of the leftmost comment is closer to the *Ramp* actor than it is to the *ThreadedComposite* actor, although their alignment makes it clear that it actually refers to the latter.

model where a comment is the only one with the largest font size, but is not a title comment.

Heuristic 5.2 (Font Size Filter)**.** Find the set of comments on the uppermost hierarchy level with the largest font size. If the set only contains a single comment, select it as the title comment and thus filter it out, provided that it is not the only comment on the uppermost hierarchy level and that its font size exceeds the default font size. □

As expected, the font size filter finds 45 out of the 57 title comments and only once generates a *false positive* (that is, filters out a comment which actually is not a title comment). It misses the remaining title comments, thus producing 12 *false negatives*, but it seems difficult to derive a simple rule that can recognize title comments based on the semantics of the text.

Text Prefix Filter

The aim of this filter is to recognize author comments and general comments to prevent them from being attached. Programs written in textual languages often contain a general description of what the program does, as well as the name of the developer who wrote the code. Visual languages usually

do not provide elaborate documentation systems such as Java's Javadoc, but it seems reasonable to hypothesize that author comments and general comments will often start with similar phrases.

The majority of models contain general comments. Many of them start with one of the following—rather sensible—prefixes: "This model," "This submodel," "This example," and "Model of." The remaining ones do not share any prefix that would work well in recognizing general comments.

Almost every model in our set contains an author comment. Except for four of them, they all start with one of three prefixes: "Author:," "Authors:," and "Demo created by."

Heuristic 5.3 (Text Prefix Filter)**.** Filter out comments that start with one of the prefixes listed above. □

Out of 1,078 comments, the text prefix filter filters out 457 comments, producing no false positives. However, it did miss 382 author and general comments.

Area Filter

The aim of this filter is to recognize general comments to prevent them from being attached. One might hypothesize that general descriptions of what a program does will often be longer—and therefore larger—than more specific comments.

On average, node comments are indeed smaller than non-specific ones (14,075 and 26,028 square pixels, respectively), but the average does not tell the whole story. Consider the histograms in Figure 5.5. They show the distribution of the area of comments first for node comments (as determined by the reference attachment) and second for those non-specific comments that would remain to be checked by the area filter once the title and text prefix filters have run. Ideally, the two histograms would differ considerably and suggest a threshold that neatly divides both sets. It would then be easy for the filter to determine whether or not to filter out a comment. However, the histograms show that the distributions have considerable overlaps, suggesting that our hypothesis will not work as well in practice as we hoped.

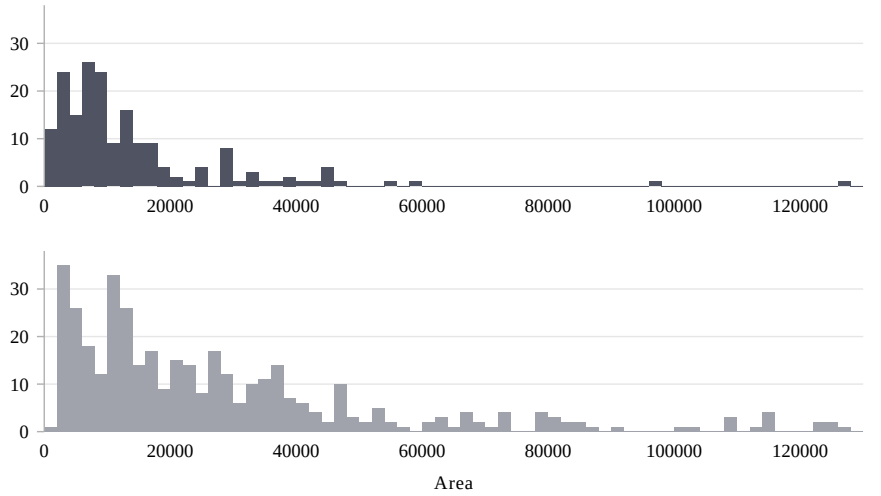


Figure 5.5. Histograms of the area in square pixels of node-specific comments (top) and of all non-specific comments not filtered out by the font size and text prefix filters (bottom). To keep the histograms readable, they are capped at an area of 130,000 square pixels, which few comments exceed.

Heuristic 5.4 (Area Filter)**.** Filter out a comment if its area exceeds a predefined threshold. □

As already assumed in the analysis, the area filter is not very effective in filtering out general comments if the amount of false positives is to be kept low. Setting the threshold too low will generate too many false positives and will thus prevent too many comments from being attached (for instance, a threshold of 14,000 square pixels will filter out about 68% of general comments, but will also incorrectly filter out 40% of node comments). Setting the threshold too high considerably limits the filter's effectiveness (a threshold of 46,000 square pixels only filters out 5% of node comments, but will only filter out 20% of general comments).

Still, with a high enough threshold, at least some general comments will be filtered out without too many false positives.

Table 5.2. The numbers of comments out of 1,078 that contain the name of a single node, along with how they are attached in the reference attachment.

	Strict		Fuzzy
	Case-Sensitive	Case-Insensitive	Case-Insensitive
Non-node comments	124	139	155
Attached to referenced node	59	59	60
Attached to different node	8	10	10
Total	191	208	225

5.3.2 Matchers

Node Reference Matcher

The aim of this heuristic is to recognize node comments and attach them to the correct node. If the name of a node appears in a comment, we call this a *node reference*. If a comment contains such a node reference, it seems sensible to assume that it should be attached to that node. This ceases to be true once further references occur in the comment: since the layout algorithm applied in our use case only allows comments to be attached to a single node, we consider such comments to be general comments.

Table 5.2 contains the results of an analysis of node references in comments. We distinguish three kinds of matchings: *strict matching* requires the node name to appear in the comment as is, either observing or disregarding case. *Fuzzy matching* relaxes this constraint: it allows arbitrary whitespace to appear between the words a node's name consists of, including the different components of "CamelCased" node names.

Note how most comments that mention a node are not actually attached to it in the reference attachment, such as the comment in Figure 5.6. These comments fall into two categories: first, general comments often mention a single node which is of particular importance to the model; and second, comments sometimes compare "this" node or model to a node mentioned by name, which it of course should not be attached to—after all, it mainly refers to "this" node or model.

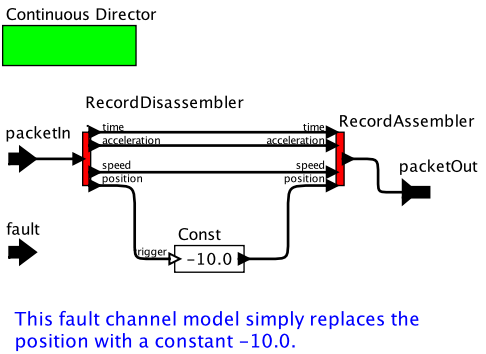


Figure 5.6. The comment in this diagram is a general comment explaining the diagram as a whole, but contains the name of the *fault* input port on the left. Such comments cause the node reference heuristic to produce false positives.

An interesting hypothesis is to assume that there is a distance threshold above which comments that reference a node are not attached to that node in the reference attachment. To test this hypothesis, Figure 5.7 shows histograms of the distance between a comment and the node it is referencing, both for comments that are indeed attached to that node in the reference attachment and for comments that, although they mention the node, are not. As it turns out, there is no specific distance threshold above or below which all references are correct according to the reference attachment. Still, most correct references seem to accumulate in the lower distances. Indeed, in about 50% of cases, comments attached to the node they mention are also closest to that node.

Heuristic 5.5 (Node Reference Heuristic)**.** If a node name appears exactly as is in the text of a comment, attach the two unless the comment contains the names of other nodes as well. Optionally refrain from attaching them if their distance exceeds a certain threshold. □

Strict, case-sensitive matching recognizes almost all attachments while producing the lowest number of false positives among the three variants. Since the heuristic will act on the filtered set of comments in practice, false positives involving comments that describe the whole diagram will

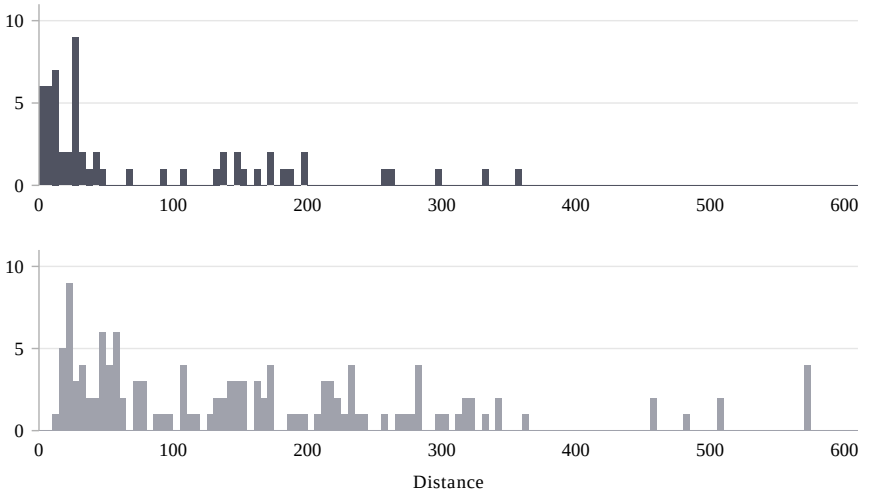


Figure 5.7. Histograms of the distance between comments and the node they mention, for cases where they are also attached to that node in the reference attachment (top) and for cases where they are not (bottom).

be reduced. However, comments that make references to "this node" and mention the name of another node are not attached correctly.

Distance Matcher

The aim of this heuristic is to recognize node comments to attach them to the correct node. The distance between a comment and a node may be the most obvious heuristic, the hypothesis being that the node a comment refers to is the one closest to it.

The mean distance between node comments and the node they are attached to, 78.48 pixels, is larger than to the node they are closest to, which clocks in at 27.42. The same is true if we compare node comments and the node they are attached to to filtered non-specific comments and the node they are closest to, which is 63.33 on average.

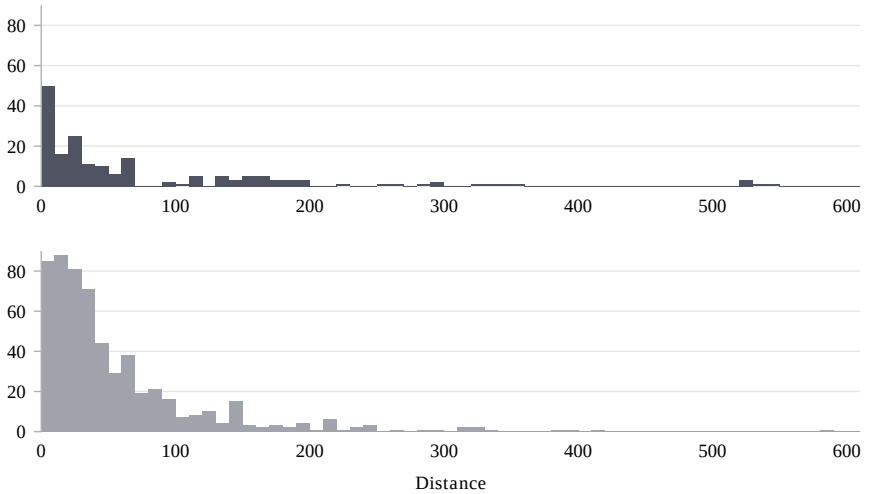


Figure 5.8. Histograms of the distance between specific comments and the node they are attached to in the reference attachment (top) and between all comments not filtered out by the title and text prefix filters and their closest node (bottom).

Out of 182 node comments, 115 (63%) are actually attached to the node closest to them. Accordingly, 963 out of 1,078 comments overall are not attached to the node closest to them.

Heuristic 5.6 (Distance Heuristic)**.** Find the node closest to a given comment. Attach them unless their distance exceeds a predefined threshold. □

As the histograms of comment-node distances in Figure 5.8 suggests, it is difficult to find a good threshold value. Setting it too low will lead the heuristic to miss a lot of attachments (a threshold of 20 will miss about 73%); setting it too high will produce a lot of false positives (a threshold of 200 will only miss about 10% of attachments, but will attach about 94% of comments that should be left unattached).

The high number of non-specific comments this heuristic would attach to nodes against their will is what originally led us to introduce attachment filters [SPH16a; SPH16b].

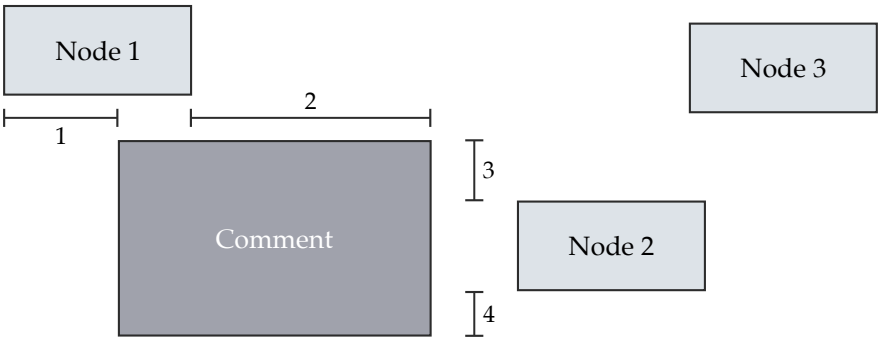


Figure 5.9. How the alignment between a comment and node is computed differs depending on where the node is in relation to the comment. The minimum distance that keeps the two from being left- (1), right- (2), top- (3), or bottom-aligned (4) is the computed alignment error. Nodes that are cater-cornered to comments, such as *Node 3*, are considered to not be aligned at all.

Alignment Matcher

The aim of this heuristic is to recognize node comments to attach them to the correct node. In graphic design, alignment between elements is used as a means to establish relationships between them. It seems reasonable to assume that comments are aligned to the node they should be attached to, as was the case back in Figure 5.4.

The bounding box of a node can be left-, right-, top-, or bottom-aligned to the bounding box of a comment. Whatever it is, a certain distance will usually keep it from being perfectly aligned (see Figure 5.9). It is the smallest such distance over the four possible alignments that we define as the *alignment error* for a given comment-node pair. If a node is cater-cornered to a comment, we consider the two to not be aligned at all.

The histograms of alignment errors in Figure 5.10 looks very similar to the distance histograms in Figure 5.8. 82 out of 182 node comments (45%) are attached to their best-aligned node. The mean best alignment value between a node comment and any node is much better than the mean alignment between the comment and its attached node (8.83 versus 27.03 pixels). This

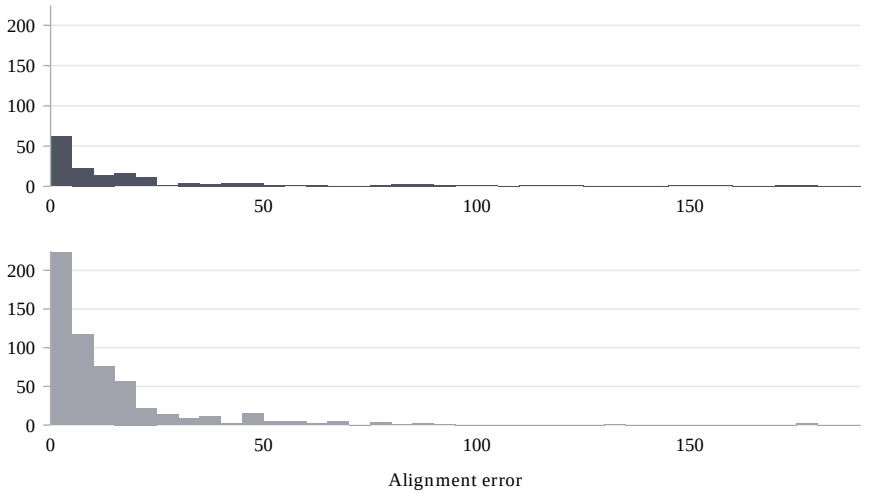


Figure 5.10. Histograms of the alignment errors (in pixels) between specific comments and the node they are attached to (top) and between all comments not filtered out by the title and text prefix filters and their best-aligned node (bottom).

makes sense, since we have not imposed a limit on their distance: it is likely that for any given comment we will find a well-aligned node somewhere in the diagram that it does not necessarily have any relation to.

Heuristic 5.7 (Alignment Heuristic)**.** For a given comment, find the node best aligned to it, optionally restricted to nodes within a certain maximum distance. Attach the two unless the alignment error exceeds a predefined threshold. □

As already expected from the analysis, the alignment heuristic does not fare very well. Without any restrictions on the maximum distance allowed between comments and nodes, it finds less than 50% of correct attachments and produces a lot of false positives. Even with a distance restriction, however, we found that results do not improve much.

5.3.3 Discussion

Based on the analyses, it seems that established conventions such as a big font size or how the list of authors is to be included in a diagram work best for comment attachment. Interestingly, both of these serve to characterize non-specific comments as opposed to the relation of node comments to specific nodes. Other heuristics work to an extent (node references, distance) or have little predictive value (area, alignment). A considerable share of the information that help link comments to nodes still seems to be in a comment's text. While it may for example be placed in the rough vicinity of the node it relates to, distance alone has its limits in how successfully it predicts attachments.

There are two limiting factors to this analysis. First, the data set contains only 348 diagrams produced by just 40 authors. Also, the number of comments actually attached according to our manual attachment (182 out of 1,078, or 17%) is not that high. The second and more severe problem is that all diagrams were created as demonstration models for Ptolemy II to help explain how certain actors or models of computation are used and how to develop using Ptolemy II; the heuristics that work well for this particular set of diagrams are not guaranteed to work well for another set. Future research will have to examine other sets of diagrams, both Ptolemy II diagrams produced by other developers as well as diagrams produced using other languages.

5.4 Implementing the Pipeline

The comment attachment pipeline as well as the filters and matchers described in the previous sections have found their way into the ELK project in the form of a general framework to be introduced in this section. Figure 5.11 shows the framework's general structure. Everything revolves around the `CommentAttacher` class, which has references to implementations of different interfaces that encapsulate customizable behavior. Using the framework entails obtaining a `CommentAttacher` instance, configuring it with an appropriate set of interface implementations, and instructing it to start the attachment process. Let us look at the interfaces one by one, along with

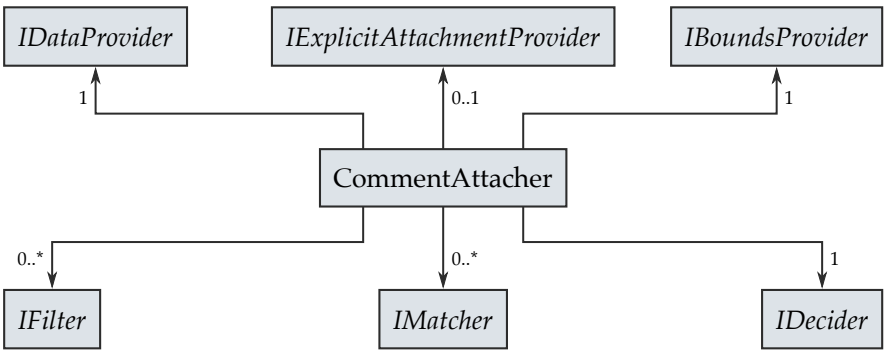


Figure 5.11. The comment attachment framework's general structure. Interfaces are set in italics.

available implementations as well as how the KIELER Ptolemy Browser uses them.

The `IDataProvider` interface decouples the framework from the actual data structure that clients want to infer comment attachments for. In fact, they do not even have to be nodes in a graph, which is why the framework uses the term *attachment target* to describe objects a comment may be attached to. The data provider provides access to comments as well as possible attachment targets, either all of them or, if implemented, a custom set of targets valid for a given comment. Custom implementations could for example include attachment targets only up to a certain distance or distinguish between different types of comments that restrict which targets they can be attached to. The data provider also provides a way to actually attach a comment to an attachment target, whatever that means in the underlying data structure. The attachment framework provides a default implementation based on the ELK graph.

An `IExplicitAttachmentProvider`, if installed, is queried for an attachment target a comment may have been explicitly attached to, provided that the visual language in question supports explicit attachments. If there is such a target, the comment will be attached to it without executing the pipeline. There is no implementation registered by default, but since

Ptolemy II supports explicit attachments we use a custom provider in the KIELER Ptolemy Browser.

`IFilter` instances implement attachment filters that decide whether or not a comment is eligible for attachment, that is, whether a comment is non-specific. If at least one filter decides that it is not, the comment will be left unattached. If no filters are registered, each comment is considered eligible for attachment, although it may still ultimately be left unattached. The framework provides two default implementations. The `SizeFilter` filters out comments that exceed a configurable area. The `TextPrefixFilter` filters out comments that either start with a configurable prefix or do not start with that prefix and can be configured to be case-sensitive or not. Both are used in the KIELER Ptolemy Browser, along with a custom filter that recognizes a title comment based on font size or on it being an instance of a special title comment type available in Ptolemy II.

Implementations of the `IMatcher` interface implement attachment matchers. Given a comment and an attachment target, they compute their assessment of how likely it is that the two should be attached on a scale of 0 (very unlikely) to 1 (very likely). The assessment of each registered heuristic is saved in a map for each pair of comment and possible target, to be acted upon later. The framework provides three default implementations, of which none are registered by default. The `DistanceMatcher` implements the distance metric and can be configured with a maximum attachment distance. The `AlignmentMatcher` implements a matcher based on the alignment of a comment with respect to an attachment target and can be configured with a maximum alignment error. The `NodeReferenceMatcher` implements a matcher based on the appearance of target names in a comment's text. It is configured with a function that provides a comment's text, a function that provides a target's name, an optional maximum attachment distance, and whether to use strict or fuzzy matching. The KIELER Ptolemy Browser uses all of these implementations except for the alignment heuristic.

Some filters and matchers rely on information about the size and position of attachment targets and comments. Retrieving these is what implementations of the `IBoundsProvider` interface do. The framework's default implementation, `ElkGraphBoundsProvider`, simply uses the information stored in each graph element. Due to differences in rendering Ptolemy II models,

however, the KIELER Ptolemy Browser usually ends up changing the size of comments while transforming the original models into its view model representation. Also, the coordinates of each element will deviate from its coordinates in the original model as soon as automatic layout is invoked. To solve this, it annotates nodes and comments with an approximation of their bounds in the original model and uses a custom bounds provider that retrieves these annotations.

With non-specific comments filtered out and attachment assessments computed, an `IDecider` decides for each remaining comment which target, if any, it should be attached to. `AggregatedMatchDecider`, the default implementation, aggregates all assessments for each possible attachment target according to a configurable aggregation function and then selects the target with the highest aggregated value. It can be configured with a lower bound on that value which, unless exceeded, will cause a comment to be left unattached. The KIELER Ptolemy Browser uses a custom attachment decider that favors the reference heuristic regardless of the values computed for other heuristics. That is, if a comment mentions a node, the two will be attached regardless of whether, for example, another node is closer to it.

There is one detail we have thus far omitted, which is preprocessing. Filtering out title comments due to font size, for example, requires knowing the maximum font size used in a diagram as well as whether there is only one comment using it or more. Information such as these need to be computed before the filter is invoked on each comment. This is why the comment attachment process starts with `CommentAttacher` allowing the registered interface implementations to preprocess the graph they will be run on once the pipeline starts executing.

5.5 Evaluation

We have looked at a number of heuristics to filter out non-specific comments and to match specific ones to nodes they relate to. Our goal was to provide information that layout algorithms could use to take secondary notation into account when it comes to comments—a task closely related to the **P-NOTATION** principle.

The heuristics were based on analyses run on a subset of the demo models that ship with Ptolemy II. What we have yet to see is whether they can be combined in a way to obtain good quality comment attachments. To evaluate that, we ran three experiments that each executed automatic comment attachment with different settings and compared the results with the reference attachment att_{ref} (refer back to Section 5.3 for how we obtained that attachment). Given a computed attachment function att_{comp}, we then classified the attachment of each comment $c \in C$ according to the following categories:

Correct An attachment is *correct* if $\text{att}_{\text{ref}}(c) = \text{att}_{\text{comp}}(c)$, that is, if a comment is attached to the same node or left unattached in both attachments.

Changed An attachment is *changed* if $\text{att}_{\text{ref}}(c) \neq \bot$, $\text{att}_{\text{comp}}(c) \neq \bot$, and $\text{att}_{\text{ref}}(c) \neq \text{att}_{\text{comp}}(c)$. That is, the comment is attached to two different nodes by the two attachment functions.

Lost An attachment is *lost* if $\text{att}_{\text{ref}}(c) \neq \bot$ and $\text{att}_{\text{comp}}(c) = \bot$, that is, the computed attachment regards the comment as non-specific although the reference attachment does not.

Spurious An attachment is *spurious* if $\text{att}_{\text{comp}}(c) \neq \bot$ and $\text{att}_{\text{ref}}(c) = \bot$, that is, the reference attachment regards the comment as non-specific but the computed attachment does not.

5.5.1 Experiment 1

Our first foray into comment attachment was based solely on the distance heuristic with a maximum attachment distance to keep non-specific comments unattached [SH14]. We ran the heuristic with different maximum distances and obtained the error rates shown in Figure 5.12. As the threshold increases, so does the error rate. Let us look at the three types of errors in turn.

The number of lost attachments decreases as the threshold increases. This is perfectly understandable: as the threshold increases, the distance heuristic finds more and more comments that are surrounded by at least one node within that distance.

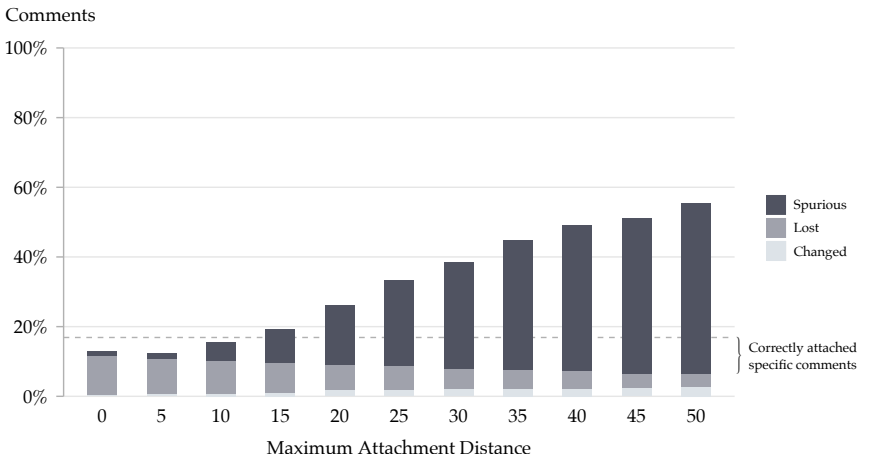


Figure 5.12. Results of experiment 1 with only the distance heuristic engaged. The graph shows the percentage of the different types of errors produced by the automatic attachment compared to the reference attachment. The horizontal line marks the percentage of specific comments according to the reference attachment. Lower values are better.

This explanation also accounts for the immense increase of spurious attachments. Non-specific comments are usually not placed far off the rest of the diagram. At some point, the distance heuristic will find a node near enough for the comment to be attached to.

Finally, changed attachments are also accounted for by a similar explanation. While they may have started out as lost attachments, many are eventually attached to a node if the threshold gets high enough. As the analysis of the distance heuristic showed, however, a substantial amount of comments relates not to the node they are closest to, but to another node further off.

Indeed if we relate the results to the percentage of comments the reference attachment considers to be specific, a considerable percentage of specific comments have either lost their attachment or have had their attach-

ment changed. Even a maximum attachment distance of 50 still leaves us with a considerable error rate.

5.5.2 Experiment 2

The most pressing problem with comment attachment based only on the distance heuristic is the sheer number of spurious attachments it yields. Reducing that is what filters are all about. Also, to counteract the fact that many comments are not in fact attached to the node they are closest to, the node reference heuristic may be helpful.

Figure 5.13 shows the results of running comment attachment with the following heuristics engaged:

- font size filter
- text prefix filter, set to match the prefixes mentioned in Section 5.3
- distance matcher, set to different maximum attachment distances
- node reference matcher, set to strict mode and to override the distance heuristic; that is, if the node reference heuristic finds an attachment, that attachment is applied regardless of what the node distance heuristic wants

As we would expect, the filters dramatically reduce the percentage of spurious attachments compared to the first experiment. The effect gets more noticeable as the maximum attachment distance increases, which makes sense.

Results for the other types of errors stay virtually the same, which is interesting since the expectation would be that the engaged text reference heuristic would actually cause a change in lost and changed attachments. The explanation might be that errors previously made by the distance heuristic are now replaced with errors made by the text reference heuristic. It can be argued that this is an improvement, though, since the nodes the comments get attached to now are at least mentioned in the text.

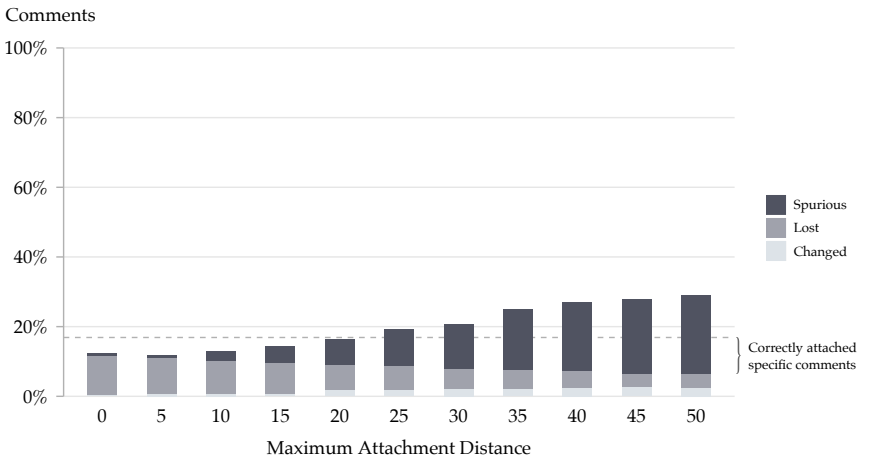


Figure 5.13. Results of experiment 2 with the node reference and distance heuristic as well as the font size and text prefix filters engaged. The horizontal line marks the percentage of specific comments according to the reference attachment. Lower values are better.

5.5.3 Experiment 3

To try and reduce the number of spurious attachments even further, the third and final experiment was based on the settings of the second experiment, but with the following modifications:

- The node reference heuristic had a maximum attachment distance of 30 applied to it.
- The area filter was engaged with a very conservative setting to filter out obvious general comments.

Figure 5.14 shows the results. We were indeed successful in that there are even fewer spurious attachments now compared to the second experiment. An interesting shift has happened concerning the other two types of errors: the number of changed attachments has decreased slightly while the number of lost attachments saw a corresponding increase. This is the result of configuring the text reference heuristic with more conservative settings.

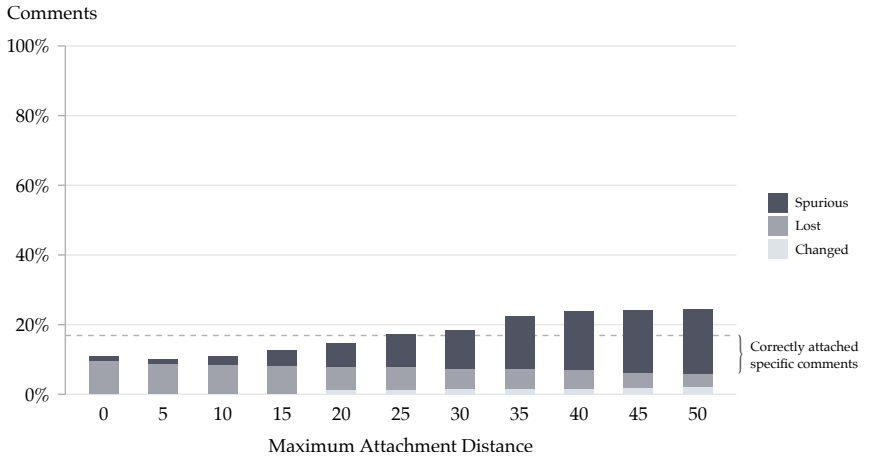


Figure 5.14. Results of experiment 3, with the same configuration as in experiment 2 but with the area filter engaged and the node reference heuristic set to a maximum attachment distance of 30. The horizontal line marks the percentage of specific comments according to the reference attachment. Lower values are better.

5.5.4 Discussion

As the evaluation shows, comment attachment can work well, with error rates as low as about 10% in our set of test models. The filtering heuristics tend to be more successful than the matching heuristics: we are better at recognizing non-specific comments than we are at finding out which node a specific comment relates to. Many of the current matching heuristics are based in some way on placement information. The analyses seemed to show, however, that a comment's text may be more important than its placement. The text prefix filter and the node reference heuristic try to make decisions based on a comment's text, but do so on a very primitive level. Parsing the semantics of the text could be a promising, albeit rather difficult venue for future research.

Overall, obtaining good results requires the involved heuristics to be configured well: finding the maximum attachment distance that works best for a given set of models, or working out the best area threshold for a

comment to be considered a general comment. This highlights a general problem with our heuristics in that if they work on our set of diagrams, that does not yet imply that they will work on another set. If there is a more general model that explains how users use and place comments, our current heuristics do not seem to be quite there yet. As it currently stands, these are issues that reduce the effectiveness of comment attachment, and providing solutions must involve obtaining more sample sets produced by more users in more visual languages.

Seeing that what we are dealing with here is the behavior of users, it might seem that a machine learning approach could be interesting to pursue. The problem here is that machine learning requires massive data sets to train the software. Even if we had a large number of diagrams, we would still have to manually produce a reference attachment function for each of them to generate proper training data.

These results may suggest that comment attachment should best be replaced by proper support for explicit attachments in visual languages. While I indeed think this to be the case, comment attachment stays relevant for browsing scenarios similar to our use case, where the underlying language either does not provide explicit attachments or users do not make use of them.

It is the latter problem that we think is most relevant to properly integrating comments into visual languages. Users tend to avoid using features that they find too cumbersome to use. As far as explicitly attaching comments to diagram elements goes, this can prevent tool developers from making more advanced features available that are based on what comments relate to, such as automatic layout, semantic reasoning, or even generating documentation automatically. It seems that the best solution may be twofold. First, provide different kinds of comments specialized on different content. General comments would contain general diagram information, author comments—which could be automatically inserted by development tools when creating new diagrams—would contain information about who developed the model, and node comments would contain additional information about a node. Dragging a node comment onto the drawing area could then include displaying "attachment lines" that indicate which node the tool will interpret the comment to relate to, thus forcing explicit attachments.

The addition of such features offers another area of application for comment attachment. Opening diagrams that do not make use of explicit attachments would trigger comment attachment and present the user with an automatically inferred attachment that they can then modify.

In this chapter, we have looked at how an opportunity enabled by automatic layout—interactively browsing hierarchical diagrams in place—required us to find ways of inferring relations between diagram elements and text that referred to them. In the next chapter, we will turn to the question of how to change that text to reduce the amount of information, the size of diagrams, or both.

Chapter 6

Label Management

In the penultimate chapter, we will concern ourselves with ways of changing the text to be displayed in a diagram. Doing so will yield solutions that observe the **P-SIZE** *principle if we do not accept losing information, and that observe both the* **P-SIZE** *and the* **P-CLUTTER** *principles if we do. I added implementations of these concepts to the ELK and KLighD frameworks as described in Section 6.2.5.*

One of the courses given at our research group deals with synchronous languages. Among other things, students learn about SCCharts and develop several of them as part of their homework using the editing environment shown back in Figure 1.6. This provided us with the perfect opportunity to carry out a first informal survey to get an idea about the usability of different aspects of the language as well as its editing environment.

The survey was distributed to 35 students as part of their homework during the semester. Among quite a few other questions about SCCharts, it contained the following question related to text in diagrams:

> Did you have problems with long labels? Describe how you solved these problems or dealt with them. Do you have any suggestions how the tooling should have helped you here?

11 out of the 35 students confirmed that they did have problems with long labels. One elaborated how long labels enlarge the diagram to the extent that it ceases to be legible and thus has no more value to them. Three students even stated that this caused them to stop using the graphical representation altogether.

For an editing concept intended to support users where purely textual editing has its limitations these results are far from good news, but the

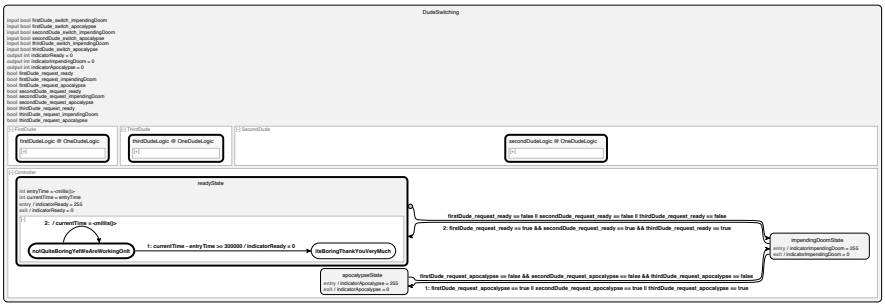
students have a valid point. Consider for example the SCChart shown in Figure 6.1a. The chart is not complex, neither in terms of the number of states nor the number of transitions. Nevertheless, the long labels cause it to be so wide that displaying it in its entirety on the page requires scaling it down to the point where its text becomes illegible. In this case, the simple act of applying line wrapping to those labels solves the problem, as Figure 6.1b demonstrates.

Diagrams too large to comfortably fit on a screen are a problem since they force users to zoom, pan, and scroll to navigate through the diagram. Cockburn et al. call this *temporal separation* and note [CKB09, Section 1]:

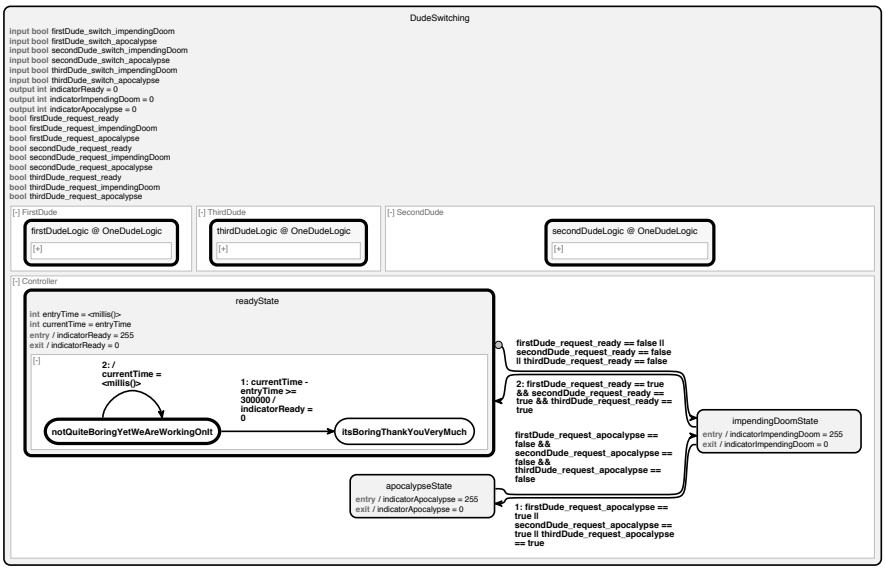
> Although scrolling and windowing are standard in almost all user interfaces, they introduce a discontinuity between the information displayed at different times and places. This discontinuity can cause cognitive and mechanical burdens for the user who must mentally assimilate the overall structure of the information space and their location within it, and manipulate controls in order to navigate through it.

Minimizing the discontinuity certainly seems like a worthy goal. Indeed, in the absence of appropriate features to manage displaying large diagrams, users seem to resort to inventing their own solutions. As one of the ETAS EHANDBOOK developers told me in personal communication, in his experience models developed using the Simulink language tend to have more levels of hierarchy compared to other visual languages simply because their nodes are larger. Developers exchange more levels of hierarchy for being able to comfortably fit each on screen.

Using terminology of the well-known Model-View-Controller (MVC) software design pattern [Ree79], introducing more levels of hierarchy because of screen space limitations is akin to applying a model-level solution to solve a view-level problem. A model's development should be guided only by the goal it is supposed to accomplish, unhindered by shortcomings of the software used to develop it. The introduction of line breaks in Figure 6.1b is an example of a view-level solution in that it changes how the model is presented to keep the presentation useful.



(a) Original labels



(b) Wrapped labels

Figure 6.1. Simply wrapping the text of long labels can already make the difference between a legible and an illegible diagram.

The size of a model is not the only potential problem. Another is the sheer amount of information displayed on the screen, which in Figure 6.1b has not changed. Today's commonly used development tools often consider all information contained in a model to be of equal importance to the developer and thus provide them with views that show the model with all of its glorious details. Referring back to Imhof's terminology introduced in Section 3.3, this can result in high label density and does not recognize that the importance of a given piece of information may not actually be set in stone. Instead, it can vary from label to label or even from time to time, depending for example on the task the user is currently trying to solve, the state the application is in, or the part of the model the user is concentrating on at a given moment. Taking advantage of this dynamic enables improved control over label density, which can be seen as one of the core functions of views as introduced by Reenskaug [Ree79]: "[The view] would ordinarily highlight certain attributes of the model and suppress others. It is thus acting as a *presentation filter*."

From this discussion, we can derive the following goals intended to improve the usability of visual languages:

Improve scaling Improve the size of diagrams to either increase the scaling at which the whole diagram can be displayed or to fit more of the diagram onto the screen at a fixed scaling. This is closely related to the **P-SIZE** principle.

Reduce visual clutter Reduce the amount of information visible on the screen by filtering information not relevant to a given situation. This is closely related to the **P-CLUTTER** principle.

Improving the scaling of a diagram can mean removing information from a label to reduce its size, but this is by no means a necessity. Figure 6.1b is a good case in point. Due to their labels, SCCharts have a tendency to grow rather wide, which makes them increasingly unsuitable to be displayed on today's computer screens or to be printed on paper. Introducing line breaks without removing information exchanges width for height, changing their aspect ratio such that they can be displayed with higher scaling.

Note that changing a label's font size, another approach that comes to mind, may or may not work with a visual language. Labels in SCCharts can

grow so wide that reducing their font size enough to have the intended effect on the diagram's width would render the text illegible. We will thus not concern ourselves with font size in this chapter. Still, a case could be made that having different fonts and font sizes available within a model could held users navigate. The particular font or font size used for an element could already hint at that element's semantics and thus provide another visual cue for users to orient themselves in the model. How this could be combined with label management is left for future work.

Neither changing the font or font size nor introducing line breaks reduces visual clutter. That goal does require removing information from labels, what is commonly referred to as *view filtering*. In the past, other methods have been proposed to reduce visual clutter, but they usually focussed on showing or hiding labels entirely, either based on the amount of space available to them [BDY06] or based on some notion of their current relevance [MJ03]. Our goal, however, allows us more freedom than just deciding between "show" or "hide." We can change a label's text to hide not all of it, but only those parts that we consider irrelevant.

We follow Fuhrmann's proposal of *label management* [Fuh11] to reach the goals we have just set:

Definition 6.1 (Label management)**.** *Label management* is the act of dynamically changing the text of labels according to rules that govern how and when to change it in order to improve the scaling of diagrams or to reduce visual clutter. □

The definition prompts three questions: how to change a label's text, when to do it, and by how much. But fear not: I will have attempted to answer all of these before this chapter is over. The *how* will be covered by introducing a number of *label management strategies* that can be applied to labels and that define what happens to a label's text. The *when* and *how much* have two aspects to them. The technical aspect concerns how label management is integrated into the view generation process. This influences when exactly labels are shortened and can provide different grounds on which to decide how much to shorten them. The User Interface (UI) aspect is about defining sensible rules and selecting appropriate strategies that will

govern label management. This is always specific to a concrete application scenario.

The basic idea of label management as well as first forays into how label texts could be changed were originally proposed by Fuhrmann [Fuh11] and lead us deep into modeling pragmatics territory [FH10]: improving the way users use and interact with visual languages to improve developer productivity. One of Fuhrmann's proposals for how to do so is *view management*, a part of which is adjusting the level of detail of each element (not just labels) to be appropriate for the current task. This is not limited to the main view, but can include auxiliary views generated on the fly with automatic layout algorithms to show a filtered and possibly highly specialized view on the model. Label management can be understood as one aspect of view management.

Throughout the rest of this chapter, we will take a closer look at label management strategies, examine how label management can be implemented and integrated into the view generation process, discuss relevant UI design issues, and finish with an evaluation. We will focus on dynamically generated KLighD views as opposed to drag and drop editing scenarios. This is because much of the power of label management lies in adapting labels to continuously changing situations, causing the layout to adjust. Being a more static editing concept, drag and drop editing does not lend itself well to this kind of scenario.

6.1 Label Management Strategies

Label management strategies define how a label's text can be changed. We can distinguish between label management strategies according to different kinds of categories. The first distinguishes between *basic label management strategies*, which operate on a label's text directly, and *composite label management strategies*, which combine existing strategies into more complex ones. The second distinguishes *lossy strategies*, which reduce the amount of information contained in a label (thus pertaining to the **P-CLUTTER** and **P-SIZE** principles since less information equals both less clutter and reduced space consumption), and *lossless strategies*, which reformat the label's text while

retaining its information (thus pertaining to the **P-SIZE** principle only in that reformatted text may change a diagram's aspect ratio, which in turn affects the scaling it can be drawn with). Since composite strategies are composed of basic ones, we will use the first distinction to guide us through the strategies we are about to discuss.

One thing to bear in mind is that many of the strategies to be introduced assume a *target width* to be provided: a maximum width a given label should not exceed in the final diagram. For the moment, we assume the target width to be magically known. We will turn to how it can actually be determined once we examine how label management can be integrated into the view generation process.

6.1.1 Basic Strategies

Basic label management strategies define direct label text transformations. We will look at two kinds of strategies. *Universal strategies* are general enough for them to be applicable to a wide range of languages, regardless of whether they employ natural-language labels or more formally structured text. That flexibility may lead them to miss opportunities specific to particular visual languages since knowledge about a label's semantics can open up possibilities for label management that go beyond the universal strategies. To illustrate this point, we will then look at a number of *SCChart-specific strategies* that exploit the specific syntax of transformation labels in SCCharts.

Universal Strategies

Fuhrmann proposed five universal label management strategies [Fuh11], illustrated in Figure 6.2 with an SCChart transition label.

The first strategy, *syntactical abbreviation*, is a lossy strategy which simply cuts the label's text off and optionally adds an ellipsis to make users aware of the fact that this has happened. By default the text is cut off once it reaches the target width, but it may also be truncated after the label's first line or after a predefined number of words. This is an easy strategy to implement and may work well for natural language, but does have

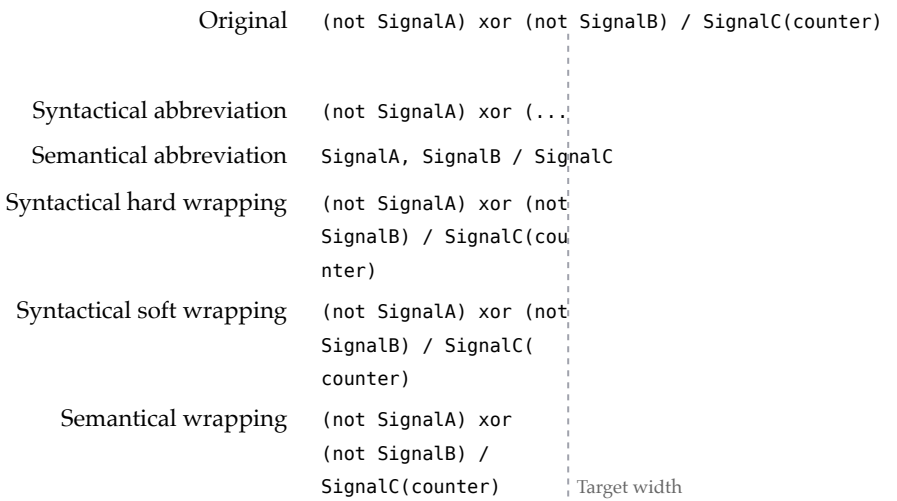


Figure 6.2. An original SCChart transition label and what the five basic universal label management strategies do to it given a target width.

shortcomings when it comes to formally structured text. In the example, the trigger contains references to three signals, `SignalA`, `SignalB`, and `SignalC`. Syntactical abbreviation removed the references to all but the first signal, which may confuse users looking for transitions involving the other two. Nevertheless, depending on the visual language this may still be a viable strategy. Regarding the two label management goals we set out to achieve, syntactical abbreviation targets both.

The second strategy, *semantical abbreviation*, was developed to solve problems such as the disappearing signal names the previous strategy produced. It uses semantic information about a label's content to reduce it to its most important elements. The aim is to give users enough information to find a particular label, which they can then inspect in more detail. In the example, the expression is abbreviated to the list of signals it references, which gives the user an idea of what it involves while hiding details about the exact expression. Note that while this strategy will usually cause labels to be shorter, mentioning all relevant elements does not guarantee that the target

width is met. Just like syntactical abbreviation, semantical abbreviation is a lossy strategy and targets both label management goals.

It could be argued that semantical abbreviation is actually not a universal strategy due to the knowledge required about a label's semantics. Nevertheless, the underlying principle of only showing the most relevant pieces of information a label contains is universally applicable. It is only the definition of what constitutes a relevant piece of information that is specific to each visual language.

The abbreviation strategies result in a smaller label size solely by manipulating a label's width. As we have already discussed, the width will often be a critical part of a label's size, whereas its height may be much less of a problem, to the point where it could even be increased. This is often the case for SCCharts, and can be exploited by inserting line wraps instead of shortening labels. Line wrapping strategies are lossless: they do not reduce the amount of information displayed on the screen, but only distribute them differently and thus only try to improve scaling.

The first line wrapping strategy, *syntactical wrapping*, inserts line wraps when a line of text is about to exceed the target width. This can be either at the exact position where this happens without regard to the text's structure (*hard syntactical wrapping*), or between any pair of tokens the text is composed of (*soft syntactical wrapping*). In the example, soft syntactical wrapping makes a lot of sense since the text is composed of tokens small enough to be rather flexible when it comes to achieving different target widths. If a label contains natural language text, a hybrid of the two, *soft syntactical hyphenated wrapping*, can be appropriate (this is not one of the original strategies mentioned by Fuhrmann). In addition to word boundaries, it also considers possible hyphenation points for inserting line wraps. Of course, this requires the language of the text to be known and a hyphenation dictionary to be installed.

In contrast to syntactical abbreviation, syntactical wrapping fares quite well with formally structured text because it does not cut any token in half. A second strategy, however, offers an interesting alternative: *semantical wrapping* restricts the token pairs between which line breaks may be inserted in an attempt to visually preserve an expression's structure. In the example, line breaks are inserted after the two binary operators: `xor`

Original	`2: max(carCount, trainCount) > 1`
Transition priorities	`2.`
Host code calls	`2: max(...) > 1`
Signal abbreviation	`2: max(car..., tra...) > 1`

Figure 6.3. An original SCChart transition label and what the three label management strategies specific to SCCharts do to it. The strategies here are not based on a target width.

and the division operator. The strategy's success depends on not being too restrictive regarding the possible line break locations in order to not exceed the target width too much.

Strategies Specific to SCCharts

We introduce three lossy SCChart-specific strategies, all of which illustrated in Figure 6.3.

The first strategy concerns transition priorities, which define an ordering among all transitions that leave a given state. If the user is only interested in the general control flow, the details of when a transition can be taken may be less interesting than their priority. That is, thus, what this strategy reduces transition labels to. Unsurprisingly, this strategy is most effective in reducing the width of transition labels.

The second strategy concerns host code calls, which are invocations of externally defined functions. Host code calls use a notation similar to that of the C programming language, with the called function's name followed by a possibly empty list of arguments surrounded by parentheses. When trying to reduce detail, the most important information contained in a host code call may not be the exact arguments, but the name of the called function and perhaps the mere presence of arguments. A call such as

```
areWobblersSynchronized(tick, wobbler1ID, wobbler2ID, 0.42)
```

can be shortened to the following:

```
areWobblersSynchronized(...)
```

This strategy targets both label management goals and can also be applied to sequence diagrams, which often include function calls.

The third strategy is based on the observation that signal names can become quite long, especially in more complex SCCharts. This complicates matters for the basic strategies, which will have a hard time to achieve good results. A possible way to solve this problem is to shorten the names of the signals themselves, both in a state's interface declaration and in all transition labels the signals appear in. This is a more radical intrusion with the potential to confuse users and should thus probably be applied conservatively. An example of where it may work is if multiple signals share the same prefix. Shortening only the prefix could still keep the signal names recognizable while keeping them shorter at the same time, although this may make it harder for users to visually scan for signal names. A simpler variant that may be less prone to this problem is to simply apply syntactic abbreviation to signal names. This strategy aims to improve scaling.

6.1.2 Composite Strategies

Some of the strategies introduced thus far only change a label's text if it exceeds the intended target width. One example is syntactical abbreviation: if the label already fits into the target width, it flat out refuses to do anything at all. Semantical abbreviation could be configured the same way, but that would only work towards the first goal, improved scaling. To reduce visual clutter, it might be helpful to define other conditions on which to shorten the label's text regardless of whether or not it exceeds the target width.

Another point is also nicely illustrated by the abbreviation strategies. Syntactical abbreviation will always meet the target width because it can simply cut off the label's text at any point of its choosing. Semantical abbreviation, on the other hand, is limited by the number and the length of the elements considered to be of importance. It has no way to guarantee to always meet the target width.

The following composite strategies are intended to address both of these observations as well as allow clients to build more complex strategies from

existing, simpler ones. We will examine how some of them can be used to integrate label management into view generation in the following sections.

The first composite strategy is the *condition strategy*, which will execute another strategy only provided that a given condition is true. If it is not, it will either leave the label's text untouched or remove it completely. Such a strategy can be used to encapsulate the logic required to check whether a label's text should be shortened or not, thereby keeping the *when* of label shortening separate from the *how* as implemented in the actual shortening strategy. In that way, the condition strategy can serve as a bridge to the application-specific parts of label management.

A special case of the condition strategy, the *type condition strategy*, will execute another strategy only on labels associated with certain types of graph elements. This way, a given strategy can be run, for example, on node and port labels, but not on edge labels.

The final composite strategy is the *list strategy*, which keeps an ordered list of shortening strategies and operates in one of two modes. The first mode simply executes all shortening strategies in order, with each strategy building upon the result of its predecessor, much like in a processing pipeline. An example would be to first execute semantic abbreviation and then apply soft word wrapping to the result, thereby increasing the chances of the label actually meeting the target width. The second mode executes the strategies in order until one of them actually bothers to do something, at which point it stops. The idea is for this mode to be used with strategies wrapped in conditions to dynamically choose from a number of possible strategies depending on arbitrary rules.

6.2 Integrating Label Management

Now that we have discussed different label management strategies, it is time to integrate them into the view generation process to provide answers to the *when* and *how much* of label management.

Recall from Figure 2.8 how the view generation process in KLighD works. Figure 6.4 abstracts away from the details a bit and broadly splits it into *view model generation* and *view model presentation*. View model generation

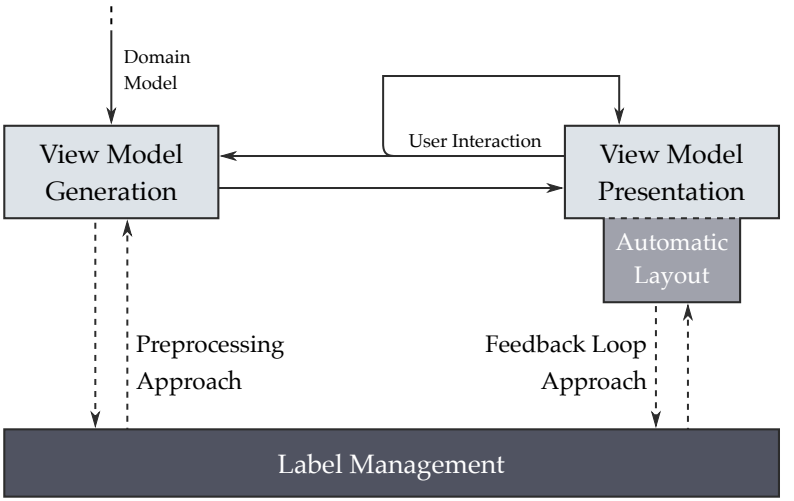


Figure 6.4. There are different approaches to integrating label management with the view generation process, differing in the amount of information they can provide label managers with for them to base decisions on.

comprises invoking a diagram synthesis on a domain model to obtain a view model that can be displayed. View model presentation then entails the loop that consists of a sequence of automatic layout invocations and user interactions. The latter can cause the whole view model to be generated afresh, but will often just change presentation properties of the existing model and does not have access to the original domain model. Whatever the case, user interaction will often require a new layout run.

As Figure 6.4 already suggests there are different points at which label management can be integrated into the whole process. They differ in the amount of information available to base label management decisions on as well as on the amount of work that has to be done to adapt label management to changing requirements.

6.2.1 The Preprocessing Approach

The first point at which label management can be invoked is during view model generation. While the view model will often represent the whole domain model, this is by no means a requirement. Instead, the diagram synthesis may well choose to apply filtering techniques to reduce the amount of detail of a domain model element's representation, or to leave it out of the view model entirely. One such filtering technique can be label management. Invoking it at this stage is what we call the *preprocessing approach* for here label management is run while the view model is being created, before it is handed off to be presented to the user and even before automatic layout is run.

At this point, the information label management decisions can be based on may for example include the following:

Preferences Most applications provide preferences for users to customize. These preferences can include settings that govern how label management works, perhaps even allowing users to specify exact templates for labels to be based on in different levels of detail.

Tool state The SCCharts environment can simulate the execution of SCCharts, allowing users to set breakpoints, step through the execution, and examine signals and variables [Gri16]. Simulating and editing an SCChart are two entirely different states of the environment, each emphasizing different aspects of user interaction as well as of presenting the SCChart itself. The simulation, for instance, may provide information as to which states are currently active—something which can be taken into account by label management.

Active task Even if users are only editing or browsing a document, they may try to accomplish different tasks at different times. Trying to find a certain transition may put other demands on the visualization than tracing the flow of a signal through a diagram. Depending on how users work with a tool, determining the active task will usually be much more of a challenge than determining the tool's current state because of its much more implicit nature.

Current selection While diagram elements can only be selected once view model generation has finished, selections in other views of the same domain model may inform label management decisions already at this stage. SCCharts, for example, are edited using a textual editor whose current caret position or selection might conceivably influence the visualization.

With this selection of available information, the concept of focus and context [CMS99] immediately comes to mind. Focus and context is based on the assumptions that the user requires both details and an overview of a model, and that both can be provided in a single view. This can be done by showing focussed elements in more detail than their context, which is a principle that label management fits well. Determining whether an element is part of the focus or not can for example be done based on the information discussed above. States active during a simulation lend themselves well to being considered the focus, as Fuhrmann and von Hanxleden discuss [FH10].

To apply focus and context to label management using the preprocessing approach, different label management strategies can be applied to an element subject to whether it is part of the focus or not. As Musial and Jacobs show [MJ03], it can also be interesting to apply different label management strategies to different elements in the context, decreasing their level of detail as their distance to focussed elements increases.

Note that there are no prerequisites to applying the preprocessing approach. Even if neither the viewing framework nor the applied layout algorithms support label management, the diagram synthesis can still apply label management techniques directly to the view model that it generates and pass the results on to the view model presentation stage, which happily displays it while being completely oblivious to what went into creating it.

6.2.2 The Feedback Loop Approach

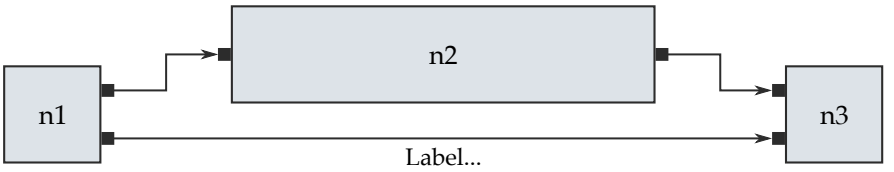
While the preprocessing approach can work well, it has two limitations: it disregards whether shortening a label is actually required from a layout perspective, and it forces the view model to be regenerated every time

label management desires to change a label. Let us examine each of these problems.

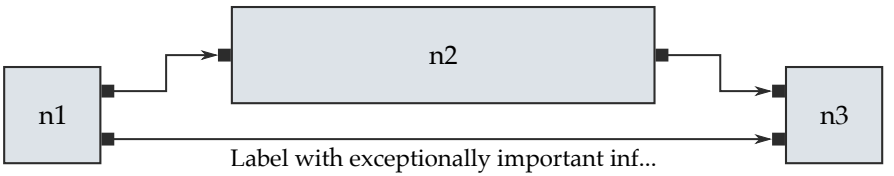
Regarding the first limitation, consider the example in Figure 6.5a, a label management result as might be produced by the preprocessing approach. Here, the label was syntactically abbreviated to an arbitrary target width. Considering our two label management goals, this may in fact be the desired result if what we wanted was to reduce visual clutter. If, on the other hand, increasing the diagram's scaling was the goal, label management seems to have been a bit too eager. The label can be a lot longer before it begins to influence the diagram's width, as Figure 6.5b demonstrates. Using a target width not derived from layout requirements may end up hiding more information than necessary, but trying to base the target width on layout requirements is impossible until one is aware of them—which the preprocessing approach is not.

The solution is to move label management from view model generation to view model presentation. Instead of invoking label management strategies right there and then, view model generation now annotates the view model with the strategies it intends to be used, but their actual invocation is deferred to the layout algorithm for it to feed back layout information into label management—hence the term *feedback approach*. The amount of space available to a label before it affects its diagram's size can now be used as the target width for label management. We will discuss exactly how ELK Layered computes that target width for different kinds of labels in Section 6.2.4. Based on that information, label managers can decide to simply leave a label untouched if it does not exceed the target width. Note, however, that they are just as free to ignore it, instead making their decisions based on all of the information they already had available in the preprocessing approach.

One interesting aspect of the feedback loop approach is that it changes the role of automatic layout. Traditionally, layout algorithms compute layouts for the exact graphs handed to them. With the feedback loop approach, the way a layout is computed can feed back into and thus influence what is being displayed if label management decisions are based on the target width. Of course, for that to be possible, let alone successful, the employed layout algorithm needs to be examined regarding whether it provides a natural way for deriving a meaningful target width for each label. We will



(a) Too aggressive shortening



(b) Just enough shortening

Figure 6.5. Two different approaches to shortening a label. **(a)** The label is shortened to a fixed target width which has no relation to the diagram's final layout. **(b)** The label is shortened just enough to not cause the diagram to grow wider than it would without the label.

soon see that there is a very obvious way for center edge labels in the layered approach, but force-based layout algorithms, for instance, will be much harder pressed to provide one.

Whether or not label management decisions take the target width into account, the feedback loop approach can also provide a solution to the second limitation of the preprocessing approach. Whenever actions by the user require label management decisions to be revised, the preprocessing approach requires the view model to be generated anew since it is run during the view model generation stage. The feedback loop approach, however, runs during automatic layout. If it suffices to change properties of the existing view model—such as annotating view model elements as being part of the focus or the context—label management can now be run on the existing view model, changing its properties and causing the view to be

updated, but sparing us the need of generating an entirely new one. This comes at the price of having to invoke label management on every layout run; the preprocessing approach only does so once, while creating the view model. If an application rarely changes label management decisions but updates a diagram's layout often, going with the preprocessing approach may turn out to be the better decision after all.

6.2.3 Combining the Approaches

We have discussed the preprocessing approach and the feedback loop approach as two different ways if integrating label management. One may well ask whether the two can be combined, and to what extent.

The feedback loop approach already seems like a combination of the two approaches: after all, the view model generation stage assigns label management strategies to the view model to be invoked during the view model presentation stage. View model generation in that way has a lot of influence over the label management results, but it does not invoke label management itself, delegating that responsibility to the view model presentation stage instead. This is not a bad thing; the feedback loop approach can have access to almost all of the information the preprocessing approach has access to—possibly with the exception of the domain model—and can thus do pretty much everything the preprocessing approach can. Yet there can be two problems.

The first problem involves said exception. We can imagine a label displaying different domain model information depending on its desired maximum size or level of detail. Not having access to the domain model in the feedback loop would seem like an obstacle to implementing this kind of behavior, but combining the two approaches can provide a solution: during preprocessing, the view model generation stage can generate all possible variants of the label's text and then install a label management strategy that knows when to switch between them. During the feedback loop, that label strategy selects the appropriate variant without having to access the domain model.

The second problem is one of performance. The feedback loop approach executes label management on every layout run, whether it will actually

do anything or not. It may well be, though, that the presentation or level of detail of certain labels will change very rarely. For example, an SCCharts editor might distinguish a focussed area and the context. While the context is always shown with reduced details, the exact presentation of focussed elements may depend on which one of them is currently selected. Since the level of detail of the context will not change, it makes sense to invoke label management on them once, when the view model is generated, instead of invoking it again and again upon every layout run.

6.2.4 Determining the Target Width

One detail we have yet to talk about is how the target width can be computed. This topic is of little relevance if none of the employed label management strategies use the target width in any way. Otherwise the options available for computing it depend first and foremost on the label management approach.

The preprocessing approach must resort to target widths computed according to fixed rules, based on the information available at this stage. The target width may be fixed for all labels or differ for different types of labels. If focus and context is used, enforcing different levels of detail for different elements in the context might be implemented as decreasing target widths, each set either by the application's interaction designers or through user preferences.

The feedback loop approach can do the same, but can also use layout information to compute the target width. Let us examine how this can work using ELK Layered as our example.

Center edge labels are represented in ELK Layered as label dummy nodes. As you will recall from Section 3.1.2, which layer they end up in gets computed in the algorithm's second phase, layer assignment (although the assignment may change later depending on the active layer selection strategy). Once this is known we have a natural way to compute a target width for center labels: simply use the maximum width of each node in the same layer which is not a label dummy itself, as shown in Figure 6.6. This ensures that label management is not told to shorten labels more than necessary.

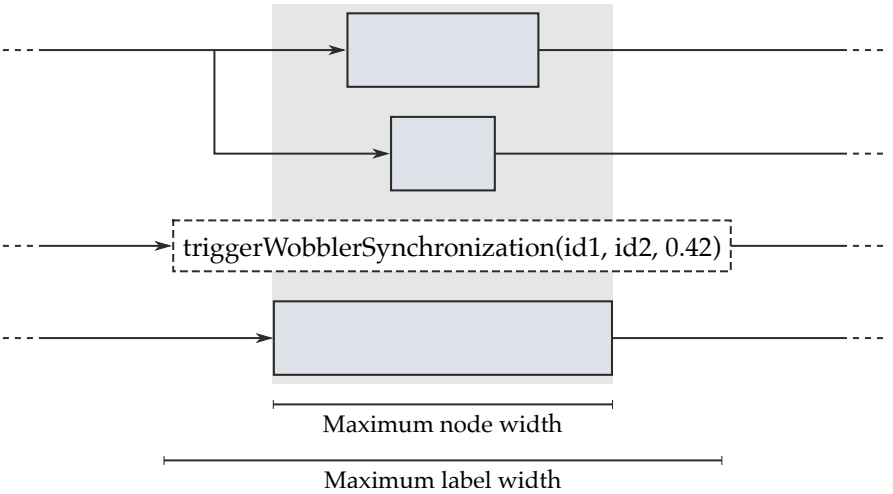


Figure 6.6. The layered approach provides an obvious way to compute the target width for center edge labels.

This leads directly to a limitation of using the feedback loop approach with ELK Layered. For horizontal layouts, the width of a center label has a direct relationship to the width of its layer, which makes it easy to derive a meaningful target width. This is not the case for vertical layouts: here, changing the text of a label actually changes the label's height instead of its width in the algorithm's internal representation of the layout problem. Label management then has an impact on node placement. While we have not looked into this yet, future research might investigate extending node placement algorithms to make use of label management to improve their placement results.

Target widths for all other kinds of labels (edge end labels, node labels, and port labels) are not as obvious to derive. Here, ELK Layered resorts to fixed target widths. Future research could investigate ways for computing meaningful target widths. For node and port labels in particular it might be interesting to examine how the micro layout algorithm introduced in Section 3.2 might be able to provide target widths.

6.2.5 A Label Management Framework

I implemented the feedback loop approach in the ELK and KLighD projects for clients to build their solutions on. The preprocessing approach can be used as well simply by writing KLighD syntheses accordingly.

Basic support for the feedback loop approach to label management was added to the ELK project. It consists of an interface that defines the notion of a label manager, aptly called `ILabelManager`. Each label manager implements a label management strategy, such as the ones described in Section 6.1. Label managers can be installed on a graph through a dedicated property set on a hierarchical node to be applied to its child graph. Layout algorithms that support label management (which ELK Layered does) look for that property and call label managers at the appropriate time during their execution.

The `ILabelManager` interface only defines a single method that needs to be implemented:

```
KVector manageLabelSize(Object label, double targetWidth)
```

The parameters to the method are self-explanatory, but the return value might not be. It is a two-dimensional vector which describes the new size of the label after the manager has wrought its magic. The new size is what layout algorithms are expected to continue their computations with.

Actual implementations of the label managers described previously are not part of ELK, but of the KLighD project. This might seem strange at first, but the reasons are rather straightforward. ELK needs to provide the basic interface for its layout algorithms to know about label management, but the label managers need to be able to find out what their actions would do to the size of a label. This requires knowledge about how the label will be rendered, which is specific to the viewing framework. Label managers also need to apply their results back to the labels somehow, which is specific to the viewing framework as well.

To make things as easy as possible for developers of label managers, KLighD provides the `AbstractKlighdLabelManager` class which all label managers intended to be used with KLighD must extend. Subclasses only have to implement a single method tailored to `ElkLabel` instances:

```
Result doResizeLabel(ElkLabel label, double targetWidth)
```

Note that contrary to the entry method specified by `ILabelManager`—which is implemented by `AbstractKlighdLabelManager`—this method does not return a label's new size. Instead, the returned `Result` instance describes whether the label manager did anything to the label and, if so, contains the label's new text which the base class then uses to compute its new size.

We already discussed how the size is returned to the layout algorithm, but what we have so far skillfully skipped over is how the new text makes its way back to KLighD. In fact, there is a problem there that KLighD needs to solve to make label management work: what it displays is the view model generated from the domain model, but what label management is called on is the layout graph created from the view model. This is the reason why the `label` parameter of the `doResizeLabel(...)` method above is of type `ElkLabel` instead of some view model class specific to KLighD.

The problem is solved by modifying the label in the layout graph in two ways: first, its new text and size are applied back to it to be used by the layout algorithm. And second, a property is set on the label that indicates whether it had label management applied to it and whether its text was changed as a result; this information is based on the `Result` object returned by the `doResizeLabel(...)` method. When automatic layout has finished and KLighD applies the results back to the view model (see Section 2.6) it looks for that property. If a label was subject to label management and its text was changed, a new label text override is set on the view model that instructs the viewer to display the new text instead of the label's original text. Otherwise, any such override is removed, causing the label's original text to be displayed. Note how the label's original text is never changed in the view model, which is an important detail: label management always needs to be applied to a label's original text for it to be able to restore that text if need be.

Apart from these technicalities, `AbstractKlighdLabelManager` also adds behavior and options to label management. The most important thing it provides is an activity state: label managers can be configured to be either active or not. If an inactive label manager is invoked on a label, it always leaves its original text untouched. The second thing is a mode of operation

that provides hints to implementations as to how they should behave. Label managers have two possible modes:

TargetWidth — Label managers are advised to only lay hands on a label if its original text exceeds the target width.

AlwaysOn — Label managers are advised to always lay hands on a label, regardless of whether it exceeds the target width or not. This mode is useful if the main goal is to reduce visual clutter.

While generally not a problem for label management strategies that work on a semantic level, not all of the universal strategies we have seen would be able to work in AlwaysOn mode. For example, it is not clear how syntactical abbreviation should shorten a label if it already fits into the target width. To fix this and to be able to implement different levels of detail for elements in the context, label managers can be configured to use a fixed width. If configured that way, subclass implementations are called with that fixed width as the target width, not with whatever value was passed to the entry method.

Besides the base class for label managers, KLighD offers another piece of functionality useful to the feedback approach in particular. If label management decisions are to depend on whether an element is focussed or not, tool developers need a way to determine just that. The `FocusAndContextAction` provides a way to do so by setting and updating a special property on the elements of a diagram. The usual procedure is to configure the action to be invoked when a diagram element is clicked. It then proceeds to do two things:

1. It removes all previously focussed elements from the focus by setting their corresponding property value to `false`.

2. It calculates which elements are part of the new focus. The selected element is always focussed. If the element has labels, those are focussed as well. If the element is a node, any attached comment nodes are also focussed, as are the node's child nodes, if any.

3. It updates the property value on the focussed elements.

The class is designed such that subclasses can customize the second step. We will see an example of such customization in our sequence diagrams case study in Section 6.4.1.

6.2.6 Label Management and Automatic Layout

The feedback loop approach to label management requires it to be invoked by layout algorithms at a time when they have enough information to provide sensible target widths. We have already seen the basic idea of how this works in ELK Layered, and have concluded that label management can be invoked as soon as layer assignment has completed. This provided us with a good first intuition, but as usual the reality is actually a bit more complicated and is what we will dive into a bit more now.

If label management is active, ELK Layered has a number of tasks to perform during its execution:

1. Micro layout needs to be invoked on all regular nodes. While a node's size *can* depend on the size of its labels, a label's position *will* depend on its size. It follows that label management must have run.

2. End labels must be placed and node margins must be updated to include them. It follows that label management must be complete for edge end labels.

3. Label dummy nodes created for center edge labels need to be moved to their final layers according to the layer selection strategies introduced in Section 3.3.3. This requires micro layout to be finished and node margins to be final since some strategies take the width of regular nodes into account.

4. Label management must be invoked. This requires that label dummy nodes have been moved to their final layer since ELK Layered uses the width of its regular node's to derive the target width for center edge labels.

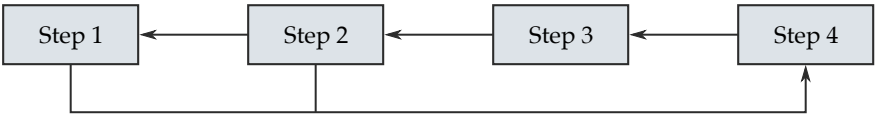


Figure 6.7. The four steps involved with label management would form cyclic dependencies when implemented naively.

What we end up with are cyclic dependencies, as shown in Figure 6.7: each step depends on the previous one, but steps 1 and 2 depend on step 4. The solution is to see that the different steps require label management to have run not on all labels, but on different subsets. Step 1 needs the size of node and port labels to be fixed, step 2 required the same for edge end labels, and step 4 then only has center edge labels left. We thus run label management twice: once on node, port, and edge end labels before step 1, and once on center edge labels in step 4.

We finish with a final detail concerning step 3. The decision of which layer to place a label dummy node in can depend on the size of other nodes in that layer. If we take the width of label dummy nodes into account, but apply label management afterwards, the result may look wrong: we may have assumed a very wide label to cause its layer to grow very wide as well, but label management may have considerably shortened the label, causing another layer to be wider. If we do not take label dummy nodes into account, we have the opposite problem: a layer may be much wider than expected if label management was not able to shorten it much. Whatever we do, we cannot expect to get a perfect result. Similar reasons also lead us to derive the target width in step 4 from the width of regular nodes only.

6.3 Designing the User Interface

Label management has the power to change the text displayed by labels, to the point of potentially hiding them completely. It does so either rather statically (preprocessing approach) or very dynamically (feedback loop approach). Especially dynamic label management needs to be well designed from a UI perspective to help users instead of confusing them. The follow-

ing sections will examine aspects to think about when presenting label management to the user.

6.3.1 Providing Label Management to Users

The way label management is applied should always be helpful to users. Having said that, deciding what is helpful and what is not is tough.

One extreme is to provide only a single label management configuration intended to work well in most situations. Finding such a configuration might entail performing a number of user studies. In this case, label management does not need to be advertised as a distinct feature in the UI, but can just be part of "the way the application works."

Whether a single label management configuration can be found that works well in most situations depends on the application it is to be used in. It seems reasonable to assume, though, that how helpful a given configuration is depends on the task the user is trying to accomplish. How that task can be determined depends on the application:

- Returning to the concepts of the MVC pattern, an application might offer different views of a model designed for different tasks, each employing a different label management configuration. Again, label management then does not need to be explicitly advertised, but becomes a part of how each view works.

- An application may provide different modes of operation. The Eclipse IDE, for example, provides different view configurations (called *perspectives*) for Java development and debugging. Similarly, an SCCharts IDE might offer a development mode, a browsing mode, and an execution mode.

Users can be given more control over how label management is configured. This might take the form of offering a selection of strategies to switch between, a concept used in the SCCharts editor in Figure 1.6. Care has to be taken to communicate what the strategies do using terminology that users have a chance of understanding.

Going one step further, users might even be provided with the means to define their own label management strategies. This could take the form

of a simple DSL or, perhaps more appropriately, a specialized and simple UI to specify transformations from the original text to the intended result [LGA+02]. While this does seem like an interesting concept, it might be a challenge to design such a feature in a way that is both easy and expressive enough. Still, advanced users might take advantage of the possibility to tailor label management strategies to their personal requirements.

However label management is presented to users, it is paramount that it works in a way that is always clear and never confusing. One example is restoring a label to its original text by clicking on it, which must make sure to preserve the user's mental map of a diagram as well as communicate clearly what changes were made to the label. The first requirement can be met by preferring layout algorithms that provide a high degree of layout stability. The second requirement can involve animating layout changes.

As a final point, note that scenarios in which visual clutter shall be reduced may not work too well when label management decisions are based solely on a target width computed by a layout algorithm. The target width has no semantic connection to the diagram since it is only influenced by its layout. Reducing visual clutter, on the other hand, will usually be achieved by reducing the semantic level of detail of a label—we have seen different levels of detail for transition labels in Figure 6.3. If the target width influences the semantic level of detail, this might be a source of irritation for users. It seems more promising to make such decisions based on other, more obvious criteria, such as whether an element is currently in focus or not.

6.3.2 Communicating Managed Labels

If label management was invoked on a label and has in fact shortened the text, that fact may need to be communicated to the user for them to know that more information is available. An obvious way to do this—and one we already encountered when we discussed syntactical abbreviation in Section 6.1.1—is to simply append an ellipsis to the label's shortened text. Users are already accustomed to this kind of hint: the standard file managers of all major operating systems do the same thing if a file name is too long to fit its column.

Another way is to change the label's graphical representation slightly by for example changing the color of its font or background, changing the typeface slightly, or adding a small graphical hint. Microsoft Excel, for instance, adds a small red triangle to a cell's corner if the cell has a comment attached to it.

Usually the user also needs to be provided with a label's original text. Again, there are different possibilities to do so. The most obvious is to display a tool tip-like hint when the mouse cursor hovers over the label. This has several advantages. First, it displays the information right next to the element it belongs to, allowing the user to maintain their visual focus. Second, it is a simple overlay and does not require the diagram to be changed in any way. Finally, and perhaps most importantly, it is a standard UI feature that users are already familiar with and can thus discover intuitively.

Another possibility is to expand the label to its original text once the user clicks on it, a special case of focus and context. The information are displayed in place without requiring the user to look anywhere else, but this requires automatic layout to be invoked again to ensure that enough space is available for the expanded label. Key issues are layout stability (users will not expect the overall layout to change just because they happened to select a label) and managing the viewport properly to keep it from changing too much.

Finally, a label's original text might be displayed in a second view next to the diagram, either upon hovering over a label with the mouse cursor or upon selecting the label. This solution requires the user to shift their view away from the label, but does not require automatic layout to be run again. Also, if the original text is rather long or if more information about a diagram element should be displayed along with it, an auxiliary view may actually be better suited to display it than the diagram itself or a tooltip. Eclipse uses this approach to show Javadoc information relevant to the current cursor location in a separate view.

6.4 Evaluation

My evaluation of label management consists of three parts: first up are three case studies, followed by an aesthetic evaluation that aims to quantify the size reductions the different label management strategies can achieve. We close with a survey done among students that used label management.

6.4.1 Case Studies

Visual languages differ in how they use labels and what tasks their users typically perform while working with a language. It follows that label management is not something that can be applied as is; it must rather be adapted to the requirements of each visual language it is supposed to be used in. We have done so for the three visual languages introduced back in Section 1.1: Ptolemy, SCCharts, and UML sequence diagrams.

KIELER Ptolemy Browser

I integrated label management into the KIELER Ptolemy Browser, which already existed prior to this work. While Ptolemy II block diagrams do not have edge labels, they do feature node labels, port labels, and comments.

Node labels help users identify nodes and distinguish them if they share the same visual representation. Given this critical function and the fact that they usually do not contribute much to the size of a diagram, we did not subject node labels to label management and instead chose to always show them in full.

Port labels are another matter entirely. Vergil, the editing environment Ptolemy II ships with, allows users to statically configure for each port whether its label should be shown (Figure 6.8).

The KIELER Ptolemy Browser takes another approach: instead of allowing for such a level of fine-grained, but ultimately static control, it provides access to label management strategies that are applied to all port labels at once: always hiding port labels, always showing port labels, or showing port labels of selected nodes only (Figure 6.9). Changing the selection in the latter case causes port labels of the previously selected node to be hidden

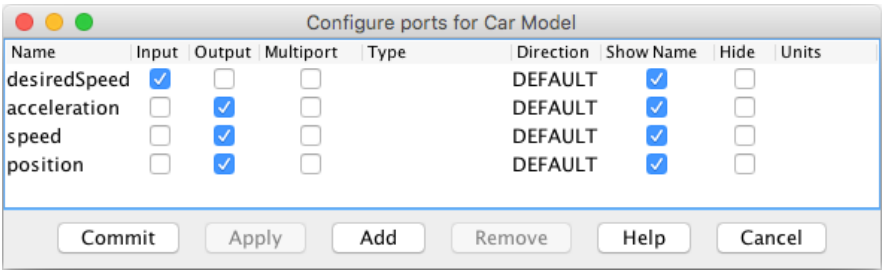


Figure 6.8. Ptolemy II's editing environment allows the label of each port to be switched on or off through the check boxes in the *Show Name* column.

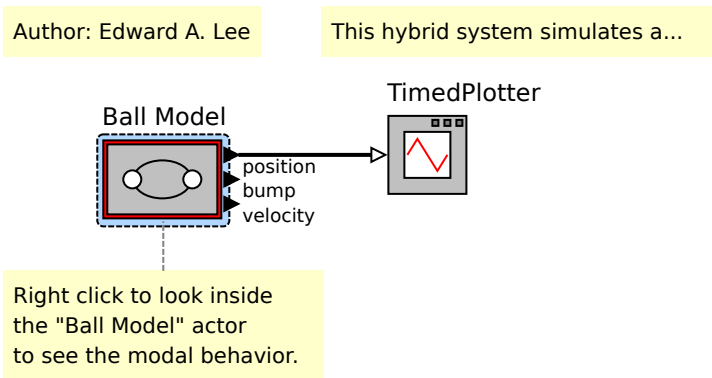


Figure 6.9. The KIELER Ptolemy Browser offers different label management strategies for port labels and comments. In this case, only the port labels and comments of the selected node are shown, which happens to be the *Ball Model*.

and port labels of the newly selected node to be shown, which of course requires automatic layout to be triggered. Hence, this is not only an example of an opportunity enabled by automatic layout but also a text book case for focus and context. It is important to help users understand what is happening, which in this case is done by animating the layout changes.

Although not quite labels at first glance, a similar selection of approaches is used for comments in Ptolemy II diagrams: we can hide all of them, show

all of them in their entirety, or show all of them with their text shortened except for those that are either selected themselves or belong to a selected node. The latter case is shown in Figure 6.9 as well. Technically, comments in the KIELER Ptolemy Browser are little more than nodes with a special flag and an inside center label which can be fed to label management algorithms.

Implementation-wise, the KIELER Ptolemy Browser simply uses KLighD's focus and context action as introduced in Section 6.2.5 without any customization.

SCCharts

I integrated a number of different label management strategies into the SCCharts editing environment we saw back in Figure 1.6:

- Host code calls
- Transition priorities
- Syntactical abbreviation
- Semantical abbreviation
- Syntactical soft wrapping
- Semantical soft wrapping

Only syntactical abbreviation, transition priorities, and semantical wrapping are actually advertised to users.

One half of the strategies work as expected, but the other half deserves further explanation. Syntactical abbreviation would usually cut a label's text off once it reaches the target width supplied by the layout algorithm. The SCCharts editor developed at our research group instead allows users to specify a custom target width to be used. Semantical abbreviation shortens transition labels to the names of signals involved in their trigger or effect. Semantical wrapping, inspired by Castelló et al. [CMT01], divides transition labels into their different components (transition priority, trigger, and effect) and invokes syntactical soft wrapping on each, recombining the results with line breaks between them.

One interesting detail concerns the way that semantical wrapping retrieves the three transition label components. The obvious way is to look for a colon, which separates the transition priority, and for a slash character, which separates trigger and effect. This approach has two problems, though. First, slash characters can appear as division operators in both trigger and effect expressions, making it hard to decide where one ends and the other begins. And second, the actual characters used to separate the three components are a mere presentation detail and might be changed in future versions, requiring the label management code to be changed as well. This is why the semantical wrapping implementation takes another route to retrieve the components: instead of parsing the text of the label to be shortened, it retrieves the actual domain model object, the transition itself. There, the three components are stored as three different properties, rendering any parsing efforts completely unnecessary.

Of course, label management can also be switched off by users. Doing so does not really switch it off, though: the option merely installs an *identity label manager*, the sole purpose of which is to restore each label's original text. Without actually turning off label management, it certainly appears that way to the user (who has nevertheless just been lied to, of course).

A final detail concerns the way users can retrieve a shortened label's original text. The tool provides the usual tool tips, but it also contains a customized focus and context approach. If a transition label is selected, only that label becomes the focus. If a signal name is selected, however, all transition labels that reference that signal are focussed.

The surveys about to be described in Section 6.4.3 involve users of the SCCharts editing environment.

UML Sequence Diagrams

Sequence diagrams, prone to becoming large rather quickly [BMM04], are another good candidate for the application of label management. They use labels at several points, which we will examine in order to discuss if and how they are label managed.

First up is the title of a sequence diagram's interaction. This will usually be much shorter than the interaction's content is wide and does not

contribute a lot to the diagram's clutter. We thus exempt it from label management.

More relevant are the titles of lifelines. We believe them to be important enough, though, for them not to be shortened, similar to node titles in Ptolemy II diagrams. We thus exempt lifeline titles from label management as well.

Message labels are more interesting. If placed near the message's source lifeline, they influence the amount of space between that and one of its adjacent lifelines, quickly pushing them apart and enlarging the diagram in the process. This is a prime candidate for label management.

We allow users to switch between two label management strategies. The first is a label management strategy which, similar to the host code call strategy in SCCharts, removes the arguments of method calls. While this seems promising for sequence diagrams that model interactions based on method calls, it will not work as well for diagrams that do not. The second strategy is thus simple syntactical abbreviation with the target width set to the minimum amount of space available between the two adjacent lifelines.

Just like messages, comments can also push lifelines apart and thereby enlarge sequence diagrams. Unlike messages, they can even contain several lines of text. We thus start by removing all but the first five words, followed by syntactical abbreviation to ensure that what remains does not enlarge the diagram.

Besides offering tool tips that provide access to the original text, the sequence diagram editor also uses focus and context. A customized version of KLighD's focus and context action calculates an appropriate set of elements to focus once the user selects an element. If a comment is selected, only that is focussed. If a message is selected, its label and any associated comments are focussed. If a lifeline is focussed, any incident messages and associated comments are focussed.

This behavior led to an interesting discovery. Imagine that the labels of the comment and the messages in the sequence diagram shown back in Figure 4.3 were much longer than they are in that diagram, so much so that they push the rightmost two lifelines apart. Suppose further that label management is engaged, which promptly pulls the lifelines closer together again by shortening the text of all messages and comments. Finally, suppose

that the user selects the uppermost message. The message's original text is restored, pushing the lifelines apart again. While this does not come as a surprise, what can come as one is what happens to the comment. It is not part of the focus, but the additional space available between the lifelines can allow label management to show more of the comment's text as well.

When this happened during a presentation, someone in the audience commented that they found this behavior confusing. This is an excellent example of what we discussed in Section 6.3: label management should help users, not confuse them. In this case, confusion might be reduced by always shortening a comment to its first few words but not applying syntactical abbreviation afterwards. We would thus accept situations where the first five words might push lifelines apart to reduce confusion among users. Having said that, we have not yet implemented and tested this strategy.

6.4.2 Aesthetics

The three case studies gave us a good impression of how well the label management framework adapts to different applications. What they did not give us were more detailed data regarding the actual effects label management has on diagram aesthetics, such as the size of the final drawing and, with that, the extent to which it abides the **P-SIZE** principle. This is what we will now be looking at in the form of two experiments, one based on SCCharts and one on sequence diagrams.

SCCharts

We based our first experiment on SCCharts, for two reasons. One, they are particularly prone to long labels, and two, SCCharts provide a number of different strategies for us to compare. We did so based on two sets of models. The first consisted of models produced by the students with whom we conducted the introductory survey mentioned at the beginning of the chapter. These were students that worked with SCCharts as part of their homework. They submitted a total of 76 SCCharts that contained 2,046 states (237 of which contained child states) and 3,198 transition labels. The second set of models contained an additional 15 SCCharts that are part of a complex

piece of software that controls 11 trains on a model railway installation. The diagrams contain a total of 2,053 states (118 of which contained child states) and 2,643 transition labels. Samples of the SCCharts are available in Section A.1.2.

Every diagram was laid out once with every available label management strategy as well as with label management switched off. We did not evaluate the signal abbreviation strategy since an implementation was not available and the results would not differ too much from those produced by syntactical abbreviation, provided the target width is not set too low. Those strategies that pay attention to the target width were used twice: once with a dynamic target width supplied by ELK Layered, and once with the target width fixed to 200 pixels, the current default in the SCCharts editing environment.

We measured each diagram's width and height, the resulting aspect ratio (width divided by height), the width and height of each label, and the length of each edge. Since one of the goals of label management is to be able to increase the zoom level a diagram is displayed with, we also calculated each diagram's maximum scale for a typical 16:9 screen for each label management strategy and calculated the scaling factors with respect to label management switched off.

The results, shown in Table 6.1, conform to expectations for the most part. Label management strategies based on word wrapping exchange width for height, sometimes producing substantially higher drawings, as in the case of semantical wrapping. On average, that strategy came closest to the aspect ratio of today's wide screen monitors (assumed here to be between 1.6 for 16:10 and about 1.8 for 16:9 screens).

Perhaps more interestingly, Figure 6.10 shows the average scaling factors. Differences to results published previously [SLH16] are due to changes in both layout algorithms and label management strategies that have since been implemented. Unsurprisingly, the transition priorities strategy allows for the biggest scaling factors, but comes at the cost of a substantial loss of information. As one would expect as well, only throwing away the arguments of method calls hardly helps (during the analysis, it only even laid hands upon about 10.5% of all labels). More interestingly, the average scaling factors achieved by the other strategies do not differ that much,

Table 6.1. Aesthetics measured on 91 diagrams with ten different label management strategies. Strategies that take the target width into account were run once with a fixed target width and once with the target width supplied by ELK Layered. We measured label size and edge length averages for each diagram and report the averaged averages here.

	Drawing				Labels		Edges
	Width	Height	Area	Aspect	Width	Height	Length
Off	7,329	718	5,297,077	11.6	315	13	636
Independent of target width							
Host Code	7,158	718	5,186,652	11.5	308	13	628
Priorities	2,758	681	2,083,025	4.2	11	13	135
Sem. Abbr.	4,853	706	3,585,191	7.6	152	13	377
Target width fixed to 200 pixels							
Sem. Wrap.	4,608	1,066	5,180,553	5.0	136	35	349
Synt. Abbr.	4,827	716	3,580,881	7.8	165	13	350
Synt. Wrap.	4,838	987	5,083,315	5.8	156	29	369
Target width computed by layout algorithm							
Sem. Wrap.	4,350	1,931	9,254,205	2.9	80	83	357
Synt. Abbr.	3,237	695	2,457,096	5.0	60	13	185
Synt. Wrap.	4,149	1,577	7,037,668	3.2	82	66	314

except for syntactical abbreviation if it uses the target width supplied by ELK Layered instead of the default 200 pixels. We believe there to be two reasons for that: first, non-macro states in SCCharts tend to not be very wide; and second, while ELK Layered usually derives the target width from the width of non-dummy nodes in a layer, it falls back to 60 pixels if a layer contains none.

Which label management strategy should be used is a trade-off between increased scaling and information loss. Since the scaling increases turn out to be comparable in most cases, we believe the choice to be mostly a matter of finding the strategy which bests supports the tasks users usually have to perform.

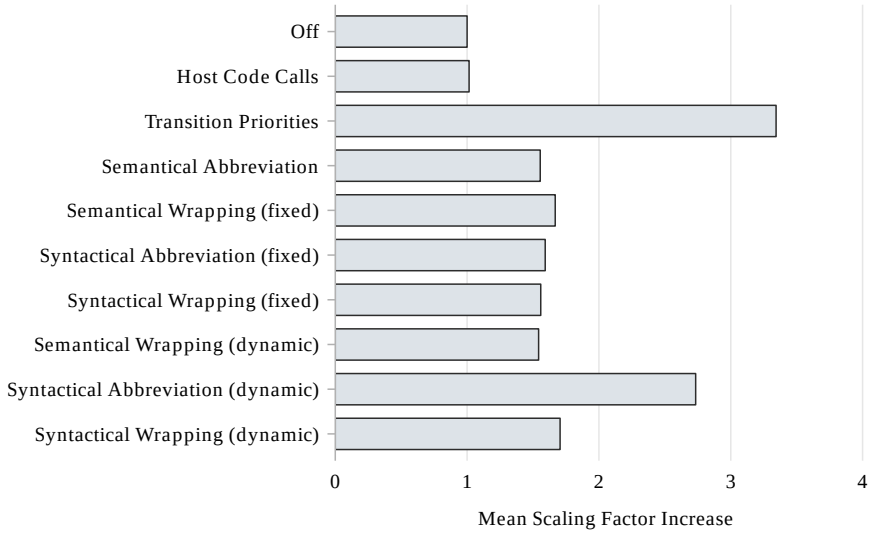


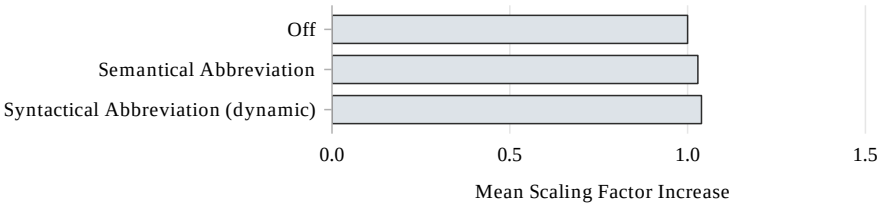
Figure 6.10. The factor by which the scaling can be increased to fit a whole SCChart on screen as compared to the scaling required without label management. Higher values are better.

Sequence Diagrams

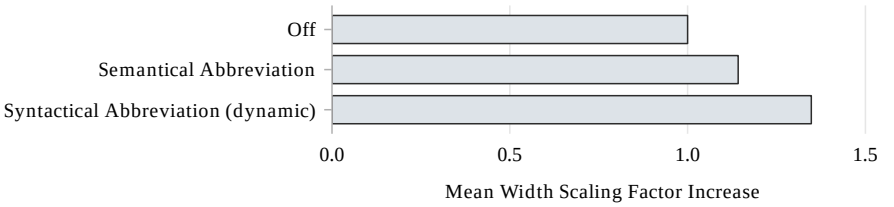
Our second experiment was based on the set of sequence diagrams already used in their evaluation in Section 4.4 (as mentioned there, samples are available in Section A.2). We followed the same procedure as in the SCCharts experiment.

Figure 6.11a shows the mean maximum scaling factors, which, perhaps surprisingly, turn out much less successful than the ones for SCCharts. The values do make sense, however: quite often, sequence diagrams are dominated by their height (true for 78% of diagrams in our data set), which dramatically reduces the impact of label management, a technique focussed mainly on reducing the width of diagrams.

Which impact, then, do the label management strategies have on the width of our diagrams? Figure 6.11b shows the results we get with respect



(a) Both dimensions



(b) Width only

Figure 6.11. The factor by which the scaling can be increased to fit a whole sequence diagram on screen as compared to the scaling required without label management. Higher values are better. **(a)** The usual scaling which takes both a diagram's width and height into account. **(b)** Scaling when taking only a diagram's width into account.

to the full-width scales. The impact is more obvious here, with syntactical abbreviation yielding a mean scaling factor of 1.35.

If label management does not yield good overall scaling increases, but makes narrow diagrams narrower, why use it in the first place? First, not all diagrams are dominated by their height. And second, applying label management allows lifelines to move closer together. This helps when zooming into a diagram since more of it can be displayed on a screen, reducing the need for users to pan the view.

6.4.3 Two Surveys

While the informal survey mentioned in the chapter's beginning gave us a first idea as to whether long labels are a problem, we wanted to obtain more feedback from users of the SCCharts editing environment. The two surveys already described in Section 3.4.4 were not just about edge label placement, but also contained questions on label management. The students had three label management strategies to choose from: syntactical abbreviation, semantical wrapping, and transition priorities. Label management could also be turned off.

In the following, we will not repeat the descriptions of the surveys (the reader is referred back Section 3.4.4 for details), but concentrate on differences and the actual results.

First Survey

Seven teams of students responded to the survey, but only six provided usable answers. The seventh team seems to have experienced technical difficulties, claiming that only the transition priority strategy had any effect on labels—which certainly would not have been the case if the strategies worked as intended.

The survey consisted of two tasks:

1. Rate the label management settings from best to worst and explain your rating.
2. Provide any thoughts and comments you might have on the label management settings.

The sums of the ratings given to the different strategies are shown in Figure 6.12. All in all, they perform about equally well, with the exception of the transition priorities strategy, which did worse than the others. Four teams highlighted that this strategy results in labels that provide little or even too little information, one complaining that "as this method just shows priorities and no guards and actions it took us the longest time to understand the diagram." At the same time, that team acknowledged that "you can see quickly which states are connected to which states," a

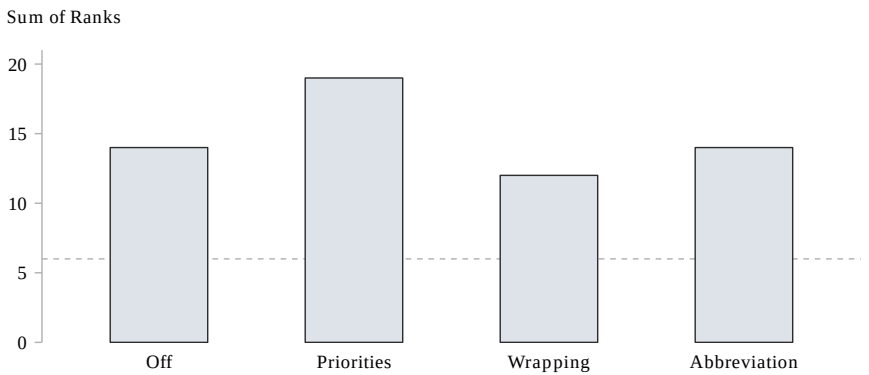


Figure 6.12. Sum of the ranks assigned by students to the four available label side selection strategies during the semester (lower sum is better). Since six teams of students provided usable data, the optimal result a strategy could have gotten is six (dashed line).

sentiment shared by two other teams that deemed this strategy to be useful for large SCCharts.

Syntactical abbreviation generally received more favorable reviews. On the negative side, three teams criticized that a lot of information may be cut from a label, with one team noting that this might be particularly inauspicious if transition labels start with similar trigger expressions and differ mainly in what the strategy cuts off. On the positive side, one team found this strategy to be "awesome for SCCharts with exceptionally long transitions," a sentiment shared by another team, and commended the tool tips that provide access to each label's original content. One team noted that whether or not they liked this strategy depended on the use case: while they were developing an SCChart, they liked this strategy, but if their goal was to understand an SCChart, they preferred lossless label management strategies.

Semantical wrapping received the highest amount of praise. Four teams liked this strategy, one explicitly mentioning its lossless nature as an advantage. Interestingly, while three teams liked how this strategy affected

the size and aspect ratio of their diagrams, one did not, noting that "most monitor formats provide a lot more width than height, making [semantical wrapping] a bit less effective for bigger SCCharts." One team simply stated that they "just don't like it," without further explanation.

Turning off label management received mixed reviews. One team found this to be the "best choice for small to medium SCCharts" because all details are still visible. Another team disagreed, eloquently stating that turning label management off "gets a very remote, but well-deserved last place" due to the unfortunate size and aspect ratio of unmanaged SCCharts, a problem also mentioned by another team.

There were several further comments of interest. Two teams proposed that changing the font size of labels may be a good idea, one stating that "one could maximize readability with a minimum of 'lost' information." One team suggested to hide labels completely once the zoom level drops below the point of readability. Unrelated to label management as such, another team suggested to allow users to switch between layer selection strategies (see Section 3.3.3).

Finally, one team stated that label management "was really good because it improved clarity, which was highly noticeable."

Second Survey

In the second survey, we asked the following question: "Since their introduction, which label management strategies did you end up using regularly?" Multiple selections were possible.

The results, shown in Figure 6.13, did not come as much of a surprise given the results of the first survey: not one of the eleven students that took part used the transition priorities strategy.

An interesting observation is that only eight students turned off label management regularly. A closer look at the data reveals that half of them did not use any other strategies, but the other half did. Based on some of the responses to the first survey, it seems probable that students activated different strategies as they worked on different tasks.

Asked for further comments, which four students provided, one student repeated an opinion we already encountered in the first survey, saying that

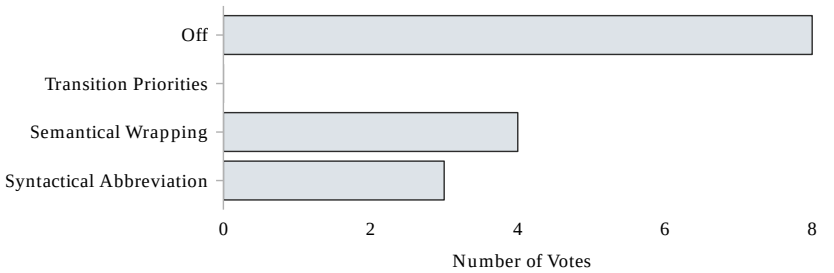


Figure 6.13. The label management strategies used by the eleven students remaining at the end of the semester. Multiple answers were possible. Longer bars are better.

syntactical abbreviation "is incredibly useful in SCCharts with increased complexity." Another response suggests that the SCCharts built by the students during the semester may have been too small for label management to be perceived as a useful feature: "I feel like the idea is very good, however with the models we made, it didn't really make sense to use anything other than the original labels." Sadly, that student did not provide further details, for example regarding why they did not consider semantical abbreviation a viable alternative to switching off label management entirely.

This concludes the two main parts of this thesis: how to lay out diagrams that contain text, and what additional problems arise once we use automatic layout to create diagrams on the fly and thus allow for new user experiences. What remains is to summarize and conclude this work.

Chapter 7

Conclusions

With the main chapters behind us, it is time to wrap things up with a summary of the results, lessons learned in the process, and possible venues for future work.

7.1 Summary

Throughout this thesis, we made our way through different aspects of text in diagrams: how support for text placement can be integrated into layout algorithms in the first place (**P-PLACEMENT** and **P-SIZE**), how we might improve support for secondary notation as it applies to comments (**P-NOTATION**), and finally how to improve the usability of text-intensive visual languages (**P-SIZE** and **P-CLUTTER**).

Layout Support for Text We distinguished labels for nodes, ports, and edges which we elevated to first-class citizens of automatic layout instead of trying to squeeze them into layouts that have already been computed.

The micro layout algorithm presented in Section 3.2, following the **P-PLACEMENT** principle, introduced support for placing node and port labels, for taking them into account when placing the ports themselves, and for calculating the size of their nodes according to a number of different configuration options originally derived from requirements of visual languages we encountered at our group. The algorithm's foundation was the cell system, a simple arrangement of containers that, if configured appropriately, made calculating a node's micro layout comparatively easy. The implementation is available to be called by any layout algorithm based on ELK.

Applying the **P-PLACEMENT** principle to edge labels, we distinguished between two kinds: end labels, placed right at the end points of their edge, and center labels, placed anywhere in between. While placing end labels was comparatively straightforward, placing center labels granted freedom along two dimensions: where along an edge the label should be placed (layer selection) and whether to place it above, on, or below its edge (side selection).

We introduced and compared a number of layer selection strategies, some aiming to minimize the width of diagrams to contribute to the **P-SIZE** principle—a hard, but worthwhile exercise. However, while reducing a diagram's width certainly seems desirable, we argued that it does introduce a certain amount of unpredictability regarding where users should look for edge labels; some languages may benefit more from consistent placement strategies.

We also introduced a number of side selection strategies. Same-side label placement is the current state of practice, which we improved upon by having augmented same-side placement solve O-No placements. The rules that governed placement decisions were developed to improve ambiguous placements, and we found that it succeeded in reducing edge length, particularly in vertical layouts. Direction-dependent side selection used label placement to encode information about where an edge is heading, but was found to be very badly received by participants in our surveys and our study. Directional decorators, which added explicit arrows, were much more successful in that regard: not only were they popular with users, but they also significantly reduced response times and error rates when inferring edge directions. We finally covered on-edge label placement along with a number of necessary design considerations tool developers may have to think about. While previous publications had already made use of on-edge labels, our user study and our surveys provided evidence that it seems to be a very well received placement strategy.

All of these layer and side selection strategies were implemented in the ELK Layered algorithm.

Text and Secondary Notation As stated in the **P-NOTATION** principle, layout creation algorithms in particular seem bad at retaining secondary

notation. One reason for this shortcoming may be that secondary notation seems inherently hard to operationalize and thus to measure and properly act upon.

Concentrating on the placement of comments in Ptolemy II diagrams, we introduced a comment attachment pipeline with the aim of computing which node any given comment describes. To that end we developed two types of heuristics: filters, meant to recognize comments that do not relate to any particular node, and matchers, meant to compute the likelihood of a comment relating to a given node.

While we were able to infer correct attachments for up to about 90% of the comments in the diagrams we surveyed, we found that our comparatively simple heuristics are not sufficient to accurately describe how comments relate to nodes and thus help close the gap towards 100% accuracy.

The comment attachment pipeline is available as part of the KLighD project and is supported by the KIELER Ptolemy Browser.

Unwieldy Amounts of Text While text is an important part of most visual languages, we saw that it can become so long that it introduces two problems: increasing the size of diagrams such that they need to be scaled down into the realms of eligibility to be completely displayed on today's screens (**P-SIZE**), and overloading the diagram with information that may not actually be important to solve a given task (**P-CLUTTER**). Building upon the original proposal by Fuhrmann [Fuh11], we investigated label management as a possible solution.

We described and introduced a number of basic label management strategies which, similar to audio compression algorithms, could be divided into lossy and lossless specimen. Some strategies were applicable to any language and some were specific to SCCharts or sequence diagrams, and we showed how to combine them into composite strategies.

We presented two different approaches for integrating label management into the view generation process: the preprocessing approach and the feedback loop approach.

The preprocessing approach applied label management while transforming a domain model into its corresponding view model, making label

management decisions based on information such as user preferences, the application's current state (including possible simulation runs), the task currently pursued by the user, and selections in other views or editors.

To keep labels from being shortened beyond the requirements of a compact layout, the feedback loop approach only required that label management be configured during view model generation, but deferred its execution to automatic layout. Since layout algorithms know how long labels can become before they start affecting their diagram's size, they can supply a target width for label management to aim for. Interestingly, this changes the role of automatic layout: feeding information back into label management allows for influencing not only the position of elements, but also the amount of detail they display.

We evaluated label management through several case studies. One involved ELK Sequence, a new layout algorithm for UML sequence diagrams, which introduced different lifeline ordering strategies as well as support for reducing the height of sequence diagrams by allowing messages to share y coordinates. Our aesthetic evaluation showed that label management successfully reduces the width of diagrams and that doing so does not necessarily have to entail losing information. Surveys among users of the SCCharts editing environment, finally, resulted in positive feedback.

A general label management framework is available to any layout algorithm based on ELK and is fully supported by ELK Layered and ELK Sequence. The KLighD project implements support for label management along with several label management strategies, which are used by the KIELER Ptolemy Browser as well as the SCCharts and KieSL editing environments.

7.2 Lessons Learned

Several years of involvement in research on automatic layout and in an active open source project that aimed at putting the fruits of said research into practice yielded the following insights.

Academic Software Development As my esteemed former colleague, Miro Spönemann, put it: "A software project in an academic context is

subject to very different goals, requirements, and conditions compared to one in a business context. The top priority is on innovation, and not on sales." [Spö15, Section 7.2]. Never has this been more apparent at our group than since we decided that the Eclipse Foundation should be home to our layout infrastructure and algorithms. Whether or not doing so has increased the project's visibility is hard to measure (at times it seems that our most successful contribution to mankind so far is ELK's JavaScript library). What is certainly true is that it has made it easier for companies to incorporate ELK into their products due to the legal safety provided by the Eclipse Foundation.

The process did, however, come at a significant cost: we had to invest a substantial amount of work to make our existing source code conform to the Foundation's requirements. Granted, this turned out to be the perfect opportunity to change aspects of the framework that we had wanted to change for a while, but never had the time to; examples are the major overhaul of the graph data structure or the introduction of MELK files to consolidate meta data. It also meant, however, that our research had to be put on hold for a while, and was slow to recover since migrating the layout code had to be followed by adapting the rest of our tool chain as well.

Whether the return on investment, so to speak, is high enough to take on such a substantial endeavor depends on the project. Personally, I am rather fond of the fact that our accomplishments are available as part of an established software project backed by industry, for developers around the world to use (not to mention the fact that this might also serve as a small contribution towards increasing Kiel University's visibility).

Students generally share the sentiment, but there is a caveat: by the time they work on their Bachelor's or even on their Master's thesis, the nature of how software development skills are acquired entails that most students produce code that is not quite up to par with professional standards yet. To be fair, this is usually not the fault of the students: most simply have not had enough time yet to hone their craft. Through feedback of some of them, we realized that we had to take this into account. True to the open source ideals, all of our development efforts had previously taken place in publicly accessible repositories, when we should perhaps have offered a closed safe space to students, only to be left after the final code review.

On Users and Perception While aesthetic criteria present useful forays into measuring the quality of graph layouts, they fail to cover many aspects of that quality. A case in point is secondary notation, which is not only hard to measure, but also seemingly impossible to agree upon by actual users in the first place [Pet95].

Being computer scientists, we like concepts that we can measure objectively: a drawing with six edge crossings remains a drawing with six edge crossings, regardless of who looks at it. The problems we run into when we apply automatic layout in practice, however, often defy clear measurements, let alone definitions, as the mere existence of the term "Obviously Non-optimal" shows.

Recall directional label side placement, introduced back in Section 3.3.4. With properly chosen spacings, it seems perfectly sensible to assume that it provides graphic association as clear as that achieved by the same-side strategy while encoding additional information in an unobtrusive way. What sounds like a valuable strategy did not survive first contact with users, who found it confusing on quite a number of different levels.

Professional software companies, at least the ones eager to come up with good user interface designs, often take advantage of a number of informal techniques such as hallway usability testing (showing a design mockup to random people to identify severe usability problems before time is wasted on implementing them). The scientific method, however, calls for more rigorous approaches, and rightly so: the aim, after all, is not to increase profit, but to generate conclusive evidence and insights regarding what works and what does not, and why.

It is for these reasons that we have slowly begun shifting from evaluations based purely on aesthetics to conducting controlled experiments and gathering feedback by real users. These methods, however, present their own challenges. Controlled experiments are a lot of work, are hard to get right, and are often rather limited in their conclusions and their generalizability. Real users are rarely available in a controlled setting; our group at least has the good fortune of having students that use our software.

While I remain convinced that seeking the feedback of real users is paramount, it may just be that a shift of perspective could offer a valuable supplement: new developments could or perhaps should be influenced by

insights from perceptual theories. Gestalt psychology, for instance, has over a century of research behind it. Applying perceptual theories to computer science is not a new idea: previous researchers have done so with interesting results, for instance Wong and Sun [WS05] or Ware et al. [WPC+02].

7.3 Open Problems

Rarely is there enough time to look at all the interesting problems that present themselves over the course of one's research.

Layout Support for Text From a theoretical perspective, it would be worthwhile to determine whether the MinWidthLabelAssignment problem actually is NP-complete or not. Answering that question would be particularly interesting since the problem is extremely similar to other NP-complete problems, but not so much so that obvious reductions present themselves.

While the micro layout of nodes often enough simply needs to follow the requirements of a given visual language, edge label placement still poses interesting challenges. First, while end layer assignment strategies as well as the median layer strategy result in predictable label placements, this is not the case for layer assignment strategies that optimize for diagram width. This may or may not have a negative impact on diagram understanding since users may or may not have a hard time finding labels of interest to them. Further user studies are necessary to answer this question.

Second, it would be interesting to further investigate the impact on-edge label placement has on the ability of users to follow edges through diagrams. Do different label designs have an effect? Is there a difference between single- and multi-line labels? What about differences between horizontal and vertical layouts?

Finally, there is a point to be made about whether having layout algorithms place edge labels is a good idea in the first place. A label thusly placed will always stay at its assigned position, at least until the next layout run. If a user zooms into the diagram, the label of an edge visible in the viewport may well be placed beyond the viewport's boundaries. Applications such as Google Maps or OpenStreetMap compute new label positions

as the viewport changes. A promising research direction would be to apply map labeling techniques to computing ad-hoc edge label positions for edges whose labels are off screen.

Text and Secondary Notation The comment attachment pipeline we have discussed in Chapter 5 does not cover all types of specific comments yet. Instead of being limited to nodes, comments often describe ports and edges as well. This is not yet supported by the attachment pipeline, mainly because placing comments accordingly is not supported by ELK Layered. Finding ways of doing so would be worthwhile since the wish of associating comments with edges is one that has also been voiced by the SCCharts community.

The heuristics themselves could be improved as well, be it through improved text parsing to understand which node a comment's text actually refers to or through adding support for comments that relate not to one, but to several nodes. Recognizing such group comments appears to be a proper challenge, however, due to the problem's fuzzy nature.

More profoundly, while the notion of secondary notation is not new, a proper theory of how it is employed by users is still lacking. It will be necessary to look at many more visual languages and to obtain diagrams produced both by novices and experts. Such a theory could provide a foundation for automatic layout algorithms to build on, not just for proper comment placement, but also to recognize and preserve other kinds of secondary notation.

One obstacle for this kind of research, however, is actually obtaining access to expert users who, working in the industry, are usually tied up with work that benefits their company directly instead of being allowed to support science which may or may not pay off in the long-term.

Unwieldy Amounts of Text Label management suggests obvious ways for further research, adding more label managers and integration into micro layout being two obvious examples.

A more difficult venue for further research concerns the target width computed by layout algorithms. As we have seen in Section 6.2.4, computing

target widths based on layer widths in ELK Layered is a natural way to do so, but only works for horizontal layout directions. In vertical layouts, label width impacts the node placement phase. Integrating support for computing meaningful target widths into node placement algorithms would make label management more useful in vertical layouts. Another aspect is that of integrating label management into additional layout approaches. Again, this will mainly entail finding natural ways of computing target widths.

As described in Chapter 6, label management can only influence the text in a diagram, but there is no reason why it should not be given control over the presentation of diagram elements as well—after all, the way elements looks has a profound impact on their size. This starts with the choice of font sizes, but can also include removing graphical detail.

Interesting results could also come out of looking for more ways of interacting with label management, both concerning its configuration and its integration into the diagram browsing process. Allowing users to customize label management, perhaps even specifying new label management strategies, runs the risk of becoming too complex, but limited customization might be viable. Perhaps more interesting, however, would be to find further ways for determining whether to shorten a label and how to display a shortened label's full text, and even to combine label management with more flexible, viewport-based edge label placement.

7.4 Conclusion

This thesis was all about text in diagrams as it relates to automatic layout algorithms, and if there is one thing at the core of that topic it is that proper support for text can be complex, but is very necessary. Interesting problems lurk around every corner, and we have looked at quite a few throughout the four main chapters: from properly computing node sizes over how to display edge labels to managing the amount of text in diagrams.

For users to accept and use layout algorithms, experience shows that it is necessary to get the details right—including text. Getting to that point poses quite a number of challenges, which the first part of this thesis was

all about. Once these have been met, layout algorithms can be used to improve the pragmatics of how users interact with diagrams, which is what the second part was concerned with. Besides improving the way we can browse through hierarchical diagrams or dynamically adapting the text in a diagram, automatic layout also gives us the opportunity to synthesize diagrams geared specifically at supporting users in their current task. For example, we can provide graphical representations of complex data structures when debugging applications, as the *DebuKViz* project[1] does.

Personally, my hope is that the research contributions presented in this thesis will help add interactive diagrammatic views to all kinds of applications, in order to always provide exactly those visualizations which are most helpful, with as much detail as necessary, but as little as possible.

[1] https://github.com/OpenKieler/debukviz

Appendix A

Sample Diagrams

This appendix contains samples from the diagrams used for some of the experiments presented throughout this thesis.

A.1 SCCharts

A.1.1 Edge Label Placement

The following diagrams are samples from the set of diagrams used for the evaluations in Section 3.4.1 and Section 3.4.2. These are some of the smallest specimen since larger ones cannot be sensibly reproduced here.

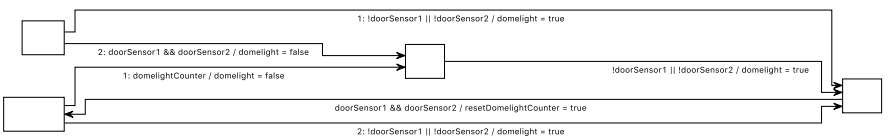


Figure A.1. A diagram with a horizontal layout and consistent label side selection.

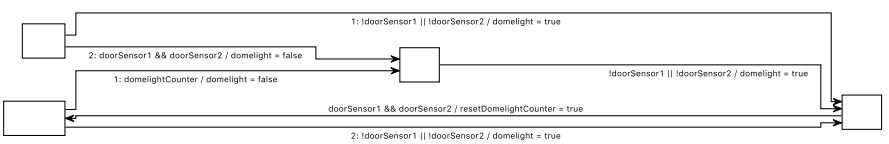


Figure A.2. A diagram with a horizontal layout and directional label side selection.

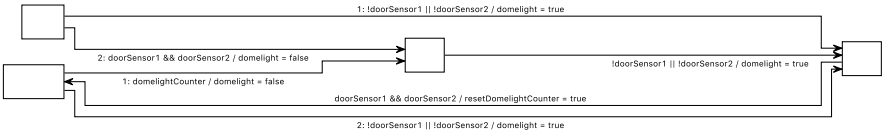


Figure A.3. A diagram with a horizontal layout and smart label side selection.

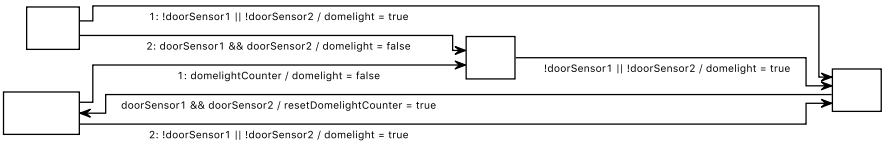


Figure A.4. A diagram with a horizontal layout, consistent label side selection, and the space-efficient layer selection. Note how this drawing is smaller than the previous ones (or rather, the zoom level is larger).

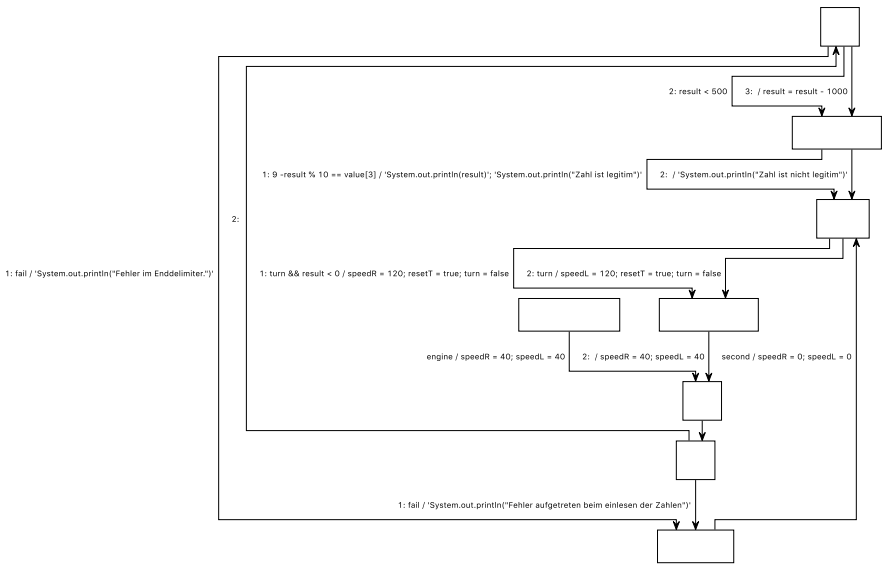


Figure A.5. A diagram with a vertical layout and consistent label side selection.

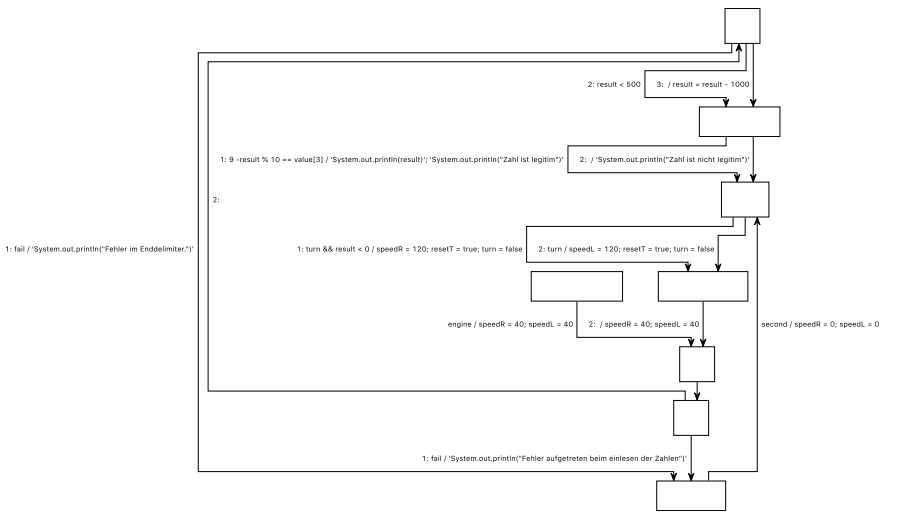


Figure A.6. A diagram with a vertical layout and directional label side selection.

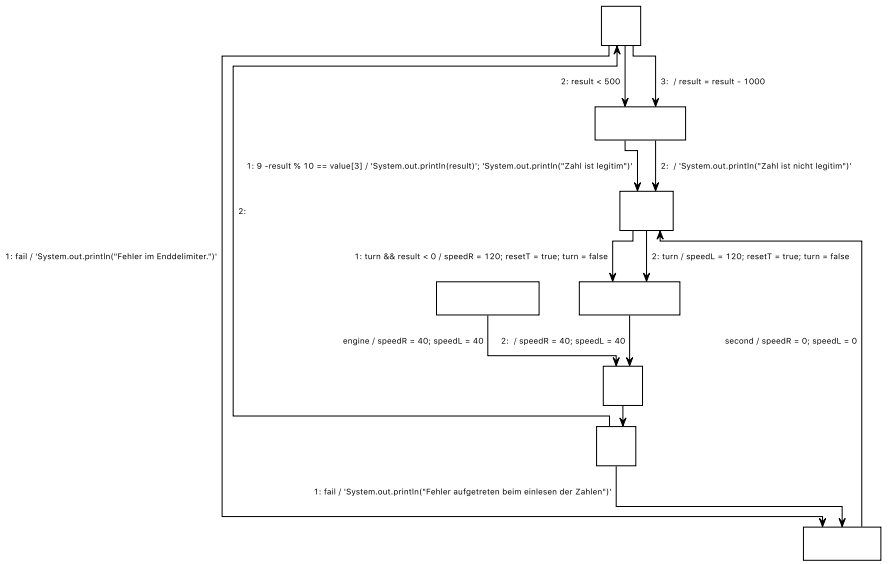


Figure A.7. A diagram with a vertical layout and smart label side selection. Note how in this drawing, multiple edges between the same two nodes are routed close to each other.

A.1.2 Label Management

The following diagrams are samples from the set of diagrams used for the evaluation in Section 6.4.2. As in Section A.1.1, these are some of the smallest specimen since larger ones cannot be sensibly reproduced here.

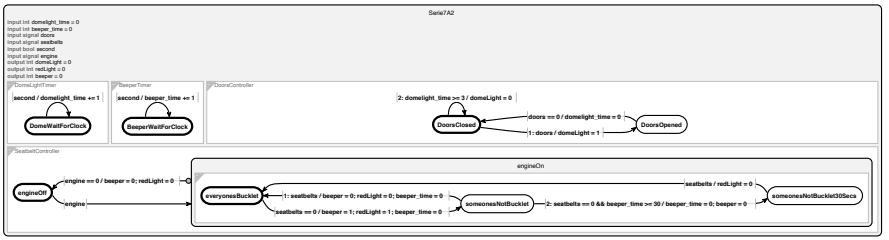


Figure A.8. SCChart "Serie7A2" with original labels.

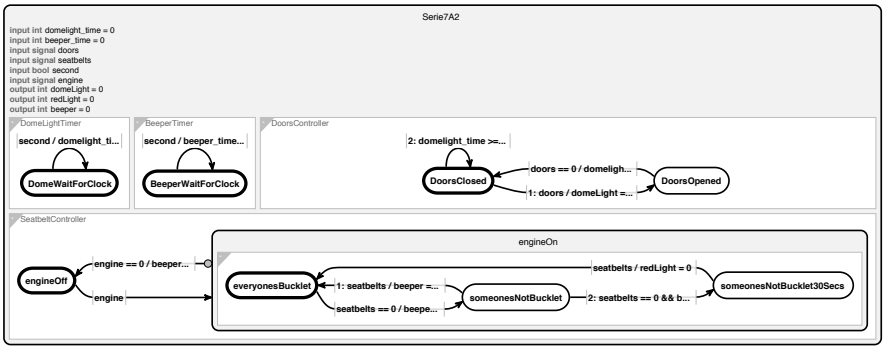


Figure A.9. SCChart "Serie7A2" with labels truncated to 130 pixels.

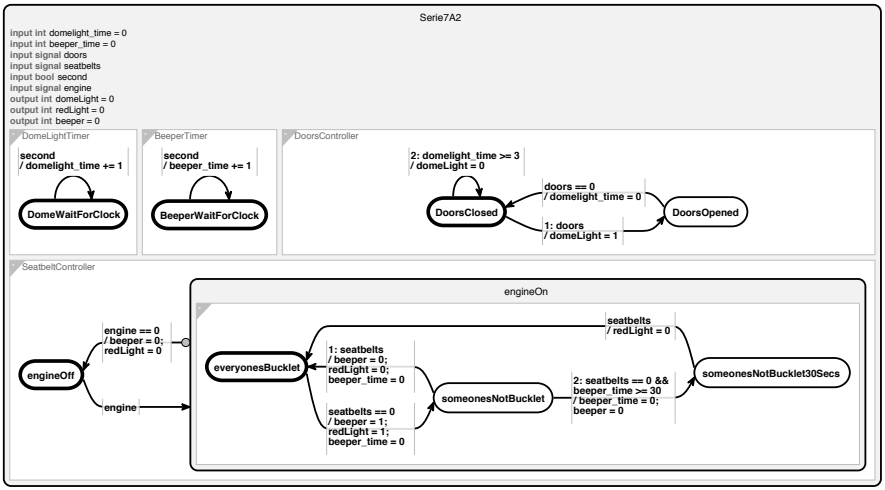


Figure A.10. SCChart "Serie7A2" with semantic line wrapping applied at 130 pixels.

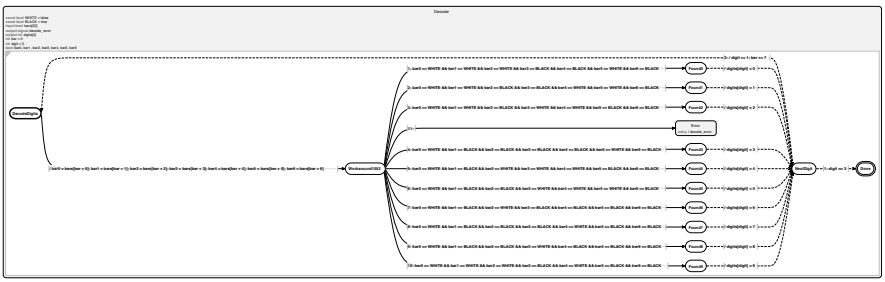


Figure A.11. SCChart "Decoder" with original labels.

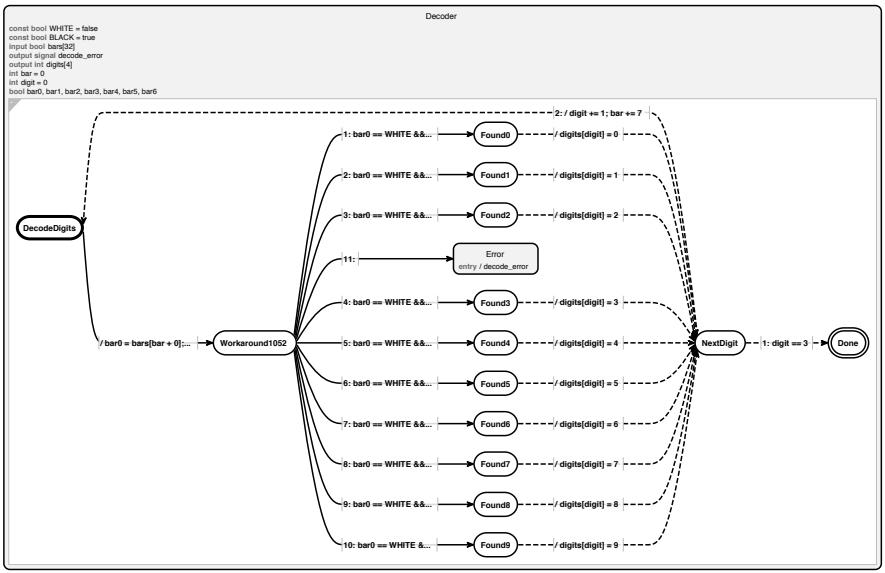


Figure A.12. SCChart "Decoder" with labels truncated to 130 pixels.

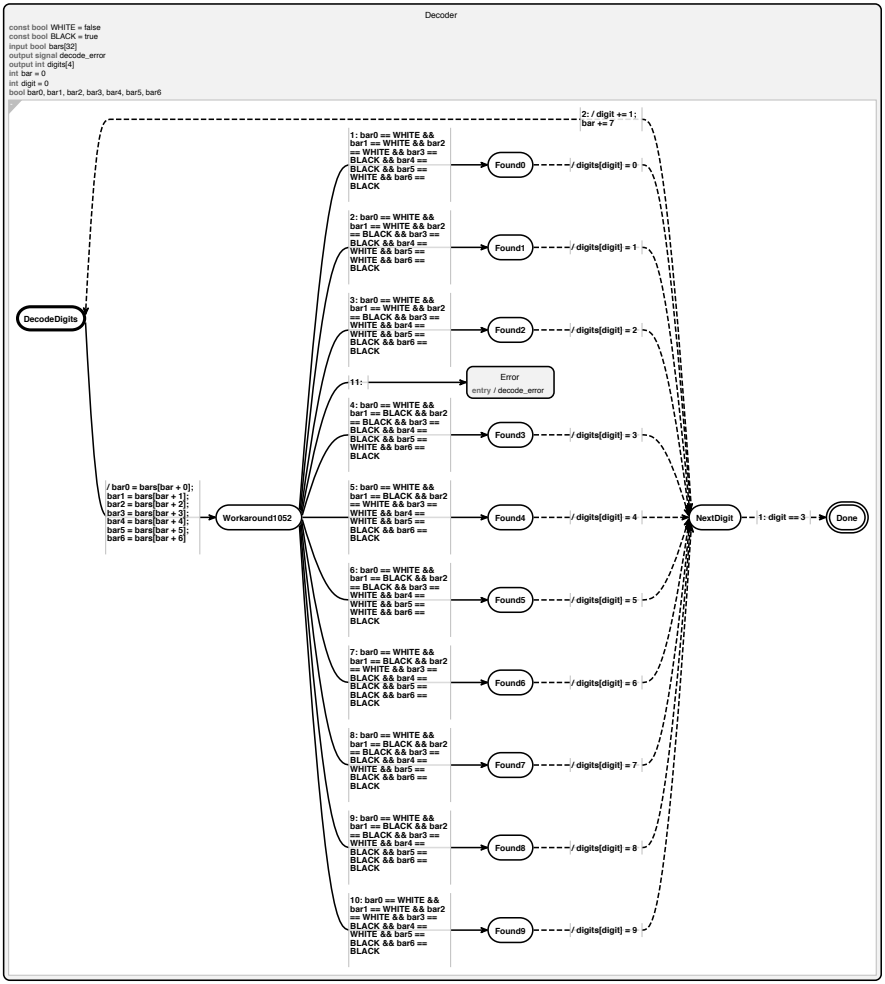


Figure A.13. SCChart "Decoder" with semantic line wrapping applied at 130 pixels. Note how this increases the height of the diagram considerably, which may or may not be okay.

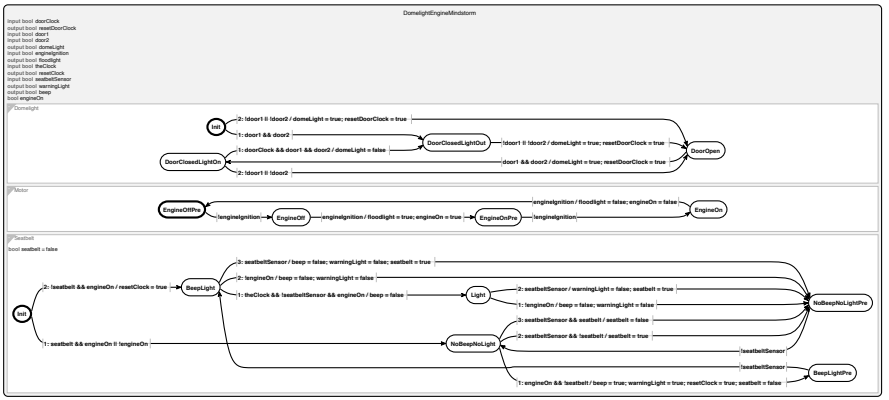


Figure A.14. SCChart "DomelightEngineMindstorm" with original labels.

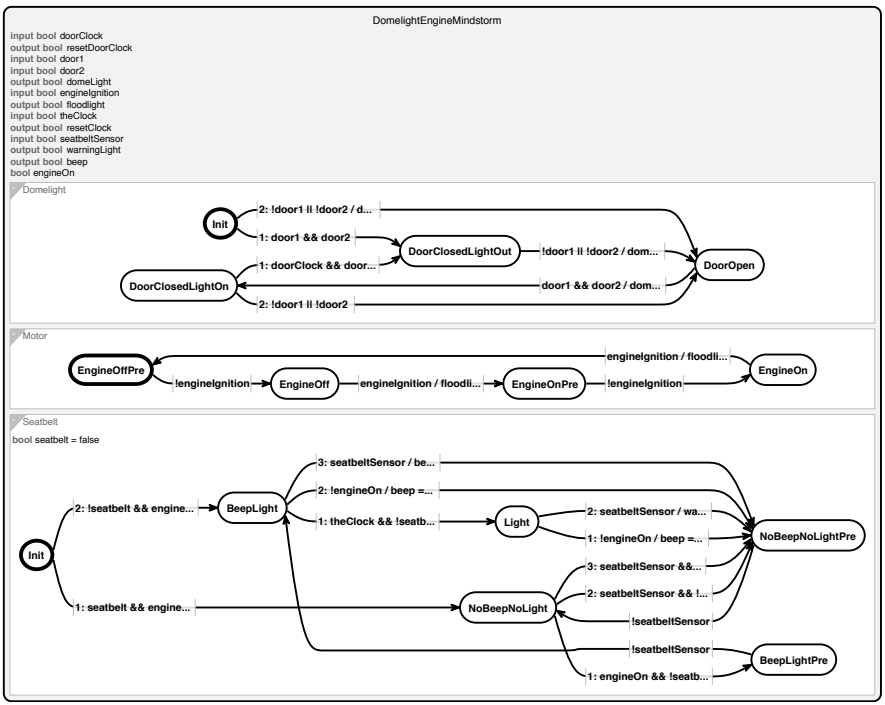


Figure A.15. SCChart "DomelightEngineMindstorm" with labels truncated to 130 pixels.

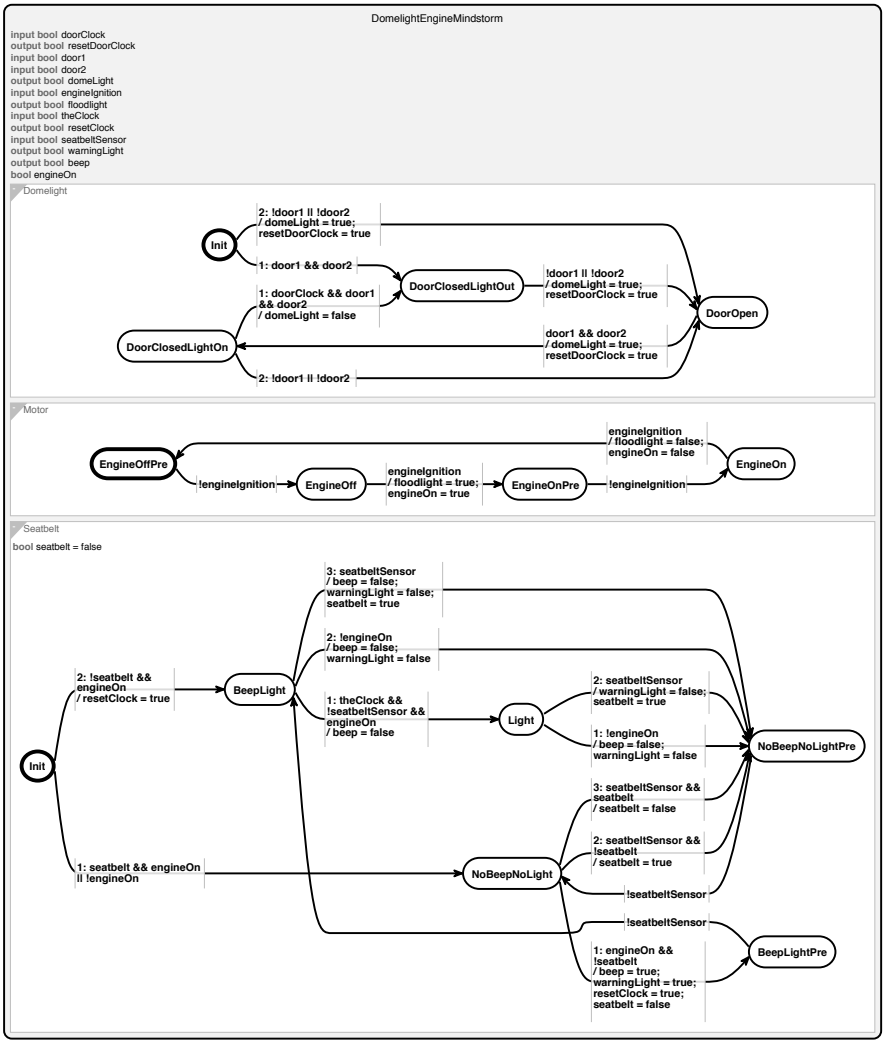


Figure A.16. SCChart "DomelightEngineMindstorm" with semantic line wrapping applied at 130 pixels.

A.2 UML Sequence Diagrams

The following diagrams are samples from the set of diagrams used for the evaluations in Section 4.4 and Section 6.4.2.

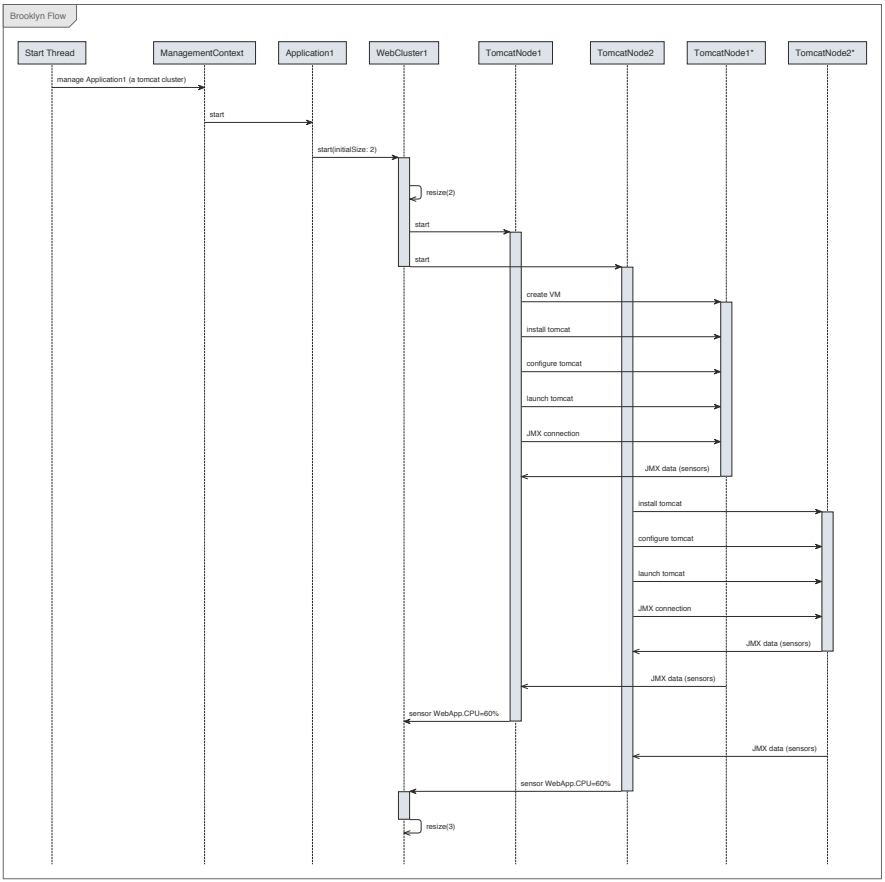


Figure A.17. Sequence diagram "Brooklyn Flow" with lifelines ordered according to the developer's original order.

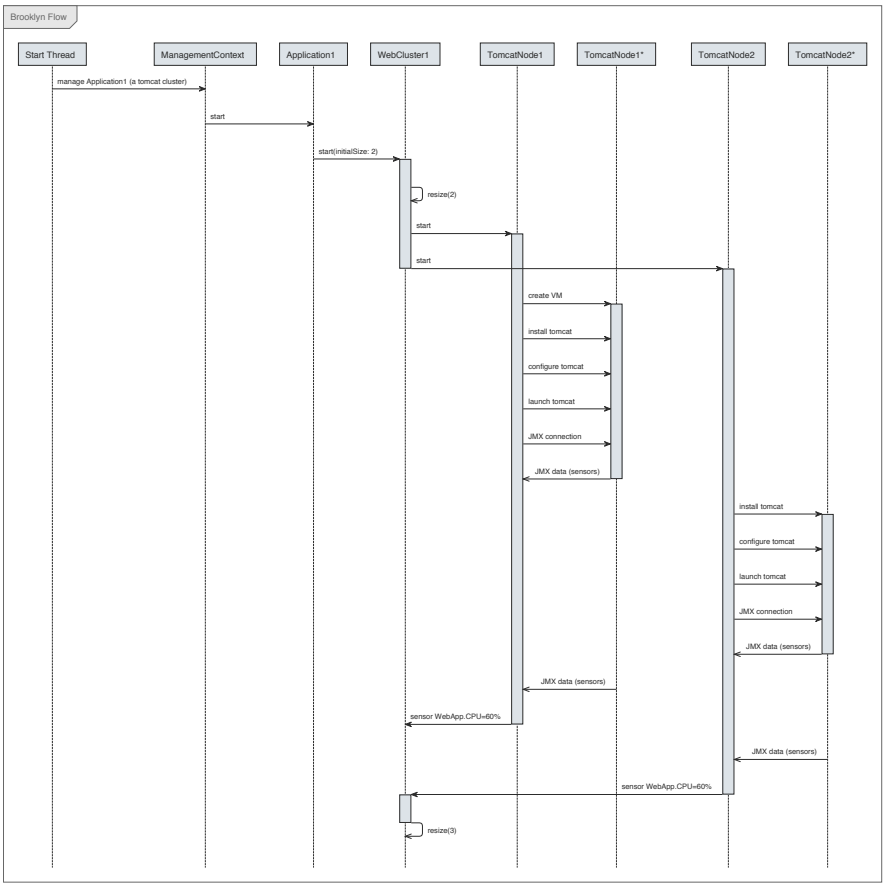


Figure A.18. Sequence diagram "Brooklyn Flow" with lifelines ordered according to the communication line order algorithm.

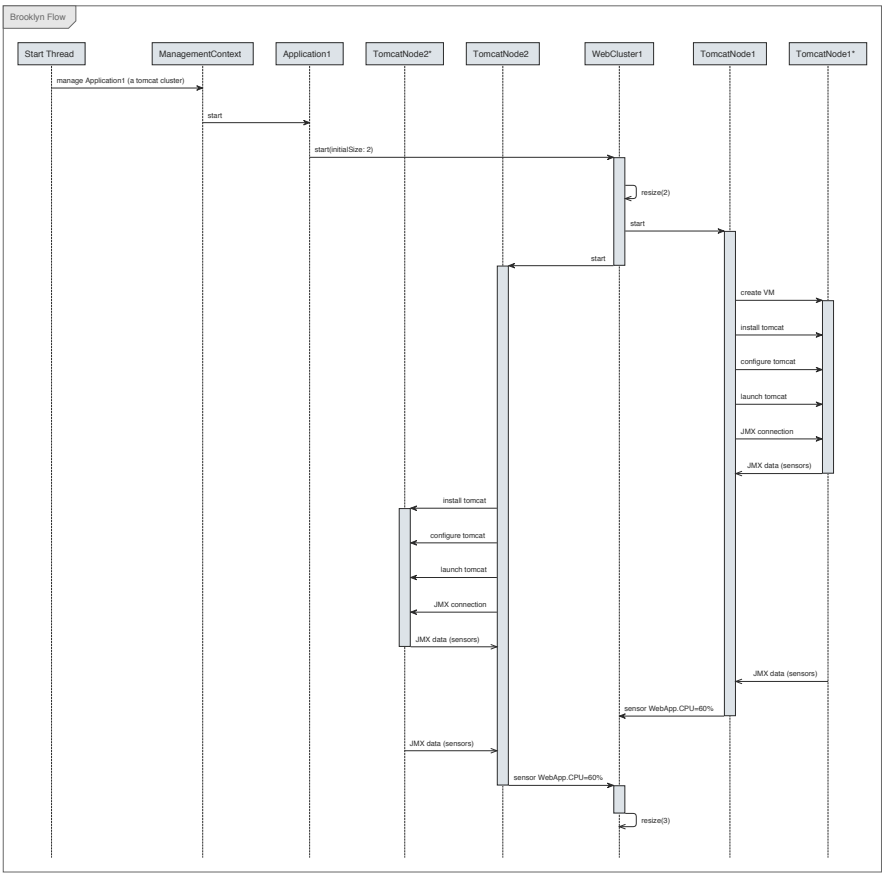


Figure A.19. Sequence diagram "Brooklyn Flow" with lifelines ordered according to the short message order algorithm.

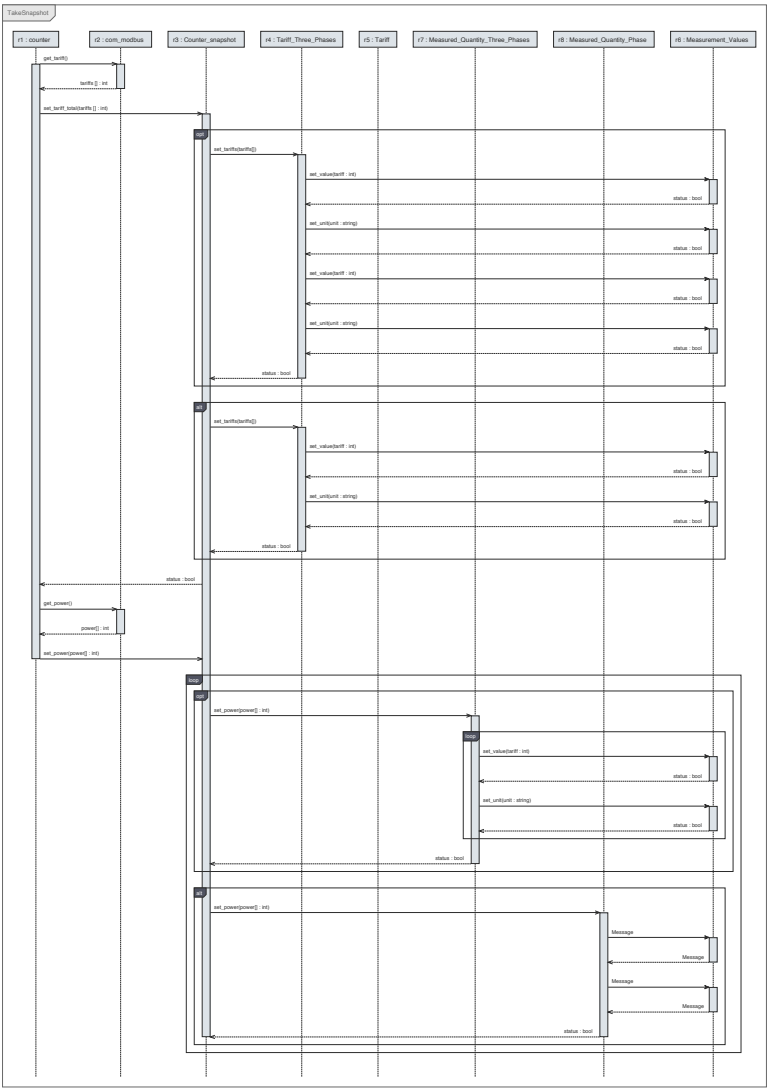


Figure A.20. Sequence diagram "TakeSnapshot" with lifelines ordered according to the developer's original order.

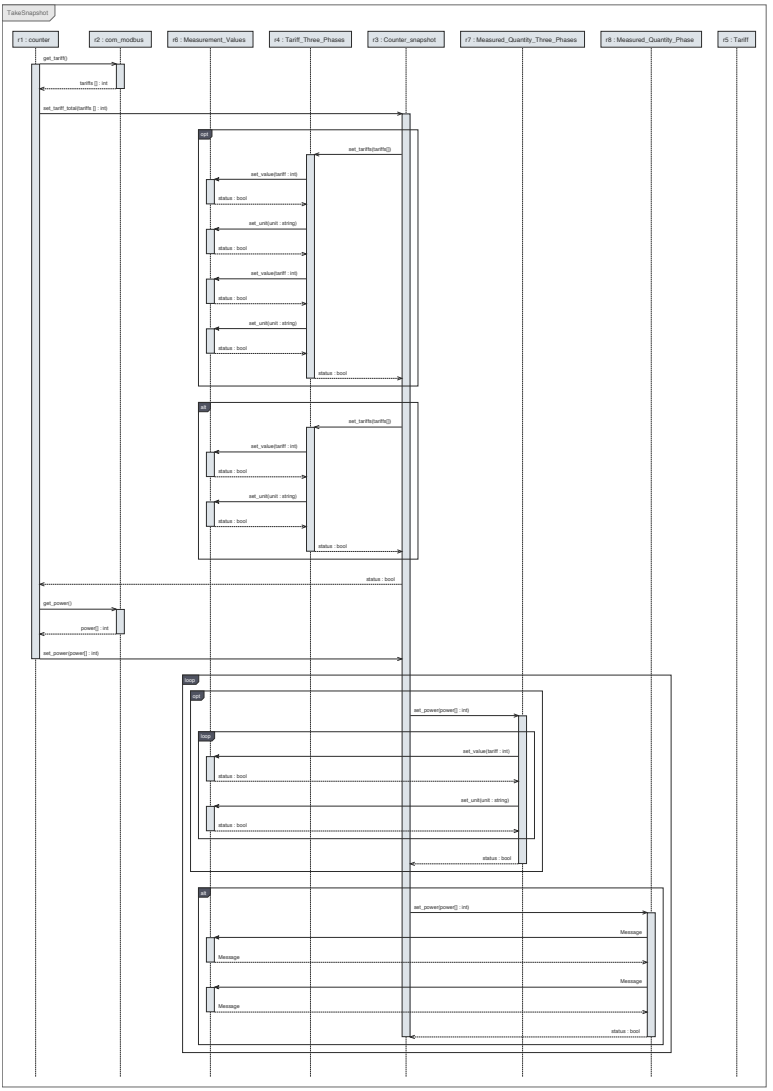


Figure A.21. Sequence diagram “TakeSnapshot” with lifelines ordered according to the communication line order algorithm.

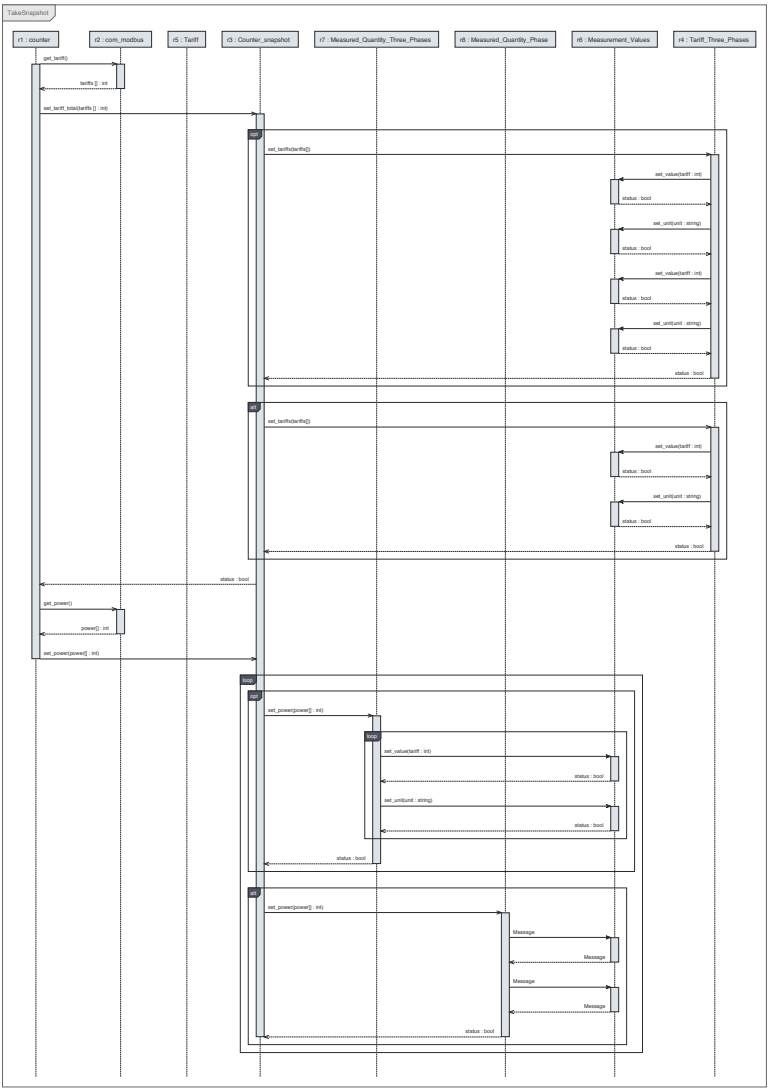


Figure A.22. Sequence diagram "TakeSnapshot" with lifelines ordered according to the short message order algorithm.

Bibliography

[Bab02] Danil E. Baburin. "Using graph based representations in reengineering". In: *Proceedings of the Sixth European Conference on Software Maintenance and Reengineering* (2002), pp. 203–206. DOI: 10.1109/CSMR.2002.995805.

[BBB+95] Margaret M. Burnett, Marla J. Baker, Carisa Bohus, Paul Carlson, Sherry Yang, and Pieter van Zee. "Scaling up visual programming languages". In: *IEEE Computer* 28.3 (Mar. 1995), pp. 45–54. DOI: 10.1109/2.366157. URL: http://dx.doi.org/10.1109/2.366157.

[BDY06] Ken Been, Eli Daiches, and Chee Yap. "Dynamic map labeling". In: *IEEE Transactions on Visualization and Computer Graphics* 12.5 (Sept. 2006), pp. 773–780. DOI: 10.1109/TVCG.2006.136.

[Beg96] Andrew Begel. *Logoblocks: a graphical programming language for interacting with the world*. Tech. rep. MIT Media Laboratory, May 1996. URL: http://www.msr-waypoint.com/en-us/um/people/abegel/mit/begel-aup.pdf.

[BGM04] Benjamin B. Bederson, Jesse Grosjean, and Jon Meyer. "Toolkit design for interactive structured graphics". In: *IEEE Transactions on Software Engineering* 30.8 (Aug. 2004), pp. 535–546.

[BH11] Franz J. Brandenburg and Kathrin Hanauer. *Sorting heuristics for the feedback arc set problem*. Tech. rep. MIP-1104. Department of Informatics and Mathematics, University of Passau, Feb. 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)*. Ed. by Joe Marks. Vol. 1984. LNCS. Springer, 2001, pp. 229–240. ISBN: 978-3-540-41554-1. DOI: 10.1007/3-540-44541-2.

[BJM02] Wilhelm Barth, Michael Jünger, and Petra Mutzel. "Simple and efficient bilayer cross counting". In: *Proceedings of the 10th International Symposium on Graph Drawing (GD '02)*. Ed. by Stephen G. Kobourov and Michael T. Goodrich. Vol. 2528. LNCS. Springer, 2002, pp. 331–360.

[BK02] Ulrik Brandes and Boris Köpf. "Fast and simple horizontal coordinate assignment". In: *Proceedings of the 9th International Symposium on Graph Drawing (GD '01)*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Vol. 2265. LNCS. Springer, 2002, pp. 33–36. ISBN: 978-3-540-43309-5. DOI: 10.1007/3-540-45848-4.

[Bla96] Alan F. Blackwell. "Metacognitive theories of visual programming: what do we think we are doing?" In: *Proceedings of the 1996 IEEE Symposium on Visual Languages*. 1996, pp. 240–246. DOI: 10.1109/VL.1996.545293. URL: http://dx.doi.org/10.1109/VL.1996.545293.

[BLM+01] Jürgen Branke, Stefan Leppert, Martin Middendorf, and Peter Eades. "Width-restricted layering of acyclic digraphs with consideration of dummy nodes". In: 403 (Sept. 2001), pp. 59–63.

[BMM04] Gary Bist, Neil MacKinnon, and Steve Murphy. "Sequence diagram presentation in technical documentation". In: *Proceedings of the 22nd Annual International Conference on Design of Communication: The Engineering of Quality Documentation (SIGDOC '04)*. Memphis, Tennessee, USA: ACM, 2004, pp. 128–133. ISBN: 1-58113-809-1. DOI: 10.1145/1026533.1026566.

[BMS+08] C. Bennett, D. Myers, M.-A. Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland. "A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams". In: *Journal of Software Maintenance and Evolution: Research and Practice* 20.4 (July 2008), pp. 291–315. DOI: 10.1002/smr.372.

[Bou86] Benedict Du Boulay. “Some difficulties of learning to program”. In: *Journal of Educational Computing Research* 2.1 (1986), pp. 57–73.

[Bro87] Frederick P. Brooks Jr. “No silver bullet: essence and accidents of software engineering”. In: *Computer* 20.4 (1987), pp. 10–19. ISSN: 0018-9162.

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

[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)*. SIAM, 1990, pp. 236–243. ISBN: 0-89871-251-3.

[BW08] Raymond P.L. Buse and Westley R. Weimer. “Automatic documentation inference for exceptions”. In: *Proceedings of the 2008 International Symposium on Software Testing and Analysis*. ISSTA '08. ACM, 2008, pp. 273–282. URL: http://doi.acm.org/10.1145/1390630.1390664.

[BW97] Ulrik Brandes and Dorothea Wagner. “A bayesian paradigm for dynamic graph layout”. In: *Proceedings of the 5th International Symposium on Graph Drawing (GD '97)*. Vol. 1353. LNCS. Springer, 1997, pp. 236–247. ISBN: 3-540-63938-1.

[Car12] John Julian Carstens. “Node and label placement in a layered layout algorithm”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jjc-mt.pdf. Master's thesis. Kiel University, Department of Computer Science, Sept. 2012.

[CG72] Edward G. Coffman. and Ronald L. Graham. “Optimal scheduling for two-processor systems”. In: *Acta Informatica* 1.3 (1972), pp. 200–213. ISSN: 0001-5903. DOI: 10.1007/BF00288685.

[CKB08] Andy Cockburn, Amy K. Karlson, and Benjamin B. Bederson. "A review of overview+detail, zooming, and focus+context interfaces". In: *ACM Comput. Surv.* 41.1 (2008), 2:1–2:31. DOI: 10.1145/1456650.1456652.

[CKB09] Andy Cockburn, Amy Karlson, and Benjamin B. Bederson. "A review of overview+detail, zooming, and focus+context interfaces". In: *ACM Computing Surveys* 41 (1 2009), 2:1–2:31. ISSN: 0360-0300. DOI: http://doi.acm.org/10.1145/1456650.1456652.

[CMS99] Stuart K. Card, Jock Mackinlay, and Ben Shneiderman. *Readings in information visualization: using vision to think*. Morgan Kaufmann, Jan. 1999. ISBN: 1558605339.

[CMT01] R. Castelló, R. Mili, and I. G. Tollis. "An algorithmic framework for visualizing Statecharts". In: *GD 2000: Proceedings of the 8th International Symposium on Graph Drawing*. Vol. 1984. LNCS. Springer-Verlag, 2001, pp. 43–44. ISBN: 978-3-540-41554-1.

[DET+99] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph drawing: Algorithms for the visualization of graphs*. Prentice Hall, 1999. ISBN: 0-13-301615-3.

[Ead84] Peter Eades. "A heuristic for graph drawing". In: *Congressus Numerantium* 42 (1984), pp. 149–160. ISSN: 0384-9864.

[ECM+96] Shawn Edmondson, Jon Christensen, Joe Marks, and Stuart Shieber. "A general cartographic labelling algorithm". In: *Cartographica: The International Journal for Geographic Information and Geovisualization* 33.4 (1996), pp. 13–24. DOI: 10.3138/U3N2-6363-130N-H870.

[EGB03] Thomas Eschbach, Wolfgang Guenther, and Bernd Becker. "Crossing reduction for orthogonal circuit visualization". In: *Proceedings of the 2003 International Conference on VLSI*. CSREA Press, 2003, pp. 107–113.

[ELS93] Peter Eades, Xuemin Lin, and W. F. Smyth. "A fast and effective heuristic for the feedback arc set problem". In: *Information Processing Letters* 47.6 (1993), pp. 319–323. ISSN: 0020-0190. DOI: 10.1016/0020-0190(93)90079-O.

[EMW86] Peter Eades, Brendan D. McKay, and Nicholas C. Wormald. "On an edge crossing problem". In: *9th Australian Computer Science Conference*. ACA. Australia, 1986, pp. 327–334.

[ES90] Peter Eades and Kozo Sugiyama. "How to draw a directed graph". In: *Journal of Information Processing* 13.4 (1990), pp. 424–437. ISSN: 0387-6101.

[EW86] Peter Eades and Nicholas C. Wormald. *The median heuristic for drawing 2-layered networks*. Tech. rep. 69. Department of Computer Science: University of Queensland, 1986.

[FBH15] Insa Fuhrmann, David Broman, and Reinhard von Hanxleden. *Interactive timing analysis for designing reactive systems*. Presentation at the 22nd International Open Workshop on Synchronous Programming (SYNCHRON '15), Kiel, Germany. Dec. 2015.

[FH10] 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)*. Vol. 6394. LNCS. Springer, Oct. 2010, pp. 196–210. DOI: 10.1007/978-3-642-16145-2.

[Flo13] Luciano Floridi. "Technology's in-betweeness". In: *Philosophy & Technology* 26.2 (June 2013), pp. 111–115. ISSN: 2210-5441. DOI: 10.1007/s13347-013-0106-y.

[For02] Michael Forster. "Applying crossing reduction strategies to layered compound graphs". In: *Proceedings of the 10th International Symposium on Graph Drawing (GD '02)*. Ed. by Stephen G. Kobourov and Michael T. Goodrich. Vol. 2528. LNCS. Springer, 2002, pp. 115–132. ISBN: 978-3-540-00158-4. DOI: 10.1007/3-540-36151-0.

[FS04] Carsten Friedrich and Falk Schreiber. "Flexible layering in hierarchical drawings with nodes of arbitrary size". In: *Proceedings of the 27th Australasian Conference on Computer Science (ACSC'04)*. Australian Computer Society, Inc., 2004, pp. 369–376.

[Fuh11] Hauke Fuhrmann. “On the pragmatics of graphical modeling”. Dissertation. Kiel: Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, 2011.

[Fuh12] Insa Fuhrmann. “Layout of compound graphs”. Diploma thesis. Kiel University, Department of Computer Science, Feb. 2012.

[GG21] Frank Bunker Gilbreth and Lillian Moller Gilbreth. “Process charts”. In: *Proceedings of the Annual Meeting of The American Society of Mechanical Engineers*. 29 West 39th Street, New York: The American Society of Mechanical Engineers, Dec. 1921.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and intractibility: a guide to the theory of NP-completeness*. New York: W. H. Freeman & Co, 1979.

[GJ83] Michael R. Garey and David S. Johnson. “Crossing number is NP-complete”. In: *SIAM Journal on Algebraic and Discrete Methods* 4.3 (1983), pp. 312–316. DOI: 10.1137/0604033.

[GKN+93] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. “A technique for drawing directed graphs”. In: *Software Engineering* 19.3 (1993), pp. 214–230.

[GNV88] Emden R. Gansner, Steven C. North, and Kiem-Phong Vo. “DAG — a program that draws directed graphs”. In: *Software: Practice and Experience* 18.11 (1988), pp. 1047–1062. ISSN: 1097-024X. DOI: 10.1002/spe.4380181104.

[Gol04] Martin Golumbic. *Algorithmic graph theory and perfect graphs*. 2nd ed. Vol. 57. Annals of Discrete Mathematics. Elsevier, 2004. ISBN: 9780444515308.

[GP96] T. R. G. Green and M. Petre. “Usability analysis of visual programming environments: A ‘cognitive dimensions’ framework”. In: *J. Visual Languages and Computing* 7.2 (June 1996), pp. 131–174. ISSN: 1045-926X.

[Gri16] Lena Grimm. “Debugging SCCharts”. http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/lgr-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2016.

[Har87] David Harel. “Statecharts: A visual formalism for complex systems”. In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274.

[Har88] David Harel. “On visual formalisms”. In: *Communications of the ACM* 31.5 (1988). http://doi.acm.org/10.1145/42411.42414, pp. 514–530. ISSN: 0001-0782.

[HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. “SCCharts: Sequentially Constructive Statecharts for safety-critical applications”. In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14)*. Edinburgh, UK: ACM, June 2014, pp. 372–383.

[HF16] Austin Z. Henley and Scott D. Fleming. “Yestercode: improving code-change support in visual dataflow programming environments”. In: *IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC’16)*. Sept. 2016, pp. 106–114. DOI: 10.1109/VLHCC.2016.7739672.

[HIW+11] Danny Holten, Petra Isenberg, Jarke J. van Wijk, and Jean-Daniel Fekete. “An extended evaluation of the readability of tapered, animated, and textured directed-edge representations in node-link graphs”. In: *2011 IEEE Pacific Visualization Symposium*. Mar. 2011, pp. 195–202. DOI: 10.1109/PACIFICVIS.2011.5742390.

[HMA+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)*. Grenoble, France: IEEE, Mar. 2013, pp. 581–586.

[HN02a] Patrick Healy and Nikola S. Nikolov. “A branch-and-cut approach to the directed acyclic graph layering problem”. In: *Proceedings of the 10th International Symposium on Graph Draw-*

ing (GD '02). Ed. by Stephen G. Kobourov and Michael T. Goodrich. Vol. 2528. LNCS. Springer, 2002, pp. 98–109. ISBN: 978-3-540-00158-4. DOI: 10.1007/3-540-36151-0.

[HN02b] 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)*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Vol. 2265. LNCS. Springer, 2002, pp. 16–30. DOI: 10.1007/3-540-45848-4_2.

[Hoo13] Gregor Hoops. "Automatic layout of UML sequence diagrams". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/grh-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, Apr. 2013.

[HQC+16] Regina Hebig, Truong Ho Quang, Michel R. V. Chaudron, Gregorio Robles, and Miguel Angel Fernandez. "The quest for open source projects that use UML: mining GitHub". In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS '16)*. 2016, pp. 173–183. DOI: 10.1145/2976767.2976778.

[HW09] Danny Holten and Jarke J. van Wijk. "A user study on visualizing directed edges in graphs". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. New York, NY, USA: ACM, Apr. 2009, pp. 2299–2308. ISBN: 978-1-60558-246-7. DOI: 10.1145/1518701.1519054.

[Imh75] Eduard Imhof. "Positioning names on maps". In: *The American Cartographer* 2.2 (1975), pp. 128–144. DOI: 10.1559/152304075784313304.

[Jah15] Daniel Jahn. "Eine textuelle Sprache zum automatischen Generieren von Sequenzdiagrammen". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/dja-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2015.

[Jam86] Geoffrey James. *The tao of programming*. Info Books, 1986. ISBN: 978-0931137075.

[JLM+97] 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)*. Vol. 1353. LNCS. Springer, 1997, pp. 13–24. ISBN: 978-3-540-63938-1. DOI: 10.1007/3-540-63938-1.

[Kar72] Richard M. Karp. "Reducibility among combinatorial problems". In: *Complexity of Computer Computations (Proceedings of a Symposium on the Complexity of Computer Computations, March, 1972, Yorktown Heights, NY)*. Ed. by Raymond E. Miller and James W. Thatcher. New York: Plenum Press, 1972, pp. 85–103.

[KBD+17] Datta Krupa R., Aniket Basu Roy, Minati De, and Sathish Govindarajan. "Demand hitting and covering of intervals". In: *Algorithms and Discrete Applied Mathematics*. Springer International Publishing, 2017, pp. 267–280. ISBN: 978-3-319-53007-9.

[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.

[KM99] Gunnar W. Klau and Petra Mutzel. "Combining graph labeling and compaction". In: *Proceedings of the 7th International Symposium on Graph Drawing (GD '99)*. Vol. 1731. LNCS. Springer, 1999, pp. 27–37. ISBN: 978-3-540-66904-3. DOI: 10.1007/3-540-46648-7_3.

[KPS14] Stephen G. Kobourov, Sergey Pupyrev, and Bahador Saket. "Are crossings important for drawing large graphs?" In: *Proceedings of the 22nd International Symposium on Graph Drawing (GD '14)*. Vol. 8871. Springer, 2014, pp. 234–245. ISBN: 978-3-662-45802-0. DOI: 10.1007/978-3-662-45803-7_20.

[Kra99] Douglas Kramer. "API documentation from source code comments: a case study of Javadoc". In: *Proceedings of the 17th Annual International Conference on Computer Documentation*. SIGDOC '99. ACM, 1999, pp. 147–153. URL: http://doi.acm.org/10.1145/318372.318577.

[KT97] Konstantinos G. Kakoulis and Ioannis G. Tollis. "An algorithm for labeling edges of hierachical drawings". In: *Graph Drawing (Proceedings GD '97)*. Ed. by Giuseppe Di Battista. LNCS. Springer-Verlag, 1997, pp. 169–180.

[KT98] Konstantinos G. Kakoulis and Ioannis G. Tollis. "A unified approach to labeling graphical features". In: *Proceedings of the Fourteenth Annual Symposium on Computational Geometry*. SCG '98. New York, NY, USA: ACM, 1998, pp. 347–356. ISBN: 0-89791-973-4. DOI: 10.1145/276884.276923. URL: http://doi.acm.org/10.1145/276884.276923.

[LA94] Ying K. Leung and Mark D. Apperley. "A review and taxonomy of distortion-oriented presentation techniques". In: *ACM Transactions on Computer-Human Interaction* 1.2 (June 1994), pp. 126–160.

[Las15] Yella Lasch. "Label Management in Graph Layout Algorithmen". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/ybl-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2015.

[Lee06] Edward A. Lee. "The problem with threads". In: *IEEE Computer* 39.5 (2006), pp. 33–42.

[LGA+02] Yongqiang Li, John Grundy, Robert Amor, and John Hosking. "A data mapping specification environment using a concrete business form-based metaphor". In: *Proceedings the IEEE 2002 Symposia on Human Centric Computing Languages and Environments*. 2002, pp. 158–166. DOI: 10.1109/HCC.2002.1046368.

[LNW03] Edward A. Lee, Stephen Neuendorffer, and Michael J. Wirthlin. "Actor-oriented design of embedded hardware and software systems". In: *Journal of Circuits, Systems, and Computers (JCSC)* 12.3 (2003), pp. 231–260. DOI: 10.1142/S0218126603000751.

[McA99] Andrew J. McAllister. *A new heuristic algorithm for the linear arrangement problem*. Tech. rep. University of New Brunswick, 1999.

[MEL+95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. "Layout adjustment and the mental map". In: *Journal of Visual Languages & Computing* 6.2 (June 1995), pp. 183–210. DOI: 10.1006/jvlc.1995.1010.

[MJ03] Benjamin Musial and Timothy Jacobs. "Application of focus + context to UML". In: *APVis '03: Proceedings of the Asia-Pacific symposium on Information visualisation*. Adelaide, Australia: Australian Computer Society, Inc., 2003, pp. 75–80. ISBN: 1-920682-03-1.

[MSH14] Christian Motika, Steven Smyth, and Reinhard von Hanxleden. "Compiling SCCharts—A case-study on interactive model-based compilation". In: *Proceedings of the 6th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2014)*. Vol. 8802. LNCS. Corfu, Greece, Oct. 2014, pp. 461–480. DOI: 10.1007/978-3-662-45234-9.

[MTL78] Robert McGill, John W. Tukey, and Wayne A. Larsen. "Variations of box plots". In: *The American Statistician* 32.1 (1978), pp. 12–16. ISSN: 00031305.

[Nic95] Jeffrey Vernon Nickerson. "Visual programming". PhD thesis. New York University, Sept. 1995.

[Nor88] Donald A. Norman. *The design of everyday things*. New York: Basic Books, 1988. ISBN: 0-465-06710-7.

[NTB05] Nikola S. Nikolov, Alexandre Tarassov, and Jürgen Branke. "In search for efficient heuristics for minimum-width graph layering with consideration of dummy nodes". In: *Journal of Experimental Algorithmics* 10 (2005). ISSN: 1084-6654. DOI: 10.1145/1064546.1180618.

[Obj17] Object Management Group. *OMG Unified Modeling Language Specification, Version 2.5.1*. https://www.omg.org/spec/UML/2.5.1/. Dec. 2017.

[OCP14] Mohd Hafeez Osman, Michel R. V. Chaudron, and Peter van der Putten. "Interactive scalable abstraction of reverse engineered UML class diagrams". In: *21st Asia-Pacific Software Engineering Conference, APSEC 2014, Jeju, South Korea, December 1-4, 2014. Volume 1: Research Papers*. 2014, pp. 159–166. DOI: 10.1109/APSEC.2014.34. URL: http://dx.doi.org/10.1109/APSEC.2014.34.

[Pap76] Ch.H. Papadimitriou. "The NP-completeness of the bandwidth minimization problem". In: *Computing* 16 (3 1976), pp. 263–270. ISSN: 0010-485X.

[PCA02] Helen C. Purchase, David Carrington, and Jo-Anne Allder. "Empirical evaluation of aesthetics-based graph layout". In: *Empirical Software Engineering* 7 (3 2002), pp. 233–255. ISSN: 1382-3256.

[Pet06] Marian Petre. "Cognitive dimensions 'beyond the notation'". In: *Journal of Visual Languages & Computing* 17.4 (2006), pp. 292–301.

[Pet13] Marian Petre. "UML in practice". In: *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*. San Francisco, CA, USA: IEEE Press, 2013, pp. 722–731. ISBN: 978-1-4673-3076-3.

[Pet14] Marian Petre. "'"No shit" or "Oh, shit!": responses to observations on the use of UML in professional practice". In: *Software & Systems Modeling* 13.4 (Oct. 2014), pp. 1225–1235. ISSN: 1619-1374. DOI: 10.1007/s10270-014-0430-4.

[Pet95] Marian Petre. "Why looking isn't always seeing: Readership skills and graphical programming". In: *Communications of the ACM* 38.6 (June 1995), pp. 33–44.

[Plö15] Christina Plöger. "Improving comment attachment algorithms". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cpl-bt.pdf. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2015.

[PMN03] Timo Poranen, Erkki Mäkinen, and Jyrki Nummenmaa. "How to draw a sequence diagram". In: *Proceedings of the Eighth Symposium on Programming Languages and Software Tools (SPLST'03)*. 2003.

[Pto14] Claudius Ptolemaeus, ed. *System design, modeling, and simulation using Ptolemy II*. Ptolemy.org, 2014. URL: http://ptolemy.org/books/Systems.

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

[RAC+17] Ulf Rüegg, Marc Adolf, Michael Cyruk, Astrid Mariana Flohr, and Reinhard von Hanxleden. *Minimum-width graph layering revisited*. Technical Report 1701. ISSN 2192-6247. Kiel University, Department of Computer Science, Feb. 2017.

[RDM+87] Lawrence Rowe, Michael Davis, Eli Messinger, Carl Meyer, Charles Spirakis, and Allen Tuan. "A browser for directed graphs". In: *Software – Practice and Experience* 17.1 (Jan. 1987), pp. 61–76.

[Ree79] Trygve Reenskaug. *Models – Views – Controllers*. Xerox PARC technical note. Dec. 1979.

[RG12] R. Reicherdt and S. Glesner. "Slicing MATLAB Simulink models". In: *2012 34th International Conference on Software Engineering (ICSE)*. IEEE, June 2012, pp. 551–561. DOI: 10.1109/ICSE.2012.6227161.

[RH18] Ulf Rüegg and Reinhard von Hanxleden. *Wrapping layered graphs*. Technical Report 1803. ISSN 2192-6247. Kiel University, Department of Computer Science, Feb. 2018.

[RLP+16] Ulf Rüegg, Rajneesh Lakkundi, Ashwin Prasad, Anand Kodaganur, Christoph Daniel Schulze, and Reinhard von Hanxleden. "Incremental diagram layout for automated model migration". In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS '16)*. 2016, pp. 185–195. DOI: 10.1145/2976767.2976805.

[RSC+15] Ulf Rüegg, Christoph Daniel Schulze, John Julian Carstens, and Reinhard von Hanxleden. "Size- and port-aware horizontal node coordinate assignment". In: *Proceedings of the 23rd International Symposium on Graph Drawing and Network Visualization (GD '15)*. 2015, pp. 139–150. DOI: 10.1007/978-3-319-27261-0_12.

[RSG+16] Ulf Rüegg, Christoph Daniel Schulze, Daniel Grevismühl, and Reinhard von Hanxleden. *Using one-dimensional compaction for smaller graph drawings*. Technical Report 1601. ISSN 2192-6247. Kiel University, Department of Computer Science, Apr. 2016.

[RSM+16] Francesca Rybicki, Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, and Reinhard von Hanxleden. "Interactive model-based compilation continued – interactive incremental hardware synthesis for SCCharts". In: *Proceedings of the 7th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA 2016)*. Vol. 9953. LNCS. Corfu, Greece, Oct. 2016, pp. 150–170. DOI: 10.1007/978-3-662-45234-9.

[RSS+13] Ulf Rüegg, Christian Schneider, Christoph Daniel Schulze, Miro Spönemann, Christian Motika, and Reinhard von Hanxleden. *Light-weight synthesis of Ptolemy diagrams with KIELER*. Presentation at the Tenth Biennial Ptolemy Miniconference, Berkeley, CA, USA. Nov. 2013.

[Rüe18] Ulf Rüegg. *Sugiyama layouts for prescribed drawing areas*. Kiel Computer Science Series 2018/1. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2018.

[San04] Georg Sander. "Layout of directed hypergraphs with orthogonal hyperedges". In: *Proceedings of the 11th International Symposium on Graph Drawing (GD '03)*. Ed. by Giuseppe Liotta. Vol. 2912. LNCS. Springer, 2004, pp. 381–386. ISBN: 978-3-540-20831-0. DOI: 10.1007/978-3-540-24595-7_35.

[San94] Georg Sander. *Graph layout through the VCG tool*. Tech. rep. A03/94. 66041 Saarbrücken: Universität des Saarlandes, FB 14 Informatik, Oct. 1994.

[San96] Georg Sander. "A fast heuristic for hierarchical Manhattan layout". In: *Proceedings of the Symposium on Graph Drawing (GD '95)*. Ed. by Franz J. Brandenburg. Vol. 1027. LNCS. Springer, 1996, pp. 447–458. ISBN: 3-540-60723-4. DOI: 10.1007/BFb0021828.

[SB92] Manojit Sarkar and Marc H. Brown. "Graphical fisheye views of graphs". In: *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. ACM, 1992, pp. 83–91. ISBN: 0-89791-513-5. DOI: http://doi.acm.org/10.1145/142750.142763.

[SBP+09] Dave Steinberg, Frank Budinsky, Marcelo Paternostro, and Ed Merks. *EMF – Eclipse Modeling Framework*. Second Edition. Eclipse Series. Addison-Wesley, 2009. ISBN: 978-0-321-33188-5.

[Sch11] Christoph Daniel Schulze. "Optimizing automatic layout for data flow diagrams". Diploma Thesis. Kiel University, Department of Computer Science, July 2011.

[Sch16a] Alan Schelten. "Hierarchy-aware layer sweep". Master's thesis. Kiel University, Department of Computer Science, Sept. 2016.

[Sch16b] Christoph Daniel Schulze. "Two opportunities and challenges of automatic layout in visual languages". In: *Proceedings of the ACM Student Research Competition at MODELS 2016 co-located with the 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*. 2016.

[SFH+10] 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)*. Vol. 5849. LNCS. Springer, 2010, pp. 135–146. DOI: 10.1007/978-3-642-11805-0.

[SH14] 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, pp. 41–44. DOI: 10.1109/VLHCC.2014.6883019.

[SHH18a] Christoph Daniel Schulze, Gregor Hoops, and Reinhard von Hanxleden. "Automatic layout and label management for UML sequence diagrams". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '18)*. 2018.

[SHH18b] Christoph Daniel Schulze, Gregor Hoops, and Reinhard von Hanxleden. *Automatic layout and label management for UML sequence diagrams*. Technical Report 1804. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2018.

[SLH16] Christoph Daniel Schulze, Yella Lasch, and Reinhard von Hanxleden. "Label management: keeping complex diagrams usable". In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '16)*. Sept. 2016, pp. 3–11. DOI: 10.1109/VLHCC.2016.7739657.

[SLP+13] Paul Schmieder, Andrew Luxton-Reilly, Beryl Plimmer, and John G. Hosking. "Visual guides for comprehending digital ink in distortion lenses". In: *Proceedings of the 27th International BCS Human Computer Interaction Conference BCS-HCI'13*. 2013. URL: http://dl.acm.org/citation.cfm?id=2578053.

[SPH16a] Christoph Daniel Schulze, Christina Plöger, and Reinhard von Hanxleden. "On comments in visual languages". In: *Proceedings of the 9th International Conference on the Theory and Application of Diagrams (DIAGRAMS '16)*. LNCS. Springer, 2016, pp. 219–225. ISBN: 978-3-319-42333-3. DOI: 10.1007/978-3-319-42333-3_17.

[SPH16b] Christoph Daniel Schulze, Christina Plöger, and Reinhard von Hanxleden. *On comments in visual languages*. Technical Report 1602. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Apr. 2016.

[Spö09] Miro Spönemann. "On the automatic layout of data flow diagrams". Diploma Thesis. Kiel University, Department of Computer Science, Mar. 2009.

[Spö15] Miro Spönemann. *Graph layout support for model-driven engineering*. Kiel Computer Science Series 2015/2. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2015. ISBN: 9783734772689.

[SSH13] 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)*. San Jose, CA, USA, Sept. 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[SSH14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. "Drawing layered graphs with port constraints". In: *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout* 25.2 (2014), pp. 89–106. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2013.11.005.

[SSR+14] 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)*. Vol. 8578. LNAI. Springer, 2014, pp. 9–15. DOI: 10.1007/978-3-662-44043-8_2.

[Stö14] Harald Störrle. "On the impact of layout quality to understanding UML diagrams: size matters". In: *Model-Driven Engineering Languages and Systems - 17th International Conference, MODELS 2014, Valencia, Spain, September 28 - October 3, 2014. Proceed-*

ings. 2014, pp. 518–534. DOI: 10.1007/978-3-319-11653-2_32. URL: http://dx.doi.org/10.1007/978-3-319-11653-2_32.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. "Methods for visual understanding of hierarchical system structures". In: *IEEE Transactions on Systems, Man and Cybernetics* 11.2 (Feb. 1981), pp. 109–125.

[SWH18a] Christoph Daniel Schulze, Nis Wechselberg, and Reinhard von Hanxleden. "Edge label placement in layered graph drawing". In: *Proceedings of the 10th International Conference on the Theory and Application of Diagrams (DIAGRAMS '18)*. LNCS. Springer, 2018, pp. 71–78. ISBN: 978-3-319-91376-6. DOI: 10.1007/978-3-319-91376-6_10.

[SWH18b] Christoph Daniel Schulze, Nis Wechselberg, and Reinhard von Hanxleden. *Edge label placement in layered graph drawing*. Technical Report 1802. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Feb. 2018.

[Tam13] Roberto Tamassia, ed. *Handbook of graph drawing and visualization*. CRC Press, 2013. ISBN: 978-1584884125.

[TMT+12] S. H. Tan, D. Marinov, L. Tan, and G. T. Leavens. "@tComment: testing Javadoc comments to detect comment-code inconsistencies". In: *Fifth IEEE International Conference on Software Testing, Verification and Validation*. Apr. 2012, pp. 260–269. DOI: 10.1109/ICST.2012.106.

[TNB04] Alexandre Tarassov, Nikola S. Nikolov, and Jürgen Branke. "A heuristic for minimum-width graph layering with consideration of dummy nodes". In: *Experimental and Efficient Algorithms* (2004), pp. 570–583. ISSN: 03029743.

[Toe14] Tibor Toepffer. "Schöne Kurven: Ebenenbasiertes Kantenrouting mit Splines". http://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/tit-dt.pdf. Diploma thesis. Kiel University, Department of Computer Science, Nov. 2014.

[TR05] Martyn Taylor and Peter Rodgers. “Applying graphical design techniques to graph visualization”. In: *Proceedings of the Ninth International Conference on Information Visualization (InfoVIS'05)*. July 2005, pp. 651–656.

[WEK+12] Johan Wagemans, James H. Elder, Michael Kubovy, Stephen E. Palmer, Mary A. Peterson, Manish Singh, and Rüdiger von der Heydt. “A century of Gestalt psychology in visual perception: I. Perceptual grouping and figure–ground organization”. In: *Psychological Bulletin* 138.6 (Nov. 2012), pp. 1172–1217. DOI: 10.1037/a0029333.

[Wer23] Max Wertheimer. “Untersuchungen zur Lehre von der Gestalt. II.” In: *Psychologische Forschung* 4.1 (1923), pp. 301–350.

[WMP+05] Pak Chung Wong, Patrick Mackey, Ken Perrine, James Eagan, Harlan Foote, and Jim Thomas. “Dynamic visualization of graphs with extended labels”. In: *INFOVIS '05: Proceedings of the Proceedings of the 2005 IEEE Symposium on Information Visualization*. Washington, DC, USA: IEEE Computer Society, 2005, p. 10. ISBN: 0-7803-9464-x. DOI: 10.1109/INFOVIS.2005.11.

[WPC+02] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. “Cognitive measurements of graph aesthetics”. In: *Information Visualization* 1.2 (2002), pp. 103–110. ISSN: 1473-8716.

[WS05] Kenny Wong and Dabo Sun. “On evaluating the layout of UML diagrams for program comprehension”. In: *Proceedings of the 13th International Workshop on Program Comprehension (IWPC'05)*. May 2005, pp. 317–326.

[XRP+12] Kai Xu, Chris Rooney, Peter Passmore, Dong-Han Ham, and Phong H. Nguyen. “A user study on curved edges in graph visualization”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.12 (Dec. 2012), pp. 2449–2456. ISSN: 1077-2626. DOI: 10.1109/TVCG.2012.189.

Detailed Contents

1 Introduction **1**
- 1.1 Three Visual Languages . . . 6
  - 1.1.1 Ptolemy II . . . 7
  - 1.1.2 Sequentially Constructive Statecharts (SCCharts) . . . 9
  - 1.1.3 UML Sequence Diagrams . . . 15
- 1.2 Principles . . . 19
- 1.3 Contributions . . . 21
- 1.4 Related Work . . . 24
- 1.5 Publications . . . 30
- 1.6 Outline . . . 33

2 Foundations **35**
- 2.1 Basic Terminology . . . 35
- 2.2 Aesthetics . . . 39
  - 2.2.1 Common Aesthetic Criteria . . . 40
  - 2.2.2 Aesthetics of Sequence Diagrams . . . 43
- 2.3 Box Plots . . . 44
- 2.4 The Eclipse Platform . . . 45
  - 2.4.1 Noteworthy Eclipse Projects . . . 48
- 2.5 The Eclipse Layout Kernel . . . 52
  - 2.5.1 Graph Structure . . . 54
  - 2.5.2 The Layout Process . . . 57
  - 2.5.3 Layout Creation and Layout Adjustment . . . 58
  - 2.5.4 A Bit of History . . . 59
  - 2.5.5 How ELK Relates to This Thesis . . . 61
- 2.6 KIELER Lightweight Diagrams (KLighD) . . . 61

I Laying the Foundations: Automatic Layout 65

3 Flow-Based Diagrams 67

3.1 The Layered Approach . . . 70
3.1.1 Phase 1: Cycle Breaking . . . 72
3.1.2 Phase 2: Layer Assignment . . . 74
3.1.3 Phase 3: Crossing Minimization . . . 76
3.1.4 Phase 4: Node Placement . . . 78
3.1.5 Phase 5: Edge Routing . . . 80
3.1.6 ELK Layered . . . 82
3.1.7 Ports . . . 84
3.1.8 Hierarchical Nodes . . . 86
3.2 Micro Layout of Nodes . . . 89
3.2.1 Node Labels . . . 95
3.2.2 Ports and Port Labels . . . 98
3.2.3 Configuring Micro Layout . . . 101
3.2.4 The Cell System . . . 107
3.2.5 A Micro Layout Algorithm . . . 111
3.2.6 Avoiding Node Overlaps . . . 124
3.2.7 Micro Layout for Hierarchical Nodes . . . 125
3.3 Edge Label Placement . . . 128
3.3.1 Edge Label Locations . . . 131
3.3.2 Head and Tail Labels . . . 132
3.3.3 Center Labels . . . 133
3.3.4 Label Side Selection . . . 141
3.3.5 Directional Decorators . . . 153
3.4 Evaluation . . . 154
3.4.1 Layout Impact of Side Selection Strategies . . . 155
3.4.2 Layout Impact of Layer Selection Strategies . . . 161
3.4.3 A Controlled Experiment . . . 165
3.4.4 Two Surveys . . . 179
3.4.5 Discussion . . . 183

4 Sequence Diagrams **187**
4.1 Sequence Diagrams as Graphs . . . 188
4.2 Laying Out Sequence Diagrams . . . 192
4.2.1 Phase 1: Lifeline Ordering . . . 195
4.2.2 Phase 2: Space Allocation . . . 200
4.2.3 Phase 3: Cycle Breaking . . . 203
4.2.4 Phase 4: Communication Line Assignment . . . 205
4.2.5 Phase 5: Coordinate Assignment . . . 209
4.3 The KieSL Language . . . 210
4.4 Evaluation . . . 212
4.4.1 Lifeline Ordering Strategies . . . 213
4.4.2 Vertical Compaction . . . 214
4.4.3 Discussion . . . 214

II Reaping the Rewards: Layout Pragmatics **219**

5 Comment Attachment **221**
5.1 Comments and Secondary Notation . . . 223
5.2 The Comment Attachment Pipeline . . . 225
5.3 Measuring Attachment . . . 227
5.3.1 Filters . . . 228
5.3.2 Matchers . . . 232
5.3.3 Discussion . . . 238
5.4 Implementing the Pipeline . . . 238
5.5 Evaluation . . . 241
5.5.1 Experiment 1 . . . 242
5.5.2 Experiment 2 . . . 244
5.5.3 Experiment 3 . . . 245
5.5.4 Discussion . . . 246

6 Label Management **249**
6.1 Label Management Strategies . . . 254
6.1.1 Basic Strategies . . . 255
6.1.2 Composite Strategies . . . 259

6.2 Integrating Label Management . . . 260
6.2.1 The Preprocessing Approach . . . 262
6.2.2 The Feedback Loop Approach . . . 263
6.2.3 Combining the Approaches . . . 266
6.2.4 Determining the Target Width . . . 267
6.2.5 A Label Management Framework . . . 269
6.2.6 Label Management and Automatic Layout . . . 272
6.3 Designing the User Interface . . . 273
6.3.1 Providing Label Management to Users . . . 274
6.3.2 Communicating Managed Labels . . . 275
6.4 Evaluation . . . 277
6.4.1 Case Studies . . . 277
6.4.2 Aesthetics . . . 282
6.4.3 Two Surveys . . . 287

7 Conclusions **291**
7.1 Summary . . . 291
7.2 Lessons Learned . . . 294
7.3 Open Problems . . . 297
7.4 Conclusion . . . 299

A Sample Diagrams **301**
A.1 SCCharts . . . 301
A.1.1 Edge Label Placement . . . 301
A.1.2 Label Management . . . 305
A.2 UML Sequence Diagrams . . . 312

Bibliography **319**

Lists **339**

Glossary **355**

List of Figures

1.1 Advantages of visual languages . . . 3
a Textual description . . . 3
b Graphical view . . . 3
1.2 A small Ptolemy II model . . . 8
1.3 Editing a Ptolemy II model in the Vergil editor . . . 10
1.4 Viewing a Ptolemy II model in the KIELER Ptolemy Browser . 11
1.5 A small SCChart . . . 13
1.6 Editing an SCChart in KIELER . . . 14
1.7 A small UML sequence diagram . . . 17
1.8 Terminology of UML sequence diagrams . . . 18
1.9 Editing a UML sequence diagram with KieSL . . . 20

2.1 The terminology of hierarchical graphs . . . 36
2.2 Inclusion trees . . . 38
2.3 Box plots . . . 45
2.4 The OSGi component system . . . 47
2.5 The basic structure of ELK . . . 52
2.6 The ELK graph data structure . . . 55
2.7 Routing hyperedges in ELK . . . 56
2.8 The KLighD view generation process . . . 62

3.1 Different ways of drawing edges . . . 69
a Directed . . . 69
b Undirected . . . 69
c Undirected with fixed connection sides . . . 69
d Undirected with explicit ports . . . 69
3.2 Layer-based layout . . . 70
3.3 Example graph before cycle breaking . . . 71
3.4 Example graph after cycle breaking . . . 72

3.5 Example graph after layer assignment . . . 74
3.6 Example graph after crossing minimization . . . 77
3.7 Example graph after node placement . . . 79
3.8 Example graph after edge routing . . . 80
3.9 The structure of ELK Layered . . . 83
3.10 Laying out hierarchical ports . . . 88
a Diagram with hierarchical ports . . . 88
b ELK Layered's internal representation . . . 88
3.11 Model simulation in ETAS EHANDBOOK . . . 91
3.12 A spectrum of layout algorithms handling node sizes . . . 92
3.13 Labels in Graphviz . . . 94
3.14 Possible node label locations . . . 96
3.15 Node label grid . . . 96
a Strict . . . 96
b Relaxed . . . 96
3.16 Placing port labels inside of a node . . . 99
a Sufficient space . . . 99
b Insufficient space . . . 99
3.17 Placing port labels outside of a node . . . 100
3.18 Avoiding edge-label crossings with outside port labels . . . 101
3.19 The effect of size constraints . . . 102
a Fixed size . . . 102
b NodeLabels . . . 102
c Add Ports . . . 102
d Add PortLabels . . . 102
e Add MinSize . . . 102
3.20 The effect of size options . . . 105
a ClientAreaMinSize . . . 105
b PortsOverhang . . . 105
c LabelsOverhang . . . 105
d EqualPortSpacing . . . 105
e CompactPortLabels . . . 105
3.21 The effect of size options (continued) . . . 106
a StrictGrid . . . 106
b Asymmetrical . . . 106

3.22 Surrounding port space . . . 107
3.23 Class diagram of the cell system . . . 108
3.24 Layout of container cells . . . 110
a Strip container cell . . . 110
b Grid container cell . . . 110
3.25 Modeling the micro layout problem using cells . . . 112
3.26 Calculating port margins . . . 116
a Two ports . . . 116
b More than two ports . . . 116
c Equal port spacing . . . 116
d Inside port labels . . . 116
3.27 Minimum node width for FIXED RATIO ports . . . 118
3.28 Placing nodes based on their size . . . 125
a Real size . . . 125
b Virtual size . . . 125
3.29 Offsetting inside labels of hierarchical ports . . . 127
a Bad port label placement . . . 127
b Good port label placement . . . 127
3.30 Edge label locations . . . 132
3.31 Overlaps with long edge end labels . . . 134
a With overlaps . . . 134
b Without overlaps . . . 134
3.32 Layer selection strategies for center edge labels . . . 135
3.33 Non-optimality of the widest layer heuristic . . . 137
3.34 Non-optimality of the space-efficient layer heuristic . . . 141
3.35 Effect of label sides on label dummy nodes . . . 142
3.36 Conflicting end label sides . . . 143
3.37 Same-side label side selection . . . 144
a Clear spacings . . . 144
b Ambiguous spacings . . . 144
3.38 Directional label side selection . . . 145
3.39 Augmented same-side strategy, Rule 1 (horizontal case) . . . 146
a Without . . . 146
b With . . . 146
3.40 Augmented same-side strategy, Rule 1 (vertical case) . . . 148

a Without . . . 148
b With . . . 148
3.41 Augmented same-side strategy, Rule 2 . . . 149
a Without . . . 149
b With . . . 149
3.42 Augmented same-side strategy, Rule 3 . . . 150
a Without . . . 150
b With . . . 150
3.43 On-edge label placement . . . 152
3.44 On-edge label design examples . . . 152
3.45 Directional decorators . . . 153
3.46 Same-side label side selection with arrows . . . 154
a Clear spacings . . . 154
b Ambiguous spacings . . . 154
3.47 Relative drawing size by label side selection algorithm . . . 159
a Horizontal drawings . . . 159
b Vertical drawings . . . 159
3.48 Width of drawings by label layer selection strategy . . . 164
3.49 Positions of center edge labels . . . 164
3.50 Experimental stimuli for the experiment's first part . . . 168
a Condition 1 . . . 168
b Condition 2 . . . 168
3.51 Box plots of the results for the experiment's first part . . . 169
a Reaction time . . . 169
b Error rate . . . 169
3.52 Experimental stimulus for the experiment's second part . . . 171
3.53 Box plots of the results of the connectivity experiment . . . 173
a Reaction time . . . 173
b Error rate . . . 173
3.54 Sums of label side selection preference rankings . . . 176
3.55 Frequencies of label side selection preference rankings . . . 177
3.56 Typical SCChart to solve the bar code assignment . . . 181
3.57 Label side selection strategy rankings . . . 182
3.58 Label side selection strategy usage (survey 2) . . . 183

4.1 Sequence diagrams are not graphs . . . 188
4.2 Sequence diagram represented as ELK graph . . . 191
a Sequence diagram . . . 191
b ELK graph . . . 191
4.3 Example graph before phase 1 . . . 194
4.4 Example graph after phase 1 (communication line order) . . . 197
4.5 Example graph after phase 1 (short message order) . . . 199
4.6 Example graph after phase 2 . . . 201
4.7 A problem with combined fragments . . . 202
4.8 Sequence diagram with cyclic element ordering graph . . . 204
a Sequence diagram . . . 204
b Cyclic element ordering graph . . . 204
c Ayclic element ordering graph . . . 204
4.9 Another problem with combined fragments . . . 206
4.10 Naive communication line assignment . . . 207
a Assignment . . . 207
b Wrong drawing . . . 207
4.11 A sequence diagram described in KieSL . . . 211
4.12 Message length by ordering strategy . . . 213
a Mean message length . . . 213
b Change in mean message length . . . 213
4.13 Message-lifeline crossings by ordering strategy . . . 215
a Message-lifeline crossings . . . 215
b Change in message-lifeline crossings . . . 215
4.14 Height of drawings with and without vertical compaction . . . 216
a All diagrams . . . 216
b Affected diagrams . . . 216
c Height change in affected diagrams . . . 216

5.1 Layout without comment attachment . . . 222
5.2 Explicit comment attachments . . . 224
5.3 Comment attachment pipeline . . . 227
5.4 Title comment in a Ptolemy II model . . . 229
5.5 Area of comments . . . 231
5.6 Problematic case for the node reference heuristic . . . 233

5.7 Distance between comments and nodes mentioned by name . 234
5.8 Distance between comments and nodes 235
5.9 Alignment between comments and attachment targets 236
5.10 Alignments between comments and nodes 237
5.11 Class diagram of the comment attachment framework 239
5.12 Comment attachment evaluation, experiment 1 243
5.13 Comment attachment evaluation, experiment 2 245
5.14 Comment attachment evaluation, experiment 3 246

6.1 Applying line wrapping to long labels 251
a Original labels 251
b Wrapped labels 251
6.2 Basic universal label management strategies 256
6.3 SCCharts-specific label management strategies 258
6.4 Integrating label management and view generation 261
6.5 Restrictions of the preprocessing approach 265
a Shortened too much 265
b Shortened just enough 265
6.6 Computing the target width for center edge labels 268
6.7 Cyclic dependency with label management in ELK Layered . . 273
6.8 Hiding port labels in Ptolemy II Vergil 278
6.9 Label management in the KIELER Ptolemy Browser 278
6.10 Mean scaling increases (SCCharts) 285
6.11 Mean scaling increases (sequence diagrams) 286
a Both dimensions 286
b Width only 286
6.12 Label management strategy rankings 288
6.13 Label management strategy usage (survey 2) 290

A.1 Horizontal layout (consistent side selection) 301
A.2 Horizontal layout (directional side selection) 301
A.3 Horizontal layout (smart side selection) 302
A.4 Horizontal layout (space-efficient selection) 302
A.5 Vertical layout (consistent side selection) 302
A.6 Vertical layout (directional side selection) 303

A.7 Vertical layout (smart side selection) 304
A.8 SCChart "Serie7A2" (original labels) 305
A.9 SCChart "Serie7A2" (truncated labels) 305
A.10 SCChart "Serie7A2" (line-wrapped labels) 306
A.11 SCChart "Decoder" (original labels) 306
A.12 SCChart "Decoder" (truncated labels) 307
A.13 SCChart "Decoder" (line-wrapped labels) 308
A.14 SCChart "DomelightEngineMindstorm" (original labels) 309
A.15 SCChart "DomelightEngineMindstorm" (truncated labels) . . . 310
A.16 SCChart "DomelightEngineMindstorm" (line-wrapped labels) . 311
A.17 Sequence diagram "Brooklyn Flow" (original order) 312
A.18 Sequence diagram "Brooklyn Flow" (communication line order)313
A.19 Sequence diagram "Brooklyn Flow" (short message order) . . 314
A.20 Sequence diagram "TakeSnapshot" (original order) 315
A.21 Sequence diagram "TakeSnapshot" (communication line order)316
A.22 Sequence diagram "TakeSnapshot" (short message order) . . 317

List of Tables

3.1 Phase implementations available in ELK Layered 84
3.2 Drawing size by side selection algorithm 157
3.3 Relative drawing size by label side selection algorithm 158
3.4 Cases of on-edge side selection improving drawing size . . . 158
3.5 Change in edge length for augmented strategies 160
3.6 Cases of augmented side selection improving edge length . . 161
3.7 Number of bend points by side selection algorithm 162
3.8 Width of drawings by label layer selection strategy 163
3.9 Label side selection preferences with distinct ranks 175
3.10 Label side selection preferences with shared ranks 176
3.11 Interview results 178

4.1 Mapping sequence diagrams to ELK graphs 189
4.2 Number of diagrams affected by lifeline ordering strategies . 214

5.1 Types of comments 225
5.2 Comment-node reference statistics 232

6.1 Aesthetic measurements of label management strategies . . . 284

List of Algorithms

3.1 Augmented same-side label side selection 147

4.1 Communication line lifeline ordering 198

Glossary

CSS Cascading Style Sheets
One of the cornerstone technologies of the *World Wide Web* that describes the presentation of HTML documents.

DSL Domain-Specific Language
A (usually small) language developed for a very specific purpose, as opposed to general-purpose languages.

ELK Eclipse Layout Kernel
An Eclipse project that provides an infrastructure for layout algorithms. Available at: `http://www.eclipse.org/elk`

EMF Eclipse Modeling Framework
A modeling framework for generating code for a class model based on a model specification. Available at: `http://www.eclipse.org/emf`

EPL Eclipse Public License
A business-friendly open source software licence. Available at: `https://eclipse.org/org/documents/epl-v10.html`

GEF Graphical Editing Framework
A framework and an associated set of tools for creating graphics in applications. Available at: `https://eclipse.org/gef`

GMF Graphical Modeling Framework
A framework for generating graphical editors based on EMF and GEF. Available at: `https://www.eclipse.org/modeling/gmp`

HTML Hypertext Markup Language
One of the cornerstone technologies of the *World Wide Web* that should not require explaining.

IBM International Business Machines Corporation
A multinational company offering hardware and software services.

IDE Integrated Development Environment
An editor that provides advanced features for developing software.

JDT Java Development Tools
A set of Eclipse plug-ins for developing Java programs.

KIELER Kiel Integrated Environment for Layout Eclipse Rich Client
An environment for experimenting with pragmatic modeling concepts. Available at: `http://www.informatik.uni-kiel.de/rtsys/kieler`

KieSL KIELER Sequence Diagram Language
A textual DSL for defining UML sequence diagrams.

KIML KIELER Infrastructure for Meta Layout
An interface between diagram viewers and layout algorithms. Superseded by the ELK project.

KLighD KIELER Lightweight Diagrams
A lightweight, extensible visualization framework built on Eclipse.

KLoDD KIELER Layout for Data Flow Diagrams
Implementation of the layer-based approach to graph drawing. Superseded by the ELK Layered algorithm.

MELK Meta Data for ELK
Information about layout providers and layout options.

MVC Model-View-Controller
A well-known design pattern that separates software components into a model, views of the model, and a controllers connecting the two.

O-No Obviously Non-optimal
A problem in a graph drawing that a human immediately sees a solution for.

OSGi Open Service Gateway Initiative
A non-profit organization founded in 1999 that maintains the OSGi standard.

SCChart Sequentially Constructive Statechart
A statechart dialect designed to provide deterministic concurrency with less restrictions than previous languages.

SCT Textual SCCharts Language
A textual language for defining SCCharts.

UI User Interface
The interface software is meant to be used through.

UML Unified Modeling Language
A collection of graphical languages for visualizing the design and the behaviour of (software) systems.

XMI XML Metadata Interchange
An XML-based format for exchanging metadata information. Available at: `http://www.omg.org/spec/XMI`

XML Extensible Markup Language
A format for representing documents in a structured and well-defined way readable by both humans and machines. Available at: `http://www.w3.org/TR/xml11`

This page intentionally left mostly blank.