Automatic Layout of UML Sequence Diagrams

Gregor Hoops

Diplomarbeit
eingereicht im Jahr 2013

Christian-Albrechts-Universität zu Kiel
Real-Time and Embedded Systems Group
Prof. Dr. Reinhard von Hanxleden

Betreut durch: Dipl.-Inf. Christoph Daniel Schulze

Abstract

Graphical modeling and the Unified Modeling Language (UML) in particular are very popular for specifying and documenting software projects and other systems. The Papyrus framework provides graphical editors for each of the fourteen diagram types of the UML.

Usually, a lot of time is spent optimizing the appearance of a diagram by moving the elements around. If this could be done automatically, the costly time of the user could better be spent on other tasks. This thesis proposes an automatic layout algorithm for the Papyrus sequence diagram editor. The algorithm optimizes the vertical position of messages. It is customizable and provides several lifeline sorting strategies that are designed to optimize different aesthetic criteria.

The produced layouts fare very well with respect to common aesthetic criteria as well as to criteria specific to sequence diagrams. The lifeline sorting strategy yields results that are very close to manually optimized sequence diagrams in many cases. All this is done without loosing the capability to compute layout in real-time even for large diagrams.

Eidesstattliche Erklärung

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

Kiel,

_____

Contents

List of Figures

List of Tables

Abbreviations

CDT	C/C++ Development Tools
dfs	depth-first-search
DSL	Domain Specific Language
EMF	Eclipse Modeling Framework
GEF	Graphical Editing Framework
GMF	Graphical Modeling Framework
IDE	Integrated Development Environment
JDT	Java Development Tools
KIELER	Kiel Integrated Environment for Layout Eclipse RichClient
KIML	KIELER Infrastructure for Meta Layout
KLay	KIELER Layout Algorithms
MARTE	Modeling and Analysis of Real-Time and Embedded systems
MBD	Model-Based Design
MDSD	Model Driven Software Development
MDT	Model Development Tools
OGDF	Open Graph Drawing Framework
OMG	Object Management Group
SysML	Systems Modeling Language
UML	Unified Modeling Language
XMI	XML Metadata Interchange
XML	Extensible Markup Language

Chapter 1

Introduction

1.1 Graphical Modeling

Software projects are getting larger and larger not only in industrial contexts, but also with open source projects. With that, they are getting more and more complex. In order to handle that arising complexity, the degree of abstraction is increased by using higher-level programming languages. These languages help improve the process of software development by abstracting from implementation details such as operating system calls and by concentrating on the main tasks. Starting with Assembler-like languages, more advanced languages such as *FORTRAN* were developed soon, abstracting from the use of registers. This process continued with languages such as *C*, which contains more data-types and the possibility to define custom data-types. Domain Specific Languages (DSLs) focus on narrowly-defined fields of software development, such as railway control applications for example. They may vary in their degree of abstraction and it is chosen very high for most of them nowadays.

Along with other high-level languages, graphical modeling languages can be used to visualize the relationships and interactions between different components of a software system. Hierarchy and connections of classes can be visualized by class diagrams as known from the UML. The rough structure of a program can also be visualized by graphical modeling languages, for example by UML state machine diagrams. These diagram types, along with others, are very suitable to show the relations of different kinds of objects in a very simple and understandable way. This helps to bridge the gap between technical and non-technical people by introducing a representation that is understandable for both of them.

The readability of a diagram has the potential to be much better than that of a textual representation. Graphical elements can be comprehended much faster than text. For complex structures, this becomes increasingly important. However, when increasing the level of abstraction, some details may be lost on the way.

Compared with the loss of the detailed low-level programming, the advantages of graphical modeling are more important in general. This especially holds when considering that some languages support the generation of code directly from the model, which saves a big amount of time for programmers.

One of the most widespread graphical modeling languages is the UML. It is especially used to specify and document software projects. The UML defines the underlying models for diagrams and the Object Management Group (OMG), which manages the issues related to the UML, proposes the XML Metadata Interchange (XMI) format for the distribution of the diagrams and models.

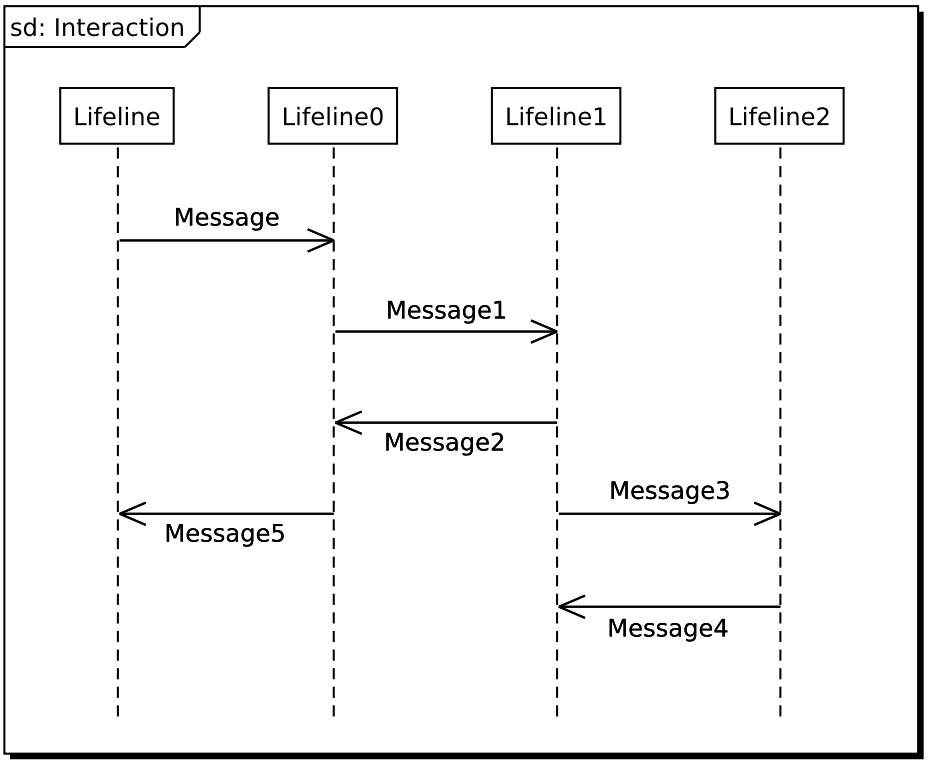
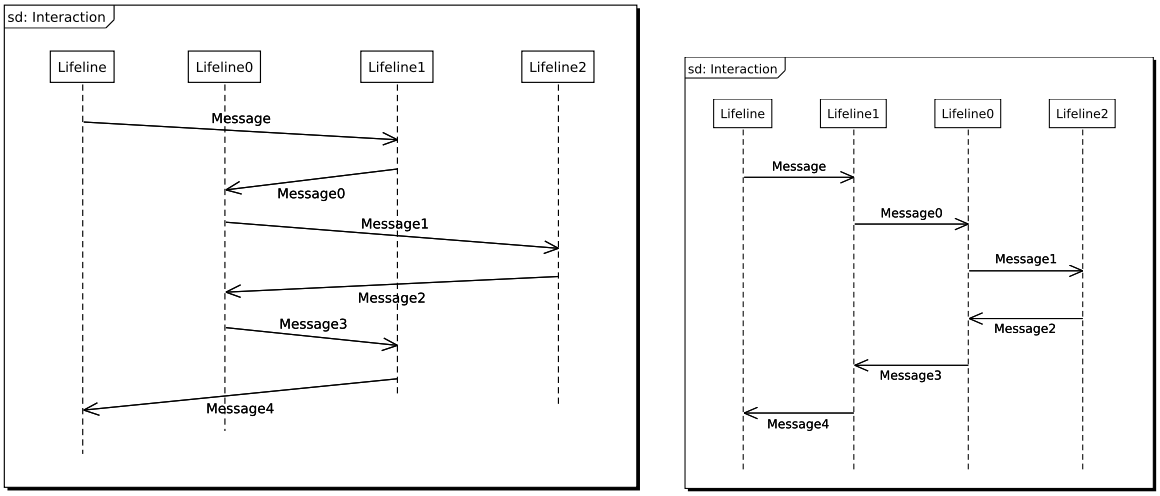


Figure 1.1. A simple sequence diagram example from the Papyrus sequence diagram editor.

Sequence diagrams, which are the main topic of this thesis, are one of the 14 diagram types in the UML. They are used to illustrate how processes interact with one another by showing the sequence of interactions between these processes. Each process is visualized by a *lifeline* that contains its name in a box and a dashed line going downwards. The lifelines are connected by several types of arrows, each indicating a certain kind of *message*. The vertical order of messages across a lifeline meets the chronological order of the events that are described by the messages. Figure 1.1 shows a simple sequence diagram that was modeled with the Papyrus[1] sequence diagram editor.

With sequence diagrams, execution traces of programs can be easily viewed and analyzed. Because of the benefits of graphical modeling, numerous graphical modeling tools were developed for the different graphical modeling languages. Many of them are capable of supporting the process of software development by offering a variety of editors and code generation components. Commercial tools are available as well as free and open source projects. As one of the open source projects, the Papyrus framework is a powerful tool, capable of modeling and editing all kinds of UML diagrams. In this thesis, we apply the technique of automatic layout to sequence diagram editing in Papyrus.

[1] http://www.eclipse.org/modeling/mdt/papyrus/



(a) Sequence diagram with manual layout.

(b) The same sequence diagram with automatic layout applied.

Figure 1.2. A simple, handmade sequence diagram with manual and automatic layout.

1.2 Automatic Layout

As mentioned in the previous section, graphical modeling can save a lot of time by abstracting from details. However, arranging the elements in the diagram can be a very time-consuming task too. Since diagrams have 2-dimensions, one cannot simply insert an element between two neighboring elements by shifting them apart as in a text editor. This is usually not possible in a graphical editor without affecting other elements in the diagram. Instead, one has to move several elements to make space for the new element. Additionally, the insertion of another element may change the meaning of the diagram in a way that the whole arrangement of the elements should be modified in order to show their relations. Therefore, a significant amount of time is spent on tasks related to the arrangement of the graphical elements of the diagram. Klauske and Dziobek state that on average 30% of the work are spent on this [KD10].

This is where automatic layout comes into play. If the computer did all this rearranging, the productivity of the designer could rise significantly. Figure 1.2a shows a very basic sequence diagram that was modeled by hand. Figure 1.2b shows the same diagram rearranged by the automatic layout algorithm developed in this thesis. The diagram that was laid out by the algorithm shows horizontal messages that have an equal distance whereas the handmade diagram is less clearly structured. Additionally, the size of the drawing was optimized automatically.

To be able to automatically compute positions for all the elements in the diagram, a computer-readable representation of the relations of the diagram elements is necessary. Similarly to many other problems in computer science, the problem of *diagram layout* can be

approached by using graph structures for the representation of the diagram. Most diagrams can be described roughly by dividing them into nodes and edges.

Graph layout is a problem that is very well known in computer science. For an overview of previous work on graph layout, see the surveys by Di Battista et al. [DETT98, DETT94], Kaufmann and Wagner [KW01] and Jünger and Mutzel [JM03].

A lot of work has been done to optimize the results of automatic graph layout. Several approaches are known for different kinds of graph structures. Most of these approaches are specialized for a certain kind of diagram types.

- Force-based approaches [FR91, GN98] try to optimize the layout by placing connected nodes near each other. This is done by applying forces that push unconnected nodes apart while pulling connected nodes towards each other. With that, the relations of the nodes are emphasized. This approach is especially used with undirected graphs.

- Planarization-based layout algorithms [Kan96, KM98, CGMW10] focus on avoiding edge crossings. To do that, they try to find a planar embedding of the graph, meaning a possibility to draw the graph without having any edge crossings in the drawing. In many cases it is not possible to find such an embedding. If so, a planar subgraph is calculated and the remaining edges are inserted in a way that produces as few crossings as possible. In the end, the graph is most commonly drawn in an orthogonal way, which means all edges are composed of horizontal and vertical segments.

- Layered graph layout [STT81, ESK04] was introduced to emphasize the flow of data in a way such that as many edges as possible point from left to right. It is applicable to many other diagram types too. The algorithm is divided into several phases that address different sub-problems.

When trying to implement one of these approaches, many of the arising sub-problems are hard to solve. Finding an optimal solution with respect to a single criterion of the algorithm is often enough NP-hard. Considering the difficulties of its single parts, the problem of automatic graph layout in general is a very difficult one. This forces developers to make use of heuristic approaches whenever runtime is a crucial issue. Hence, heuristics are in use in most of the common graph layout algorithms. They do not provide the optimal solution in general, but the solution is close to that optimum in most cases. The runtime of these heuristics is significantly faster than that of the optimal solution especially for large diagrams. Therefore the ratio of the runtime compared to the optimality of the results is best for heuristic approaches in general.

1.3 Goals of this Thesis

The main goal of this thesis is to implement a layout algorithm capable of laying out all kinds of sequence diagrams. Layout of sequence diagrams, however, is different than the layout of other diagram types. This is because sequence diagrams cannot be easily mapped to graphs.

Therefore, a new approach has to be developed in order to match the peculiarities of sequence diagrams.

The layout algorithm is to be integrated with the sequence diagram editor of the Papyrus framework. In order to do so, the infrastructure of the Kiel Integrated Environment for Layout Eclipse RichClient (KIELER) [FvH09] is used. With the KIELER Infrastructure for Meta Layout (KIML), a layout button is available within the Papyrus editor, enabling the user to trigger the automatic layout process.

As a novel feature in sequence diagram layout, the algorithm is to be able to rearrange the horizontal order of the lifelines. Several sorting strategies are available that relate to different aesthetic optimization goals.

1.4 Outline

The next chapter covers related work. It introduces related diagram types and existing layout algorithms for sequence diagrams. Additionally, it describes what is important for such layout algorithms by examining what kinds of aesthetic criteria have to be met in order to produce a layout that is considered a good layout. Chapter 3 then introduces the products and technologies used in this thesis. Chapter 4 introduces the underlying theoretical concepts necessary to understand the motivation and the concepts of the algorithms. After that, the concepts of the different phases of the algorithm are explained and justified. The implementation details are documented in Chapter 5 where the data structures and the algorithms are described in depth. The evaluation in Chapter 6 is divided into the evaluation of the different aesthetic criteria, the quality of the produced layouts in general, the performance of the algorithm, and a comparison to different sequence diagram layout approaches. The thesis concludes in Chapter 7.

Chapter 2

Related Work

This chapter starts with related work on graph drawing and automatic layout. This is followed by the introduction to related diagram types in Section 2.2, which are compared to sequence diagrams especially with respect to diagram layout. Thereafter, an overview of existing sequence diagram layout algorithms is presented in Section 2.3. Most of these algorithms are delivered as part of special sequence diagram editors. Finishing this chapter, Section 2.4 introduces the aesthetic criteria that are important to be able to measure the quality of diagram layout. These criteria are discussed with respect to their compatibility to sequence diagrams.

2.1 Graph Drawing and Automatic Layout

Graph drawing is a research area that has been the subject of research for a long time. There are many real world problems that can be mapped to graph problems, such as all kinds of maps and path finding problems. Social networks can be modeled as graphs too. It is desirable to be able to visualize these graphs in a way that is simple to understand for all kinds of users. Therefore, a lot of work has been done to build and improve different kinds of graph layout algorithms to be able to produce visualizations automatically.

Fundamental work has been done by Sugiyama, Tagawa and Toda in 1981. They proposed *Methods for visual understanding of hierarchical system structures* [STT81], which was the foundation of the *layered graph layout* approach that divides the layout into several phases. This approach is also known as *hierarchical graph layout*.

Roberto Tamassia and Giuseppe Di Battista had remarkable impact on the topic of graph drawing too. Tamassia published several articles on special graph problems such as drawing of planar graphs [TT86] and minimizing the number of edge bends [Tam87], before they passed on to more general graph drawing topics. Their article on *Automatic graph drawing and readability of diagrams* [TDB88] introduces a general purpose graph drawing algorithm that draws the graph on a grid. The aspect of readability was later picked up by Hellen Purchase. She examined the different kinds of graph drawing aesthetics such as *edge crossings* and *edge bends* and validated their importance for the understanding of graph drawings [PCJ96, Pur97].

2.2 Related Diagram Types

In contrast to other diagram types, sequence diagrams cannot be trivially mapped to graphs. This is because the lifelines, which at first glance could be regarded as the nodes of a graph,

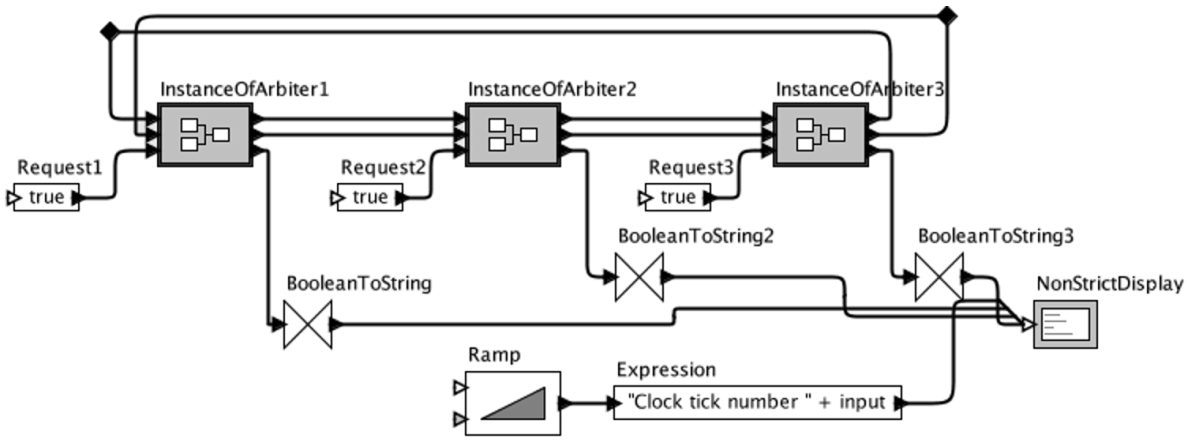


Figure 2.1. An example of a data flow diagram taken from the Ptolemy framework of the UC Berkeley.

are modeled as vertical lines whereas nodes are normally intended to be circles or boxes. The connection points of incoming and outgoing edges are placed on one of the sides of a node. In sequence diagrams, however, messages are distributed over the vertical line and are usually drawn horizontally.

Another difference between sequence diagrams and other diagram types is that lifelines are aligned side by side, sharing the same vertical coordinate in general (this is only voided for lifelines that are created by a create message). The distribution of the lifelines in sequence diagrams thus becomes a one dimensional problem whereas it is a two dimensional one in most other diagram types.

There are a lot of diagram types in use for different purposes. For all of these diagram types, different layout results are desired. Because of that, a lot of different layout algorithms have been developed. However, some of these algorithms are applicable for different diagram types. To obtain good results for different types of diagrams, some parameters may have to be altered depending on the diagram type. Some diagram types and their possible layout algorithms are introduced in the following.

Data-flow diagrams are one example of typical diagrams that are used in the process of software development or to visualize the flow of data in a non-technical process. Since data-flow is directed in most cases, these diagrams are often laid out by layered layout algorithms. This is because these algorithms are designed to emphasize the direction of the flow in a directed graph. Figure 2.1 shows a Ptolemy[1] diagram that was laid out with a layered layout algorithm called *KLay Layered* [KSSvH12].

Electrical networks are visualized by special diagram types. Especially in the design of electrical devices these diagram types are used very often to show the connections of the single components such as resistors, capacitors, and switches. A layout of these diagrams has to be different to a layout of data-flow diagrams, though. Wires cannot be routed in a diagonal way. In order to map this to the diagram, edges have to be routed orthogonally.

[1] http://ptolemy.eecs.berkeley.edu/

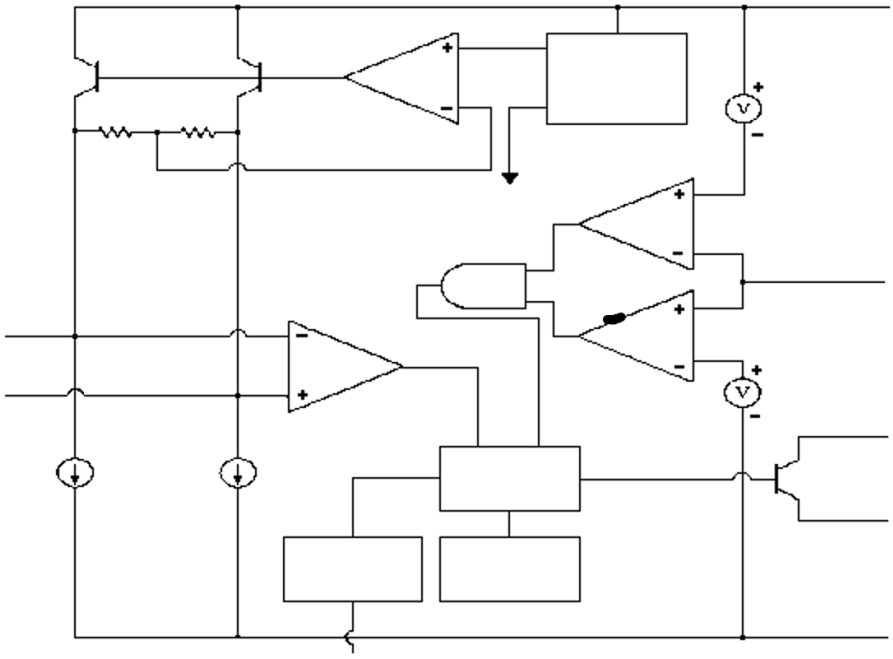


Figure 2.2. An example of an electrical switching circuit diagram. The wires are laid out in an orthogonal style.

This is normally done by planarization-based algorithms. These algorithms are specialized to minimize the number of edge crossings. In addition, these algorithms tend to route edges orthogonally. Layered layout algorithms, however, may also be equipped with orthogonal edge routing algorithms. They are not optimal for most circuits, though, because they tend to produce too many layers to fit onto the rather quadratic shape of a chip. Additionally, layered layout produces more edge crossings in general. The number of edge crossings is increased because messages may not connect nodes of the same layer, which prevents edges from pointing upwards or downwards. A typical electrical switching circuit is shown in Figure 2.2.

Class diagrams are widely used in computer science and software development to visualize the relationships between different classes and their attributes and methods. Classes are connected by inheritance or relationship connections. These diagrams are usually laid out by hierarchical layout algorithms. These algorithms are specialized on highlighting the hierarchy between the different classes. Layered layout can be used too, applying a top-down direction instead of the left-right direction that is very common for data-flow diagrams.

Data structures based on trees are widespread in many areas of computer science. Trees do not contain any circular connections. With that, layered layout should be an option for these diagrams too since it is specialized on cycle-free graphs. Additionally, the layering of the nodes according to the longest path to the source node may create very accurate results as nodes with the same distance to the root of the tree are placed in the same layer. However,

force-based layout is also often used for tree diagrams. Especially for non-binary trees the structure is represented very clearly.

With these layout algorithms at hand, why can we not simply apply one of them to sequence diagrams? The short answer is that these algorithms simply do not match the requirements of sequence diagrams as described above. The constraints for sequence diagrams, such as the side by side alignment of the lifelines, prevent the usage of well known layout algorithms. In addition to that, the aesthetic criteria (see Section 2.4) for sequence diagrams are not considered by these algorithms.

Therefore, a new approach specialized for sequence diagrams is developed in this thesis. The ideas and concepts of this approach are introduced in Chapter 4.

2.3 Existing Algorithms and Editors

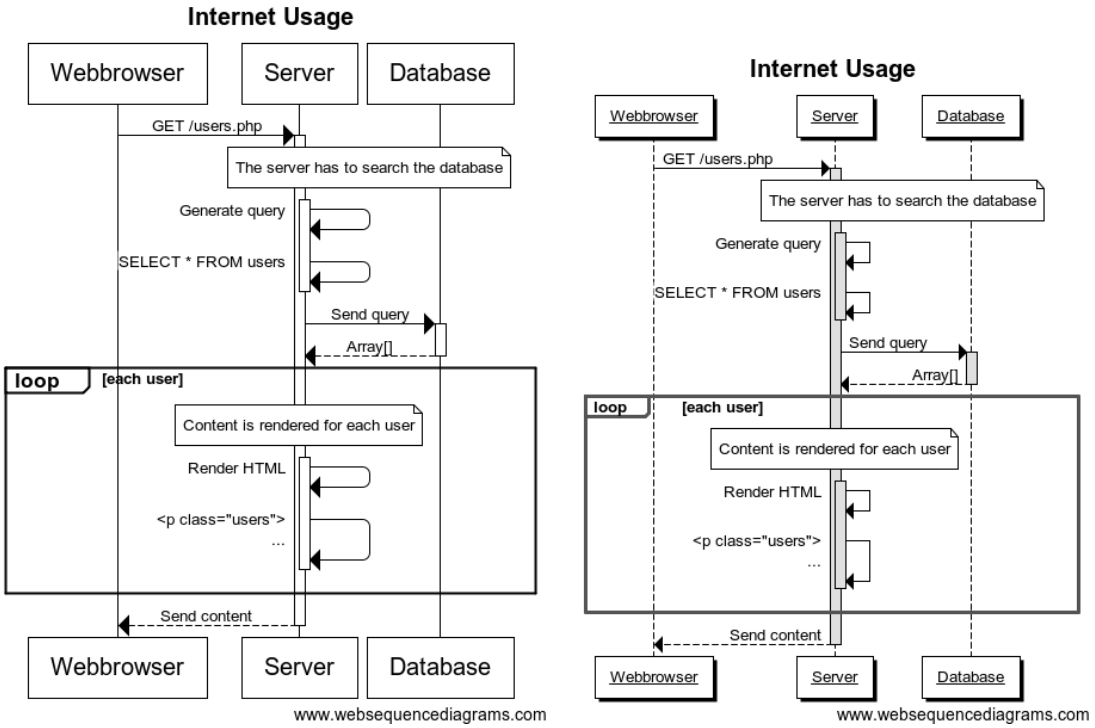
Since sequence diagrams are part of UML, there are several editors available for drawing such diagrams. Some of these editors even offer some kind of automatic layout to the user. Most of the editors that include automatic layout facilities are commercial applications, though. In the following, a selection of these editors is introduced in order to give an overview of the current state of sequence diagram layout.

2.3.1 websequencediagrams

The sequence diagram editor provided by websequencediagrams[2] is a textual editor that uses a very simple syntax. As the name implies, it is a web-based service that does not need to be installed locally. The diagram is displayed as an image in the common PNG format. Upon typing inside the textual input field, the service automatically updates the produced image. The editor provides different styles for the generated image that influence the colors and fonts as well as the shapes of diagram elements. Figure 2.3 shows an example of a sequence diagram drawn using two different styles: spacing is modified, activations are filled (Figure 2.3b) or not (Figure 2.3a), and corners are rounded or not. This affects the width and height of the diagram, which is easily visible in Figure 2.3 too.

Lifelines are placed in the order of their declaration. However, lifelines do not have to be declared. If they are not, the order is derived from the order of their appearance in the textual representation. Messages may have a label that is placed above the message and is aligned either at the beginning of the message or near its center, depending on the style used (see Figure 2.3). They are always drawn horizontally and no message may share the same vertical position with another message except for parallel messages that are supported too. Unlike in UML, messages may only connect lifelines among themselves. Lost and found messages or messages connected to the interaction itself are not supported. Create messages are not supported either. Lifelines may be activated by incoming messages. Likewise, they can be deactivated by outgoing messages.

[2] http://www.websequencediagrams.com



(a) *UML style* drawing. **(b)** *qsd style* drawing.

Figure 2.3. A simple sequence diagram modeled with the *websequencediagrams* editor. Two different styles are used..

Comments may be attached to lifelines in different ways. They can be placed at the left or right side or on top of the lifeline. It is possible to have comments obscuring several lifelines as well. Both is shown in Figure 2.3. The vertical position is determined by the place where the comment is declared in the textual view. Line-breaks, however, have to be inserted manually. If not, the comment will be as wide as its text, which is a waste of space often.

Grouping of messages is supported and common kinds of groups like `alt`, `loop` or `opt` may be nested to any depth. Layout of these groups, however, is very basic. Groups always span over all the lifelines in the diagram without regard for how many of them are connected by the messages inside the group. Figure 2.3 also shows that label placement is problematic in some cases as well. The labels of self-transitions overlap the lifeline on the left side.

2.3.2 Trace Modeler

Trace Modeler[3] is a tool designed for sequence diagram modeling only. It is not integrated into a framework of UML editors and it has a drag and drop interface that is very intuitive and

[3]http://www.tracemodeler.com

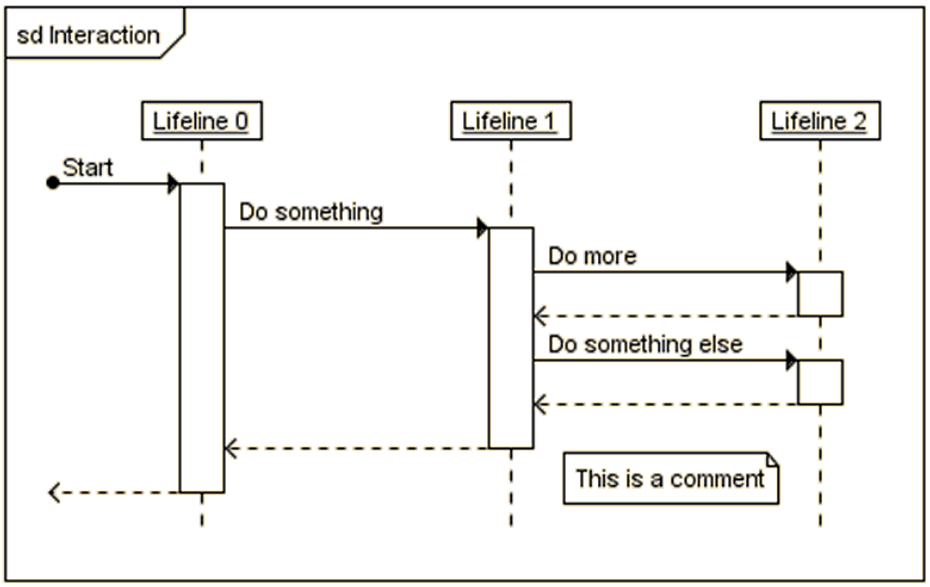


Figure 2.4. A simple sequence diagram modeled with the *Trace Modeler* editor.

powerful. Messages are connected to the right lifelines at the right positions just by dragging them near those positions. The editor provides a feature that highlights the complete trace of the messages with a blue background, which should help the eye to follow the path easily. Several layout options, such as horizontal and vertical spacing, are provided to customize the layout. An example diagram is provided in Figure 2.4.

Lifeline sorting has to be done by the user when inserting the lifeline at the desired position. Lifelines can be inserted with different lifeline headers. Rectangles are supported as well as circles and matchstick men.

However, Trace Modeler is not very flexible. The editor's capabilities for manipulating objects that are already placed are very limited. Activations are automatically inserted at the end of every message and ended by the receipt of its corresponding *reply* message. It is not possible to remove them by hand, hence one is not able to manipulate them in any way. Additionally, it is impossible to have more than one trace in the whole diagram since messages are inserted as *call* and *reply* pair only. The first message always starts at the left border of the diagram. There is no possibility to start at a lifeline. Considering these facts, Trace Modeler only implements a rather limited subset of sequence diagrams as specified in UML.

2.3.3 Quick Sequence Diagram Editor

The *Quick Sequence Diagram Editor*[4], like the *websequencediagrams* editor in Section 2.3.1, is a tool that provides text-to-diagram sequence diagrams. The syntax of the input text is a little

[4]http://sdedit.sourceforge.net/

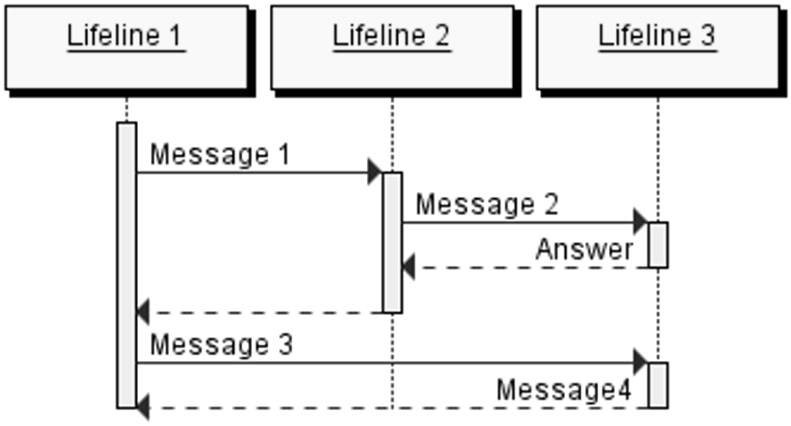


Figure 2.5. A simple sequence diagram modeled with the *Quick Sequence Diagram Editor*.

confusing at first, but supports even advanced features of sequence diagrams. Its appearance is a little programming language style. Identifiers are used together with names and options to define lifelines and messages.

Messages are inserted by referencing the identifiers of the objects a message connects. Reply messages are inserted automatically for each message and the program automatically sorts them, thus allowing just one path through the diagram. Unlike in other editors, the label for the reply messages can be non-empty and is specified together with the first message's label. Activations are inserted automatically at every lifeline for the duration of the `call` (see Figure 2.5).

Areas such as loops or conditionals can be added to the diagram by surrounding the affected lines of text with tags that identify the different kinds of areas. The colors of the different objects can be modified in the preferences as well as the margins between lifelines, messages, and other objects.

To summarize, the editor gives the impression of having been developed especially for software developers that visualize traces of programs by generating the editor's input automatically. It is very well suited for that purpose, the creation of a sequence diagram by hand, however, is a little unhandy.

2.3.4 Effexis Sequence Diagram Editor

The *Sequence Diagram Editor* provided by Effexis[5] is an editor designed especially for sequence diagrams. It is capable of modeling a big subset of the UML specification for sequence diagrams. However, its handling is not very intuitive and not fast at all. Drag and drop is not possible. Instead, the user has to enter all the elements by using an input mask that contains several fields and drop-down menus.

A lifeline is created by selecting the type of the lifeline from a drop-down menu, typing

[5]http://www.effexis.com/sde/

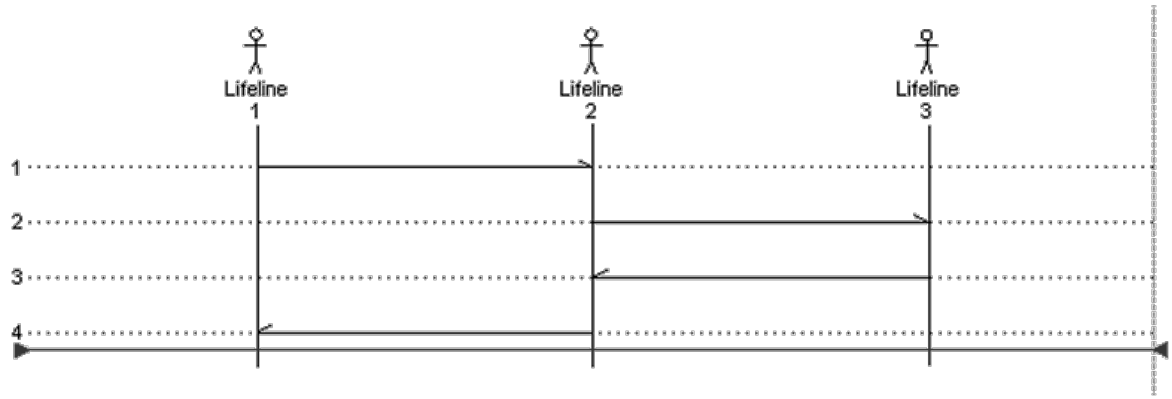


Figure 2.6. A simple sequence diagram modeled with the *Effexis Sequence Diagram Editor*. Dotted lines are visible that show the fixed positioning of the elements. The vertical line on the bottom indicates where the next message will be inserted.

the desired name into a text field and clicking the `Add` button. The position of the lifeline, in turn, is determined by a vertical line within the diagram that can be moved with the mouse. The same holds for the creation of messages. Source and target lifelines, as well as the type of the message, have to be chosen from two drop-down menus. The position of the message, again, is chosen manually by the vertical line within the diagram that turns into a horizontal line (see Figure 2.6) when selecting a message type.

There are different lifeline headers available (for example matchstick men or simple rectangles). The labels of the lifelines, however, are wrapped around even if there is enough space, which leads to odd-looking results in some cases (see Figure 2.6). It is possible to show several levels of detail of the diagram. Different objects such as comments and labels are displayed depending on the desired level of detail.

As a summary, the Sequence Diagram Editor by Effexis has some very nice features, but lacks the usability of other editors.

2.3.5 Altova UModel Sequence Diagram Editor

The *UModel* tool by Altova[6] is an UML tool for software modeling and application development. It is able to generate code out of the diagrams that can be created with the different editors. It is also capable of creating diagrams from reverse engineered code. In addition to that, it automatically generates documentation for the produced code.

Drag and drop is available for all diagram editors, making the creation of diagrams very fast and easy. Automatic layout is claimed to be available for the diagram editors; however, it is only applicable for few of the diagram types. For sequence diagrams no automatic layout is available. This only serves to show that sequence diagram layout seems to be difficult.

[6] http://www.altova.com/umodel/sequence-diagrams.html

Table 2.1. Comparison of the different sequence diagram editors that were examined. The table shows a selection of features and properties of the editors. *Drag & Drop* indicates whether the editor is able to create sequence diagram elements by dragging them from a palette and dropping them into the diagram. *Path* indicates if it is possible to have different paths through the diagram. *Lifeline Sorting* means that the horizontal order of the lifeline is optimized by the layout algorithm. An editor is said to be *customizable* if the spacings or different layout relevant parameters can be changed by the user. At last, the editors are compared in terms of free distribution vs. commercial tools.

Editor Name	Drag & Drop	Path	Lifeline Sorting	Customizable	Free
Websequencediagrams	-	✓	-	(✓) [a]	✓
Trace Modeler	✓	-	-	✓	-
Quick SD Editor	-	-	-	-	✓
Altova UModel [b]	✓	✓	-	-	- [c]
Effexis SD Editor	-	✓	-	-	- [d]
Papyrus SD Editor	✓	✓	✓	✓	✓

[a] Different layout styles are available.
[b] The Altova UModel editor does not provide automatic layout for sequence diagrams.
[c] Windows only
[d] Windows only

2.3.6 Comparison

As shown, there are several editors that have very different capabilities. Table 2.1 summarizes these differences. The check marks are not very numerous, indicating that there are only few editors that provide many helpful features. Especially the free editors lack most of the features that were examined in this comparison. However, the commercial solutions do not provide much more functionality in general either. In particular, the lack of any lifeline sorting functionality is noticeable. One could reason that lifeline sorting is a task that can be easily done by hand, but especially for diagrams that contain many participants the ability to rearrange the lifelines automatically according to several aesthetic criteria can be very useful to maintain the readability of big diagrams.

None of these editors is able to display more than one message at the same vertical position even if the messages do not overlap in any way. Even worse, most editors do not support more than exactly one trace through the diagram. In addition to that, layout is done implicitly by most of them, leaving the user no possibility to influence the layout. Most of the tools re-layout the diagram after each modification. The Altova UModel tool, however, provides layout as an option, which is disabled for sequence diagrams since there is no support for this diagram type.

Considering this, the sequence diagram editor of the Papyrus framework with its full UML support in combination with the layout facilities developed in this thesis is a very powerful tool with regard to sequence diagrams. It combines all the features that are desirable for the work with these diagrams. Nevertheless, it is a free open source tool.

The next section introduces a variety of aesthetic criteria that may be applied to graphs

and to sequence diagrams in particular. It also states which of these criteria we can consider useful for the evaluation of sequence diagram layout.

2.4 Aesthetic Criteria

The quality of a layout is a very subjective matter. Especially for complex diagrams it is very user-dependent what is considered a good layout. However, in order to have a somewhat objective measurement for the quality of a layout, it is necessary to define some aesthetic criteria that are able to represent this quality. There has already been a lot of research on this topic and there are common aesthetic criteria that have been established. A selection of criteria as proposed by Poranen et al. is listed below [PMN03].

- Edge crossings: The number of edge crossings in the diagram. Fewer crossings lead to a more readable drawing since it is easier to follow a line that is not intersected by another line.
- Edge length: Shorter edges imply better readability because the eye of the user does not have to follow the line for a long way. There are three different versions of this criterion:
  - Maximum: The longest edge in the drawing.
  - Sum: The sum of the length of all edges in the drawing.
  - Uniformity: The uniformity of the length of the edges in the drawing.
- Area: The size of the smallest rectangle containing the whole drawing. Smaller drawings are desirable in order to be able to view the diagram without scrolling or with scrolling as little as possible.
- Bends: The number of edge bends. As mentioned above, it is important that the eye is able to easily follow the edges through the diagram. Too many edge bends make this difficult.
  - Maximum: The maximum number of bends of a single edge.
  - Total: The total number of bends.
  - Uniformity: The uniformity of the number of bends per edge.
- Angle: The angle between two edges connected to the same node. It is desirable to have as big angles as possible in order to be able to distinguish the edges easily.
- Aspect ratio: The aspect ratio of the whole diagram. The optimal solution depends on the medium that is used to show the diagram. Computer screens need a wide diagram whereas tall diagrams are preferred for paper.
- Balance: The balance of the nodes over the drawing area. There should not be areas where no nodes are located; neither should the nodes be clumped in other areas.

So there are formal and common criteria for the aesthetics of graphs. However, sequence diagrams are not exactly graphs. For that reason, most of these criteria are not desirable or even applicable to sequence diagrams. In this thesis, the focus is on two of them, namely *edge length* and *edge crossings* (see Section 6.1 for detailed justification of this choice). Since edges only cross lifelines and lifelines have (nearly) equal distances between them, the two criteria are strongly related.

For sequence diagrams it is generally desirable for the lifeline that starts the communication process to be placed at the leftmost position. This is a property that cannot be easily adapted to normal graphs. Source nodes are placed leftmost in data-flow diagrams, but the lifeline with the uppermost outgoing message can not be regarded as a source node in general since it usually has incoming messages too. Nevertheless, it is an aesthetic criterion that is well accepted for sequence diagrams as it improves readability of these very much [PMN03].

In sequence diagrams, messages can be grouped together by several constructs such as combined fragments and interaction operands. Obviously, it is desirable to have the lifelines that are connected by these messages placed near each other. Additionally, the "slidability" was proposed by Poranen et al. [PMN03] as a new sequence diagram specific aesthetic criterion. It measures if it is possible to view all the relevant information in a window of fixed size

The next chapter introduces some technologies this thesis builds upon. This includes the Eclipse framework and related projects, such as the Graphical Modeling Framework (GMF), KIELER, and the *Papyrus* framework.

Chapter 3

Technologies

Since the layout algorithm that is dealt with in this thesis is part of a large research project, it is necessary to introduce some tools prior to the description of the algorithm itself.

3.1 Eclipse

The Eclipse project is a plug-in-based application framework based on Java [Ecl08]. The plug-in management of Eclipse is done by the OSGi [OSG08] framework Equinox that was developed especially for the Eclipse project. With that, software components can be loaded, updated and unloaded at runtime.

One of the most important components is the Java Development Tools (JDT) bundle. With the JDT loaded, Eclipse acts as an Integrated Development Environment (IDE) for Java development. Meanwhile, there are several extensions that allow users to develop software with different kinds of programming languages. C/C++ (with the C/C++ Development Tools (CDT)), PHP, Cobol, and even Android App programming is possible with the corresponding plug-ins. This makes the Eclipse project a widespread software development tool.

Its run-time kernel is very minimalistic, its extensibility by plug-ins in favor is huge. The Eclipse project was released out of an internal IBM project in 2001. In 2004, IBM decided to found the independent Eclipse Foundation that is responsible for the further development of the Eclipse Framework. Since Eclipse is open-source software, there are many third-party plug-ins for different purposes.

There are several important commercial as well as open source projects related to Eclipse. Some projects use Eclipse as a platform, others are designed to run as parts of different Eclipse projects. In the following, some projects that are related to this thesis will be introduced.

3.2 Graphical Modeling Framework

The GMF is a framework that simplifies the process of building graphical editors for the Eclipse framework. It is based on two further Eclipse frameworks, the Eclipse Modeling Framework (EMF) and the Graphical Editing Framework (GEF).

EMF is an Eclipse framework for building tools and other applications based on a structured data model. It is capable of generating Java classes for a given meta model. Furthermore, it provides adapter classes for viewing and editing the model and a basic editor.

GEF is an Eclipse framework designed to generate rich graphical editors. These editors include a tool palette for drag-and-drop creation of graphical elements, adapted to the customized figures generated together with the editor.

Along with the figures, *EditParts* are generated. EditParts can be understood as the *controller* that is responsible for making changes to the underlying model. *Requests* are used to communicate with the EditParts. For each EditPart, there is a corresponding *EditPolicy* that handles the incoming Requests and transforms them into appropriate *Commands*. Commands are organized by *Command Stacks* that provide the useful *undo* and *redo* operations. They are responsible for applying the requested changes to the model.

With the help of these two projects, GMF is able to create graphical editors along with all the necessary code and tooling. The ability of creating simple, fully-functional editors in very few steps is the main reason for its popularity. There is a huge amount of GMF-based editors in use all over the world.

3.3 KIELER

The KIELER is a research project founded by the Real-Time and Embedded Systems Group at the *Christian-Abrechts-Universität zu Kiel* [FvH09]. Its focus is on the pragmatics of graphical model-based design of complex systems. One of its main components is an infrastructure that provides automatic layout algorithms to graphical editors. There are several algorithms for different kinds of graph structures and different optimization goals. The most important sub projects with respect to automatic layout will be introduced in the following. However, automatic layout of static diagrams is just one part of KIELER's capabilities. Another research focus is in the area of pragmatics which includes the view management that is responsible for dynamic visualizations in diagrams. Recent research here points at *focus on context*, which dynamically collapses parts of the diagram (e.g. edge labels or sub-states) that are not important for the current context. Additionally, KIELER provides possibilities to simulate several kinds of systems that are modeled with KIELER's editors. Main focus of the research in the simulation domain is concentrated on the simulation of synchronous languages like *ESTEREL* together with related graphical languages such as *SyncCharts*.

3.3.1 KIML

KIML is the core component of the KIELER layout facilities. It connects the various graphical editors and viewers that are supported by KIELER with the layout algorithms. As shown in Figure 3.1, KIML extracts the graph from the diagram editor, transforms the graph into its internal graph structure, which is then passed to the chosen layout algorithm. When the layout algorithm is finished, its results are applied to the diagram editor.

To be able to handle very different kinds of graphs coming from different types of editors, KIML provides an intermediate graph format. This so called `KGraph` is adapted to the layout algorithms that are part of KIELER it is able to handle a very wide range of graphs though.

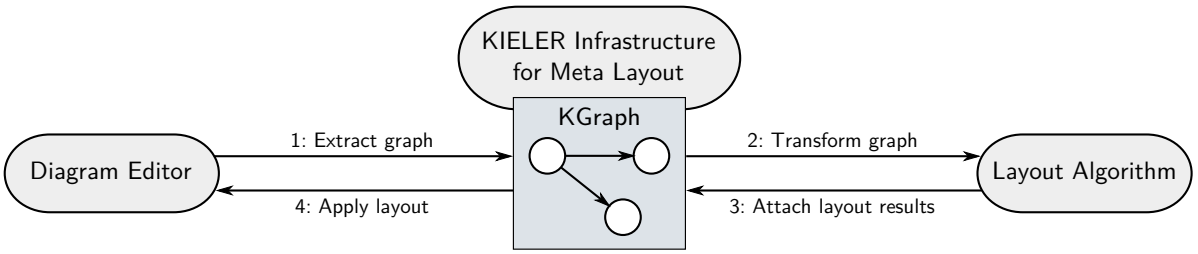


Figure 3.1. Layout process: KIML is the interface between the diagram editor and the layout algorithm.

Chapter 5 introduces the data structure more precisely. Different layout algorithms are available through different sources. KIML is able to use several open source graph layout libraries such as the Open Graph Drawing Framework (OGDF) or GraphViz. However, through the KIELER Layout Algorithms (KLay) project, several custom graph layout algorithms are provided too.

Besides acting as a connector between graphical editors and layout algorithms, KIML's main benefit is its capability to configure the layout process in-depth. This is especially important for supporting different types of graphs since they do have different requirements towards layout. In particular, they have different aesthetic criteria that should be met by the layout. To be able to meet these demands, KIML provides a set of layout options that are displayed in the Layout View amongst others. These options offer the choice between different layout algorithms. Additionally, several parameters such as spacing and orientation can be modified through the Layout View. The different layout algorithms, in turn, may offer their own options that are provided through the Layout View too.

3.3.2 KLay

The KLay project is a collection of different layout algorithms implemented in Java. These layout algorithms are integrated with KIML to be used with all kinds of graphs that are supported by KIML. Currently, three fundamental layout algorithms are offered by KLay:

- KLay Layered: the most advanced layout algorithm in the KIELER project. It is based on the layer-based hierarchical approach by Sugiyama et al. [STT81]. However, the three phases described by Sugiyama et al. are extended to five layout phases. Intermediate processors between each of the phases ensure as much flexibility as possible. Each layout phase declares which intermediate processors it requires. With that, different implementations for each of the phases can be combined freely to match the characteristics of different kinds of diagrams. KLay Layered, however, is mainly intended for diagrams with an inherent data flow direction as their data flow is emphasized by this approach.
- KLay Force: a layout algorithm based on physical models. It interprets edges as springs pulling nodes towards one another. In addition to that another force is introduced between each pair of nodes, pushing them further apart. With that, nodes are placed homogeneously over the diagram and edge length's are made homogeneously as well.

- KLay Planar: a layout algorithm that is based on the planarization of the graph. Edges are routed in an orthogonal way. Nodes and edges are placed on a given grid in the diagram. This approach especially aims at minimizing the number of edge-crossings in the drawing.

3.4 Papyrus

The Papyrus framework [1] is a framework designed for multiple purposes related to UML diagrams. It is capable of editing and creating all kinds of UML diagrams. As a part of the Model Development Tools (MDT) project, it is an official Eclipse project. The Papyrus framework was initialized and founded by the French Commission for Atomic Energy and Alternative Energies (CEA) that runs a technical research institute *CEA LIST*.

Papyrus provides an integrated environment for editing any kind of EMF-based diagrams. Especially, UML and related modeling languages are addressed. These modeling languages particularly include the Systems Modeling Language (SysML) and the Modeling and Analysis of Real-Time and Embedded systems (MARTE) project. Special editors for UML 2, SysML and other modeling languages are delivered with the Papyrus framework. In addition to that, the project provides components to integrate these editors into different Model Driven Software Development (MDSD) and Model-Based Design (MBD) tools. UML profiles, which are extensions of the UML meta model, are supported too, enabling the user to define own editors for DSLs based on the UML 2 standard.

This thesis introduces a new layout algorithm for UML sequence diagrams into the Papyrus project. The next chapter introduces the theoretical concepts of this algorithm.

[1] http://www.eclipse.org/modeling/mdt/papyrus/

Chapter 4

Theory and Concepts

This chapter introduces the central concepts used in this thesis as well as the theoretical foundations necessary to understand them. First Section 4.1 starts by giving some definitions of the underlying theory and elements that are used in sequence diagrams. Section 4.2 introduces the layout process before Section 4.3 talks about those elements of sequence diagrams that are easy to lay out. Section 4.4 about the assignment of layers to the messages then starts the block of more challenging layout issues. Section 4.5 continues by describing lifeline sorting strategies. Finally, Section 4.6 closes this chapter by describing the problem of label placement which tends to be neglected in various graph layout algorithms.

4.1 Theoretical Foundations

This section presents the definitions and explanations necessary to understand the following sections and chapters. There are two main topics for that: the definitions related to graphs and those that are related to sequence diagrams.

4.1.1 Graph Definitions

The most basic definition that is important when handling graph problems is the definition of a graph itself.

4.1 Definition. A *graph* G is a pair (V, E) of sets of *vertices* and *edges*. Vertices v may also be called *nodes* in the following.

4.2 Definition. An *edge* e of a graph G is a set $\{u, v\}$, where u and v are vertices of the graph.

4.3 Definition. An edge may be *weighted* in some graphs. The weight of an edge e is denoted $w(e)$. If an edge has no attached weight, the weight is assumed to be 1.

4.4 Definition. The *degree* $d(v)$ of a node v is the number of edges that are connected to v:

$$d(v) := \sum_{\{u,v\} \in E} 1$$

4.5 Definition. The *weighted degree* $d'(v)$ of a node v is the sum of the edges weights of edges that are connected to v:

$$d'(v) := \sum_{\{u,v\} \in E} w(\{u, v\})$$

So far, these definitions describe *undirected* graphs. However, for many purposes this is not sufficient. Edges often have a predefined direction that should be modeled too.

4.6 Definition. A graph G is said to be *directed* if edges have a direction, that is, if the pair (u, v) is ordered. In this case, u is called the *source node* and v is called the *target node*.

For certain layout algorithms, including the one described here, a graph has to be partitioned into several *layers*. Splitting a directed, acyclic graph into distinct layers is a well known and commonly used approach since Sugiyama, Tagawa and Toda [STT81] published their work on layered graph drawing. The partition of the nodes then holds the following properties.

4.7 Definition. A *layering* is a partition of the set of nodes of a given graph into several numbered layers such that the following properties are satisfied.

1. Every node belongs to exactly one layer.
2. Every layer contains at least one node.
3. There are no edges connecting nodes that share the same layer.
4. Every edge has a source node that is in a layer with number lower than the one of the target node's layer.

4.8 Definition. A *path* p of length n in a graph G is a finite sequence of edges

$$p := \{v_0, v_1\}, \{v_1, v_2\}, ..., \{v_{n-2}, v_{n-1}\}, \{v_{n-1}, v_n\}$$

that connects node v_0 (the *start node*) with node v_n (the *end node*).

4.9 Definition. A *connected component* C of a graph G is a set of nodes that are connected among each other. In addition to this, these nodes are not connected to any node that is not part of C. More formally, a connected component $C \subseteq L$ is a set of nodes for which the following holds:

$$\forall c_1, c_2 \in C : \exists p \in G, c_1 = start(p) \wedge c_2 = end(p)$$

and

$$\forall c_1 \in C, c_2 \notin C : \nexists p \in G, c_1 = start(p) \wedge c_2 = end(p)$$

The definitions presented were general definitions of graphs which are important to understand the graph-based data structures used to represent different diagram types. In the following, definitions specific to sequence diagrams will be presented.

4.1.2 Sequence Diagram Definitions

This thesis is about the layout of sequence diagrams. Hence, the basic terminology for such diagrams is presented here.

4.10 Definition. A *sequence diagram*[1] is a tuple $S := (L, M, A, C)$ of a set of lifelines L, a set of messages M, a set of areas A and a set of comments C. It is a special kind of diagram that visualizes the communication of different processes. It is often also called *interaction diagram* and is part of the UML.

4.11 Definition. A *lifeline* $l \in L$ is an interaction partner or process in a sequence diagram. It is drawn as a vertical line that is decorated with a *lifeline header* at the top, which is a rectangle with the lifeline's name in it.

4.12 Definition. A *message* is a tuple $m := (s, t)$ that contains a *source* object $source(m) := s$ and a *target* object $target(m) := t$. It is a connection between two objects that in most cases are lifelines. Other than this, messages may also originate from or point at the surrounding interaction. A message is drawn as a horizontal arrow connecting the lines of its lifelines. There are several types of messages:

1. Asynchronous messages are drawn as solid arrows with a stick arrowhead (see *Message* in Figure 4.1).
2. Synchronous messages are drawn as solid arrows with a filled arrowhead.
3. Reply messages are drawn as dashed arrows with a stick arrowhead.
4. Create messages are drawn as dashed arrows with a stick arrowhead that points to a lifeline's head (see *Message6* in Figure 4.1).
5. Delete messages are drawn as a solid arrow with a filled arrowhead pointing to the end of a lifeline (see *Message7* in Figure 4.1).
6. Lost messages are asynchronous messages that do not point at any lifeline but at a filled circle that is placed next to the source lifeline at the right side. Their target lifeline is undefined (see *Message5* in Figure 4.1).
7. Found messages are asynchronous messages that do not originate from a lifeline but from a filled circle that is located next to the target lifeline at the left side. Their source lifeline is undefined (see *Message1* in Figure 4.1).

With the definition of messages in mind, the incoming and outgoing messages of a lifeline may be defined now.

4.13 Definition. For a lifeline l, the set of *outgoing messages* id defined by $outgoing(l) := \{m \in M \mid source(m) = l\}$.

4.14 Definition. For a lifeline l, the set of *incoming messages* id defined by $incoming(l) := \{m \in M \mid target(m) = l\}$.

[1] http://www.ibm.com/developerworks/rational/library/3101.html

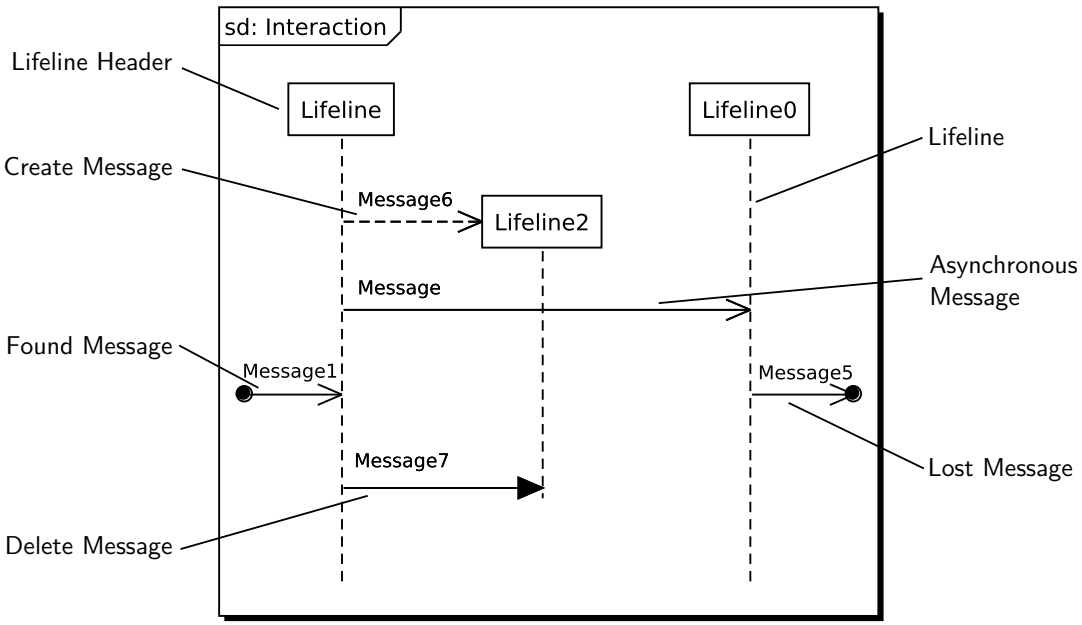


Figure 4.1. An example of lifelines and different message types.

4.15 Definition. A message's *send* or *receive event* is the point at the lifeline, where the message leaves or arrives at the lifeline. It is not graphically highlighted normally, but comments may be attached to it.

In addition to these very basic elements of sequence diagrams, there are some advanced elements that should be mentioned here too. The first of these are execution specifications.

4.16 Definition. An *execution specification*, also called *method call boxes* or *activations*, are opaque rectangles that are placed on top of the dashed line of a lifeline. They represent processes that are started in response to the arrival of a message. Execution specifications may extend other execution specifications by building "sub-executions". Again, this may only happen when a message is received at the starting point of this execution specification. Examples of these elements are given in Figure 4.2

While execution specifications are used to describe the inner structure of the object that is represented by its lifeline, there are different kinds of outer structuring elements. Messages can be grouped in sequence diagrams by surrounding them with different kinds of blocks.

4.17 Definition. An *area* $a := \{m \in M \mid m \text{ is completely covered by } a\}$ is a set of messages that are contained in the area.

There are different kinds of areas which are defined in the following.

4.18 Definition. An *interaction use* area is a transparent rectangle that covers a set of messages in order to show their close relation. This may indicate that these interacting lifelines and

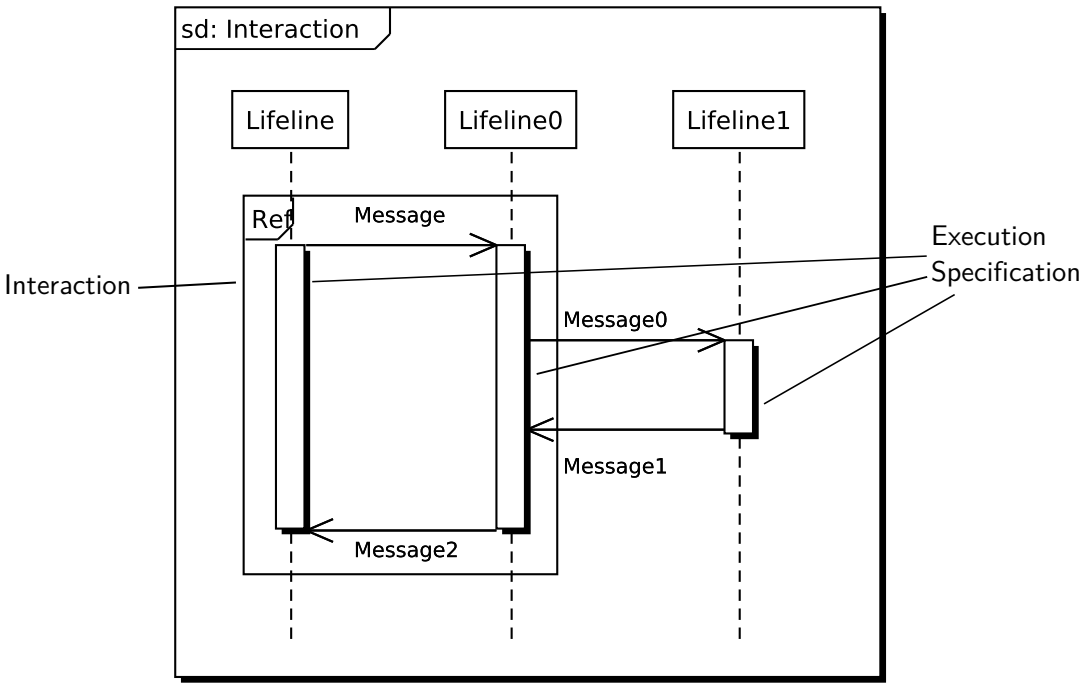


Figure 4.2. An example diagram with execution specifications and an interaction. Execution specifications are attached to every lifeline to highlight the active part of the lifeline. An interaction visualizes a relationship of *Message* and *Message2* by grouping them together.

messages could be a sequence diagram on its own. Furthermore, the interaction may have a label that yields instructions on how to interpret it. For example, labels may contain "repeat this x times" or other instructions. An example of an interaction use is given in Figure 4.2.

Another element for grouping messages is the combined fragment.

4.19 Definition. A *combined fragment* is a transparent rectangle that covers a set of messages. It is divided into a header section and several sub-areas that cover a set of messages each. Each of the subsections may have a label that indicates how to interpret it. One use-case of combined fragments is to describe *if-then-else* blocks.

With the foundation of these definitions, the theoretical ideas behind the sequence diagram layout algorithm can be introduced. The next section will explain the basic concepts of the layout process before going into the detailed explanation of the different steps that are performed during the layout process.

4.2 The Layout Process

Given a sequence diagram in some editor's representation, the first step will always be to convert this sequence diagram representation into a structure that is specialized for the use

in the layout algorithm. With a sequence diagram as defined in Section 4.1, the sequence diagram layout algorithm then is able to compute a layout for the diagram by calculating positions and sizes for all the elements of the sequence diagram. To be able to do so, several preparation steps are required to be computed. Applying the calculated coordinates to the sequence diagram is only the final step in a chain of algorithms.

The next section will explain the concepts that are used to place those elements that do not need any complicated algorithms for their layout before going on to more advanced concepts in later sections.

4.3 Foundations of Sequence Diagram Layout

Sequence diagrams have a very strict and well defined graphical syntax that restricts the flexibility of a layout algorithm. While this simplifies the layout of these diagrams a lot, it also limits the possibilities for improving the graphical appearance of a given diagram. This section deals with the simple tasks that are related to these restrictions.

4.3.1 Lifelines

Lifelines have to be drawn side-by-side at the top of the diagram. They normally share the same height, vertical position, and usually the same width as well. Hence, the only degree of freedom is their horizontal position, which is handled in Section 4.5. The height and vertical position of a lifeline may only be altered if it has incoming create messages or delete messages. If so, the lifeline's vertical position has to be moved to the position of the create-message. The height then has to be modified such that the lower border of the lifeline matches the other lifeline's lower border. The height may also be modified if a delete message points to the lifeline. Its lower border should be located near the position of the delete message. The width of a lifeline may depend on its label in some cases. However, this should not be modified by the layout algorithm.

4.3.2 Messages

Messages are usually drawn as straight, horizontal lines without any bend points in sequence diagrams. They directly connect their associated lifelines in the shortest way. Therefore, their send and receive events are located at the same height normally. This height is constrained by the messages among each other. Their order at each lifeline has to be preserved through the layout process. In the approach proposed in this thesis, the messages are organized in layers, which is described in Section 4.4.

4.3.3 Execution Specifications

Execution specifications are placed on the vertical line of their lifeline. They have a predefined width and are horizontally centered on their lifeline. Their height and position corresponds to

the vertical position of the connected messages: an execution specification starts at the point where its first message is connected and ends at the last message's vertical position. Execution specifications may contain other execution specifications that are connected to messages that cover a subset of the executions specification's vertical space. These "sub-executions" are located on top of the original execution specification, they are drawn slightly rightwards so that they overlap with their "parent" only on half width but do not completely cover the original execution specification.

4.3.4 Interactions

Interactions and other elements like combined fragments are rectangular areas that cover a certain set of messages. Since these areas group the messages they cover into sub-diagrams, they should be drawn as a rectangle that covers all of their messages plus a certain margin. Messages that are not completely covered by the area are not considered to be part of it.

4.4 Layer Assignment

This section introduces the concept of layer assignment to messages. This is necessary to determine the vertical position of messages. Most sequence diagrams consists of messages that build a continuous path through the diagram. They are not restricted to this scenario, however. In either case the relative order of the messages that are connected to the same lifeline is important. Messages that do not share the same lifeline may be sorted freely. In particular, they may share the same y coordinate if that does not cause them to overlap.

In order to model this the main idea is to assign a layer number to each message. Contrary to the common use of layering in graph layout, layers in this context are not collections of nodes that have the same horizontal position. Instead, layers are collections of messages that share the same vertical position. Layer numbers increase from the topmost messages downwards. At every lifeline, every pair of messages has a relative order. Hence, for each of these pairs the lower message should have a higher layer number than the upper one.

To be able to use common layering algorithms, a new, temporary graph has to be build based on the structure of the sequence diagram. As shown in Figure 4.3, each message maps to a new node in the temporary graph. Directed edges are introduced for every neighbored pair of messages that share the same lifeline, connecting the corresponding nodes. The neighborhood relation here indicates vertical distance between the messages at the shared lifeline. The direction of the edges is from the message with higher position to the one with lower position (regarding the lifeline that they share).

The directed graph that is generated this way can then be processed by any common layering algorithm. Since the sequence diagram layout algorithm is integrated into the KIELER project, it seems reasonable to use KIELERs KLay project for this. The KLay project includes the KLay layered project, which in turn includes several implementations of graph layering algorithms. The *longest path layerer* and the *network simplex layerer* were used in this approach.

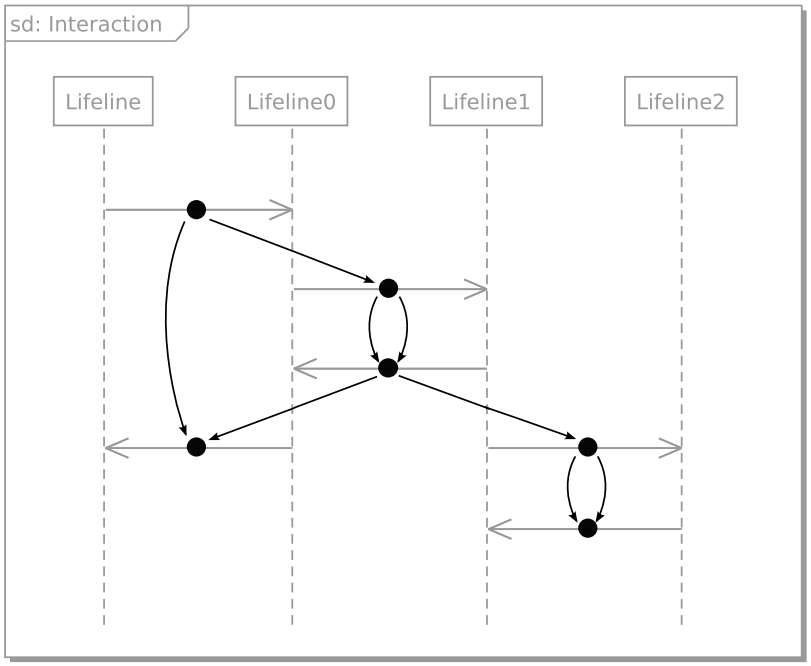


Figure 4.3. Correlation of message relations and edges in the layered graph. Edges are inserted for every pair of neighbored messages at a lifeline.

The network simplex layerer turned out to be the algorithm of choice because it yields more compact results than the longest path layerer.

The calculated layering of the messages is important for some of the lifeline sorting algorithms that are introduced in the next section. They compute the horizontal order of the lifelines, which strongly influences the look of the diagram.

4.5 Lifeline Sorting

Lifelines may be sorted freely over the horizontal range of the diagram. This is the most interesting part of the layout for sequence diagrams since it provides the largest opportunity for improving the readability of a sequence diagram. There are different approaches to the arrangement of lifelines, each targeting different optimization goals. It is very difficult to combine these goals since some of them contradict each other [PMN03]. If, for example, the lifeline that starts the communication process is highly connected to different lifelines, it would be desirable to place it between the connected lifelines in order to minimize the length of the messages. This contradicts the optimization goal of placing the starting lifeline on the left side of the diagram.

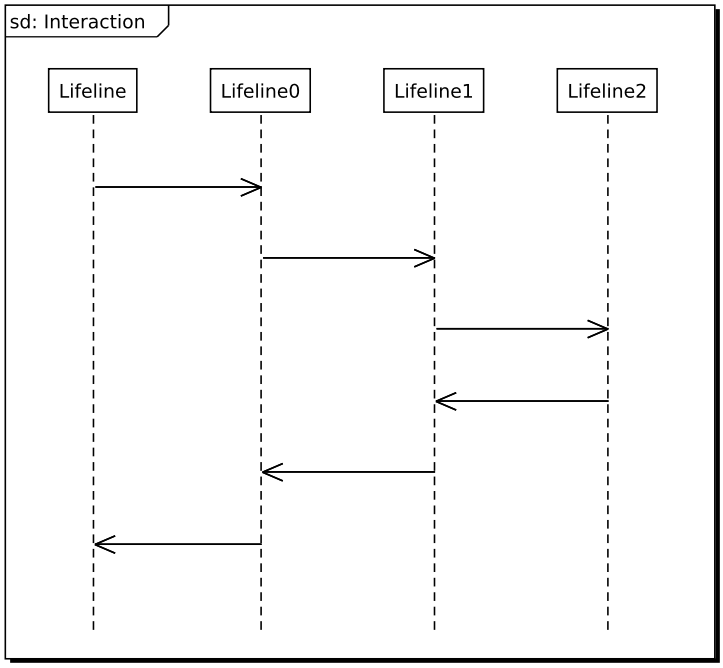


Figure 4.4. Lifeline sorting that places the messages in a stairway-like fashion.

4.5.1 Layer-Based

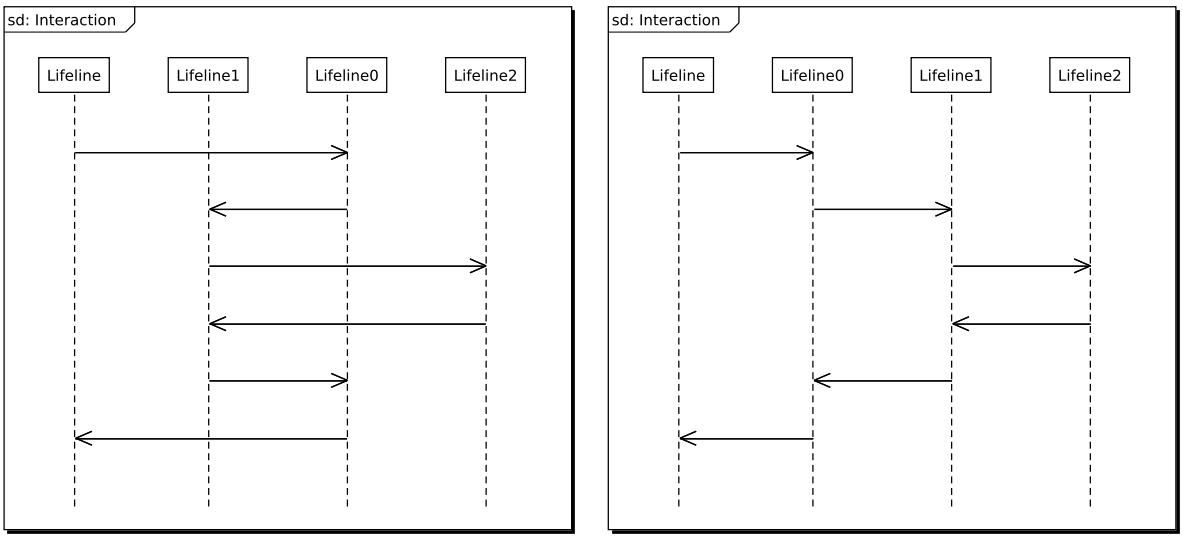
As a first approach, we developed a layer-based lifeline sorting algorithm that aims at making it easy to follow the flow of messages in the diagram. As the name implies, it depends on the layering of the messages that is done in a previous step by the layering algorithm, producing a layout that places messages in a stairway-like fashion (see Figure 4.4). The lifelines that are connected by a message of the uppermost layer are placed leftmost in the diagram (lines 4 to 6 in Listing 4.1). This implies that the message's source lifeline has to be the leftmost one. Its target lifeline is placed next in order to minimize the message's length. Afterwards the uppermost outgoing message of that lifeline is looked for (lines 12 to 13). Its target lifeline is set next and so forth. If a target lifeline was already set in a previous step, it obviously cannot be set again. Therefore, only those messages are looked for, whose target lifeline was not assigned a position yet. This step is iterated as long as there are outgoing messages with that property. If the sequence diagram is divided into several connected components, the whole process is repeated for each of them.

4.5.2 Avoiding Long Messages

The next approach to lifeline sorting that was implemented for the sequence diagram layout algorithm in this thesis is related to avoiding long messages in the diagram. Avoiding long messages is highly related to crossing minimization for sequence diagrams, since long messages cross lifelines. Every lifeline that is crossed by a message increases the length by

Listing 4.1 The algorithm for *layer-based* lifeline sorting. Lifelines are placed one after another from left to right.

```
1:  function SortLifelines(L)
2:      p ← 0
3:      while L not empty do
4:          M ← {m | x ∈ L ∧ m ∈ outgoing(x) ∧ target(m) ∈ L}
5:          m_0 ← min_{m∈M} layer(m)
6:          x = source(m_0)
7:          repeat
8:              place x on position p
9:              p++
10:             L ← L\{x}
11:             x ← target(m_0)
12:             M ← {m | m ∈ outgoing(x) ∧ target(m) ∈ L}
13:             m_0 ← {m | min_{m∈M} layer(m)}
14:         until m_0 is null
15:     end while
16: end function
```



(a) Lifeline sorting that yields too many message-lifeline crossings.

(b) Optimal lifeline sorting that yields no message-lifeline crossings.

Figure 4.5. Comparison of differently ordered lifelines: the readability of the diagram is decreased a lot if too many message-lifeline crossings occur.

the lifeline's width and spacing. Therefore the minimization of these crossings is crucial for the readability of the diagram. Figure 4.5 illustrates this by comparing differently ordered lifelines; Figure 4.5a contains too many message-lifeline crossings, whereas Figure 4.5b does not have any crossings.

This minimization can be divided into the avoidance of very long messages and the minimization of the summed length. The problem of minimizing the maximum length of messages can be transformed to the problem known as bandwidth [PMN03], shown to be NP-complete by Papadimitriou [Pap76]. The bandwidth problem is defined as the minimization of the length of the longest edge. Strongly related to this is the problem known as the profile problem, which minimizes the length of the longest leftwards leading edge. However, these are not as interesting as the minimization of the summed length of all messages: a single long message may be tolerated if messages are short in general, whereas several long messages could degrade the readability of the diagram much more.

Minimization of the summed length of messages was shown to be equivalent to the linear arrangement problem for general graphs by McAllister [McA99]. Let $f : V \rightarrow \{1, 2, ..., |V|\}$ be an arrangement function associating a unique integer value with every node. The weighted sum of the edge length's for a given graph G and an arrangement function f is defined as

$$L(G, f) := \sum_{e \in E} l(e)w(e)$$

with $l(e) := |f(u) - f(v)|$ defining the length of an edge $e = \{u, v\}$ and $w(e)$ defining its weight. The linear arrangement problem for a graph G then is the minimization of L by choosing a proper arrangement function f.

In order to be able to solve the lifeline sorting problem by solving the linear arrangement problem, a temporary graph is created based on the structure of the sequence diagram:

▷ For each lifeline, a new node is created.

▷ For nodes whose lifelines are connected by messages, an edge is inserted.

▷ The weight of the edges is the number of messages that connect the corresponding lifelines.

With that, the arrangement of the nodes in the temporary graph is exactly the linear arrangement problem, which was shown to be NP-complete by Garey, Johnson and Stockmeyer [GJS76].

However, there are several heuristics that attempt to solve the problem in polynomial time. Several heuristic algorithms for the related bandwidth and profile problems are divided into two basic steps: first, select a starting node that is placed in first position by some kind of selection function. After that, place the remaining nodes one after another according to another selection function that evaluates the connections of the remaining nodes in order to find a solution.

McAllister [McA99] proposed such a two-phased heuristic approach to solve the linear arrangement problem. It reuses common heuristics for the bandwidth and profile problems. These heuristics do not consider edge weights, since these weights do not affect the bandwidth and profile problems. This is because these problems optimize the length of a single (the longest) edge. For the linear arrangement problem, however, edge weights do affect the optimal solution, since the weighted sum of edges is minimized. Therefore, the author combines the unweighted approaches to generate a heuristic for the weighted problem.

In the first phase, the heuristic by McAllister chooses the node with the lowest degree to be the one that is placed first. This corresponds to the strategy that is used by other two-phased algorithms. However, in contrary to these algorithms, the weighted degree, which considers edge weights, is used here. The second phase then makes the real difference. For each node, several values are computed in order to calculate the best node to be placed next. Let $P \subseteq V$ denote the set of nodes that have already been placed in earlier steps. Similar to that, $U \subseteq V$ denotes the set of nodes that have not been placed yet. Let then $F(G) := \{v \in V | \{u, v\} \in E \wedge u \in U \wedge v \in P\}$ be the *front* of a partially arranged graph. The first of the values that is needed to compute, is the $tl(v)$ value that measures, how highly connected to the nodes that where already placed a node v is.

$$tl(v) := \sum_{u \in P, \{u,v\} \in E} w(\{u, v\})$$

Likewise, the $tr(v)$ value measures the connectivity of v to unplaced nodes.

$$tr(v) := \sum_{u \in U, \{u,v\} \in E} w(\{u, v\}) = d'(v) - tl(v)$$

The selection factor $sf(v)$ is then calculated by

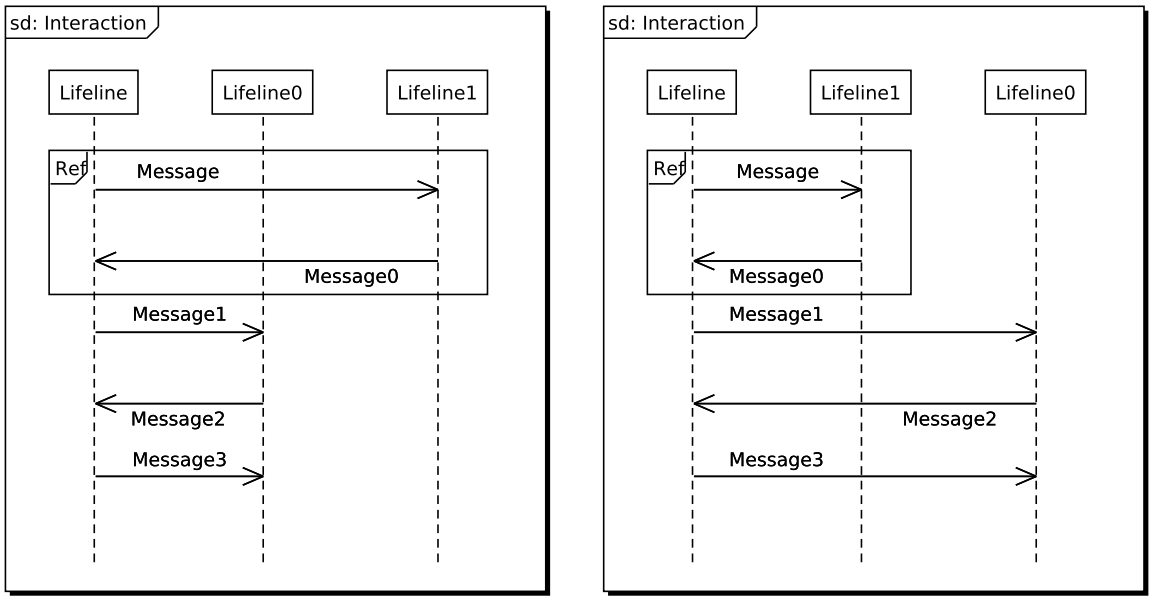
$$sf(v) := tr(v) - tl(v) = d'(v) - 2tl(v)$$

In each step, the node $v \in F$ with the lowest selection factor $sf(v)$ is selected to be placed next. After that, the tl and sf values of each node connected to v have to be updated. If $F = \emptyset$, the whole connected component has been placed. For the remaining nodes, the whole process is started again.

However, there are some modifications to the algorithm forced by the characteristics of sequence diagrams. Messages may be connected to the surrounding interaction, which is not modeled as a node in the temporary graph since its position is all around the lifelines. Additionally, there may be *lost* or *found* messages or self-loop messages. These are connected to one lifeline only. Messages that are connected to the surrounding interaction should be drawn from left to right. Therefore, such messages, which have a lifeline as their source and the interaction as their target, introduce a penalty to their corresponding source node's tl value in the temporary graph. This forces the node to be placed later than it would have been without the penalty, which minimizes the length of the message that introduced the penalty. Similarly, messages that have the interaction as source and a lifeline as target get an advantage to their corresponding target node's tl value, which forces them to be placed earlier.

4.5.3 Pivoted Long Message Avoidance

Considering the structure of the two-step heuristic for the linear arrangement problem, I adapted the first step to sequence diagrams. As already mentioned, it is generally desirable for the lifeline that starts the communication process to be placed at the leftmost position. Whereas approaches to the general linear arrangement problem in their first step choose the



(a) Normal lifeline sorting.

(b) Lifeline sorting according to areas.

Figure 4.6. *Lifeline0* and *Lifeline1* are swapped since the grouped messages have higher priority in the message length minimization process if the *group according to areas* option is enabled.

starting node according to its connections to other nodes or even randomly, an approach specific to sequence diagrams could choose the node that represents the starting lifeline. This approach of well-proven algorithms for the general problem in combination with a customization related to sequence diagrams combines the different aesthetic criteria. The edge length is minimized and the familiar positioning of the starting lifeline at the left side is kept.

4.5.4 Group According to Areas

Another optimization of the approach by McAllister is related to another optimization goal. Areas, such as combined fragments or interactions may group several messages together. In order to keep the size of these areas as small as possible, which improves readability, those lifelines that are connected to messages contained in an area should be drawn close to each other where possible.

Edge weights in the temporary graph can be modified in order to group the lifelines that are connected by messages in areas. For every message that is contained in such an area, the edge weight in the temporary graph is increased. With that, such messages will be kept shorter than others. This, in turn, tends to group the lifelines that are connected to these messages. Total edge length, however, may be increased by this approach. Figure 4.6 shows

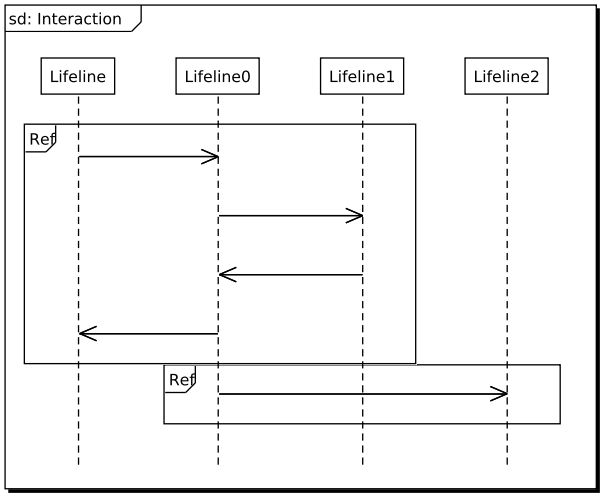


Figure 4.7. Two areas that are conflicting with respect to message grouping. It is not possible to minimize the length of the messages in both areas.

an example of this scenario; in Figure 4.6a the lifeline sorter does not consider the presence of the area, Figure 4.6b shows the result when the *group according to areas* option is activated.

The grouping of messages in different areas may be contradicting each other in matters of edge length minimization. Figure 4.7 shows an example of such a conflicting situation. *Lifeline0* is placed in the middle of an area that contains messages of three lifelines because of their internal connections. The other area's message, however, is connected to that central lifeline and a lifeline outside of the first area. The two areas then are contradicting with respect to lifeline sorting. Nevertheless, the lifeline sorting algorithm aims for the optimal solution with respect to these areas. Since message's weights are increased for the messages contained in these areas, the algorithm will prefer to shorten them but it will not get stuck into their contradicting priorities.

4.5.5 Interactive Lifeline Sorting

In addition to the lifeline sorting algorithms described so far, a very basic lifeline sorting method was implemented. Since users may want to handle lifeline sorting themselves, the interactive lifeline sorter was integrated into the sequence diagram layout algorithm. This lifeline sorter simply keeps the order that was passed from the editor. The layout algorithm, nevertheless, puts the right spacing between the lifelines and cares for the right vertical position.

The next section will give an overview over the problems of message label placement, especially with respect to very long labels that may cause overlappings in some cases. Several approaches for this problem are discussed and justified. In addition to that, the alignment of the labels is discussed since there are several approaches that have different advantages.

4.6 Label Placement

4.6.1 Theoretical Concepts

Label placement is a well-known problem in automatic graph layout. It is a problem that is generally difficult to solve since it is often regarded as the last step of the layout process, when everything else is already aligned and little space is left for the labels. Label placement thus often has to be done with little space or by modifying the already calculated coordinates from earlier steps.

Instead of reducing label placement to a mere postprocessing step, it may also be directly integrated into the layout algorithm. Carstens [Car12] already showed that this is possible by integrating label placement into the KLay Layered algorithm of the KIELER framework. Klau and Mutzel [KM99] and Binucci et al. [BDLN02] integrated label placement into the compaction phase of orthogonal layout algorithms.

In sequence diagrams, message labels do not introduce any difficult problems. Generally, providing enough space between the messages is straightforward and the visual association between labeled objects and their labels is simple since messages do not cross each other. This may be voided, however, if the spacing is chosen too big, since message labels may be confused if they are placed closer towards another message than to their message. Sequence diagrams do not contain vertical messages either, which are difficult in terms of label placement because labels are oriented horizontally in general. Only wrapped labels may need more vertical space. The most difficult problems are thus avoided in sequence diagrams.

Message labels that are very long may cause some trouble, though. If a label exceeds the distance between the two lifelines connected by its message, the label does not fit into the space between the lifelines without overlapping at least one of them. Since messages do not share the same vertical position if connected to the same lifeline, the labels of different messages do not overlap except for really long labels that span more than two times the length of the message. Slightly exceeding the length of the message would not have too big of an impact on the readability of the whole diagram. The crossing of the lifeline and the message's label, however, reduces the readability of the label itself.

There are three major approaches to that problem that will be explained and discussed in the following:

1. Enlarging the lifeline spacing in order to have enough space for the labels.
2. Changing the order of lifelines to place lifelines further apart that are connected by messages with very long labels.
3. Wrapping message labels at convenient positions to be short enough to fit between the connected lifelines.

When talking about the modification of lifeline spacing, it is not clear which spacing should be modified when messages connect non-neighbored lifelines; only one of the affected spacings or the spacing of all the affected lifelines could be modified in order to balance the

spacing as well as possible. Modifying all the spacings, however, may lead to conflicting situations when spacings are affected by different long message labels: the right spacing in terms of homogeneous distribution of lifelines would then be difficult to determine. In addition, such a distribution would have to be a new step in the layout process. Modifying only one spacing simply needs to check for excess length message labels when placing their source lifelines, which is much easier and faster. For neighbored source and target lifelines, which is the usual case, however, there is no difference between these approaches.

When modifying the lifeline sorting, excess length messages would span several lifelines, if possible. While the label would cross the lifelines that are located between the source and the target lifeline, it would not exceed the message length anymore and therefore, crossing other message's labels would be avoided. The *long message avoiding lifeline sorter* described in Section 4.4 can implement this approach with some modifications: for each message the priority is decreased if the message label exceeds the space available between the source and target lifeline. However, it is not always possible to find an order of the lifelines that respects these constraints. This is particularly true if several messages have excess length. Additionally, such an intervention into the lifeline sorting process would possibly contradict other aesthetic criteria.

Wrapping labels at convenient positions is another approach to improve label placement. The problem is difficult to solve since contradicting goals are involved. The main goal is to save horizontal space by wrapping long labels at appropriate locations. This, however, increases the vertical size of the label.

4.6.2 Implementation

Labels are placed automatically by the Papyrus framework if they were not previously placed manually by the user. The standard way of placing labels in Papyrus is at the horizontal center of the corresponding message. The vertical position relative to the message is determined by the direction of the message: labels are placed on the left side of the message with regard to the message's direction. More precisely, labels are drawn above the message if the message points from left to right and below otherwise. Centered labels may overlap a lifeline if their message spans an even number of lifelines (see Figure 4.8).

However, labels that were once moved by hand by the user are not placed properly anymore by Papyrus. Instead, their placement relative to the message is preserved, which yields strange results particularly if the order of the lifelines is changed by the algorithm. Therefore, the label placement of the Papyrus framework was overridden by this layout algorithm. Considering the thoughts of Section 4.6.1, the first of the two options was implemented: Lifeline spacing is modified in a way that the whole label of each message fits into the space between the source lifeline and the neighbored lifeline in the corresponding direction. This may result in the drawing becoming a little too wide in case of very long message labels that span more than the typical lifeline spacing; however, the readability of the labels is improved a lot by this approach.

In order to serve different preferences, three different horizontal label placement strategies

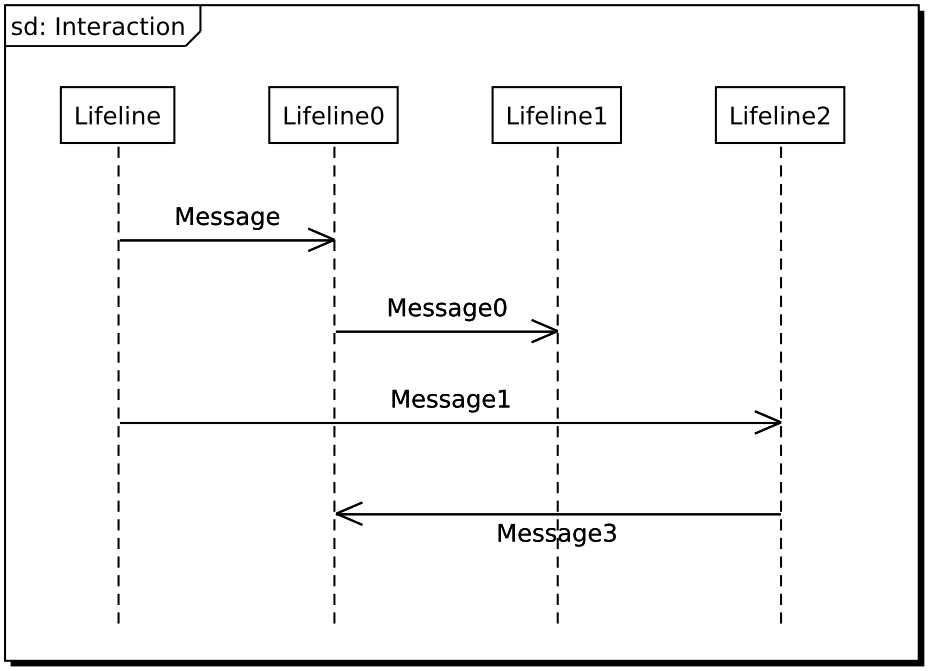


Figure 4.8. Label placement as implemented in the Papyrus framework originally. Labels are placed at the horizontal center of the message regardless of the length of the message. Labels may overlap lifelines if their message spans an even number of lifelines (see *Message3*).

were implemented and are available through the layout options:

1. Center (Figure 4.8): this is the way that Papyrus places the labels when they were not touched manually. The labels are horizontally centered in relation to the message. This may lead to overlapping labels and lifelines if a message spans more than one lifeline.

2. Source (Figure 4.9): labels are placed near to their source lifeline. This is a more intuitive way since it highlights the origin of the message, thus leading the eye through the diagram according to the control flow that is described by the diagram.

3. First center (Figure 4.10): labels are placed central in the gap between the source lifeline and its direct neighbor. With that, the drawback of overlapping labels and lifelines can be avoided while keeping the advantages of optically pleasing, centered message labels.

Vertical label placement is fixed. As originally implemented in the Papyrus framework, labels are drawn above the message if the message points from left to right and below otherwise.

As already mentioned in Section 4.6.1, wrapping labels could save horizontal space at the expense of vertical space. However, since it is not possible to apply changes of message label text to the Papyrus sequence diagram editor, no such feature was implemented in the layout algorithm.

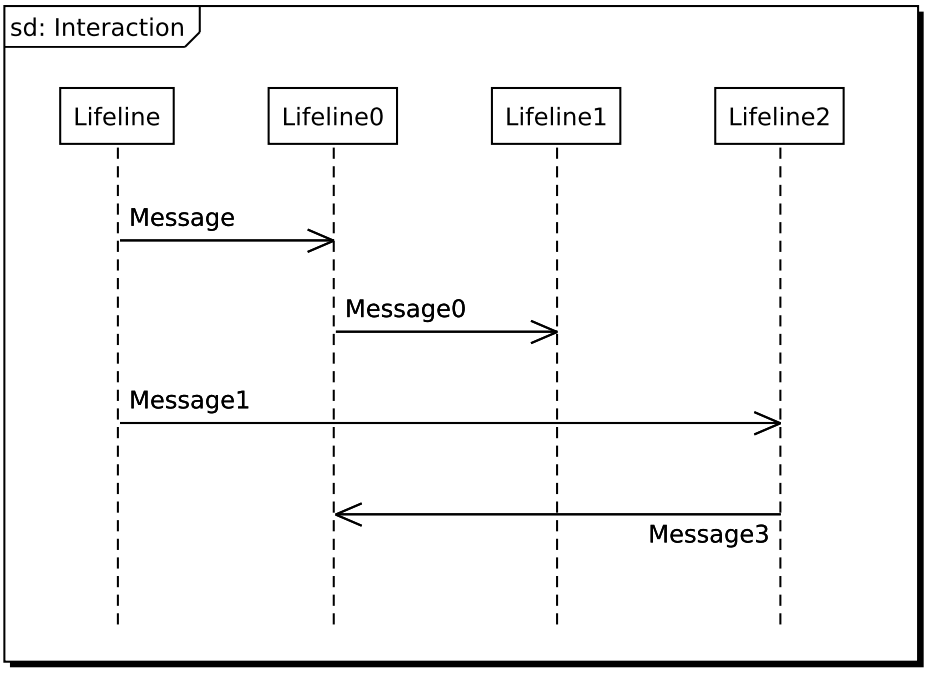


Figure 4.9. Label placement as tail labels. Labels are placed near the source lifeline. The source lifeline of a message is highlighted thereby while the target lifeline is highlighted by the arrow head.

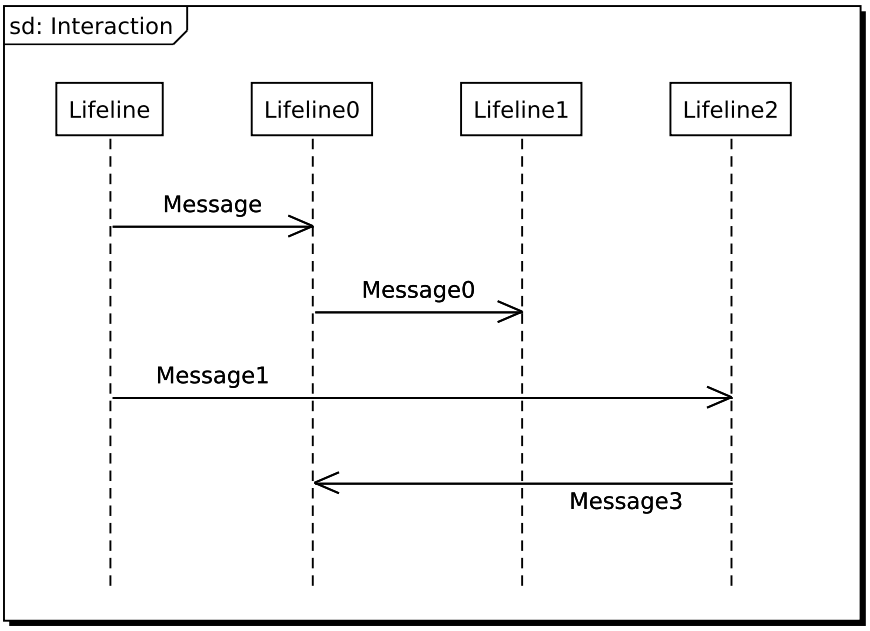


Figure 4.10. Label placement at the center of the first gap between the lifelines. The source lifeline is highlighted for long messages. However, the placement yields a more harmonic result than the tail labels since labels are centered regardless of their direction.

4.7 Putting things together

This section finally describes how the presented concepts are combined to build the sequence diagram layout algorithm developed in this thesis. An overview of the different phases and data structures is given in Figure 4.11. In the following, the phases that are passed by the layout algorithm are described consecutively.

4.7.1 Import Graph

At first, the graph has to be imported and the special, sequence diagram specific temporal graph has to be created. All the nodes and edges of the `KGraph` are transfered into lifelines, messages, areas, activations and comments.

4.7.2 Create Layered Graph

After that, another temporal graph is build for the layering of the messages. Namely, an `LGraph`, which is in use in the KIELER framework for the layered graph layout, is build according to the relative order of the messages at each lifeline.

4.7.3 Allocate Space

In the `LGraph` structure, dummy nodes are inserted a several positions. This clears the space for several objects that are not covered by the lifelines and messages relations. The headers of lifelines, which are created by a create message and therefore do not have their header at the top of the diagram, should not be crossed by messages for example. Comments, too, need some space that is not crossed by any message.

4.7.4 Break Cycles

With these preconditions, the layered graph is then scanned for cycles, which may occur in rare cases when asynchronous messages are involved (an example is given in Figure 5.6). Every cycle that is detected is broken by reversing as few messages as possible, producing a cycle-free graph.

4.7.5 Assign Layers

With a cycle-free graph, the layerer is able to find a layering of the nodes such that all relative orders of the diagram are preserved. Nevertheless, several messages may share the same vertical layer after the layering process.

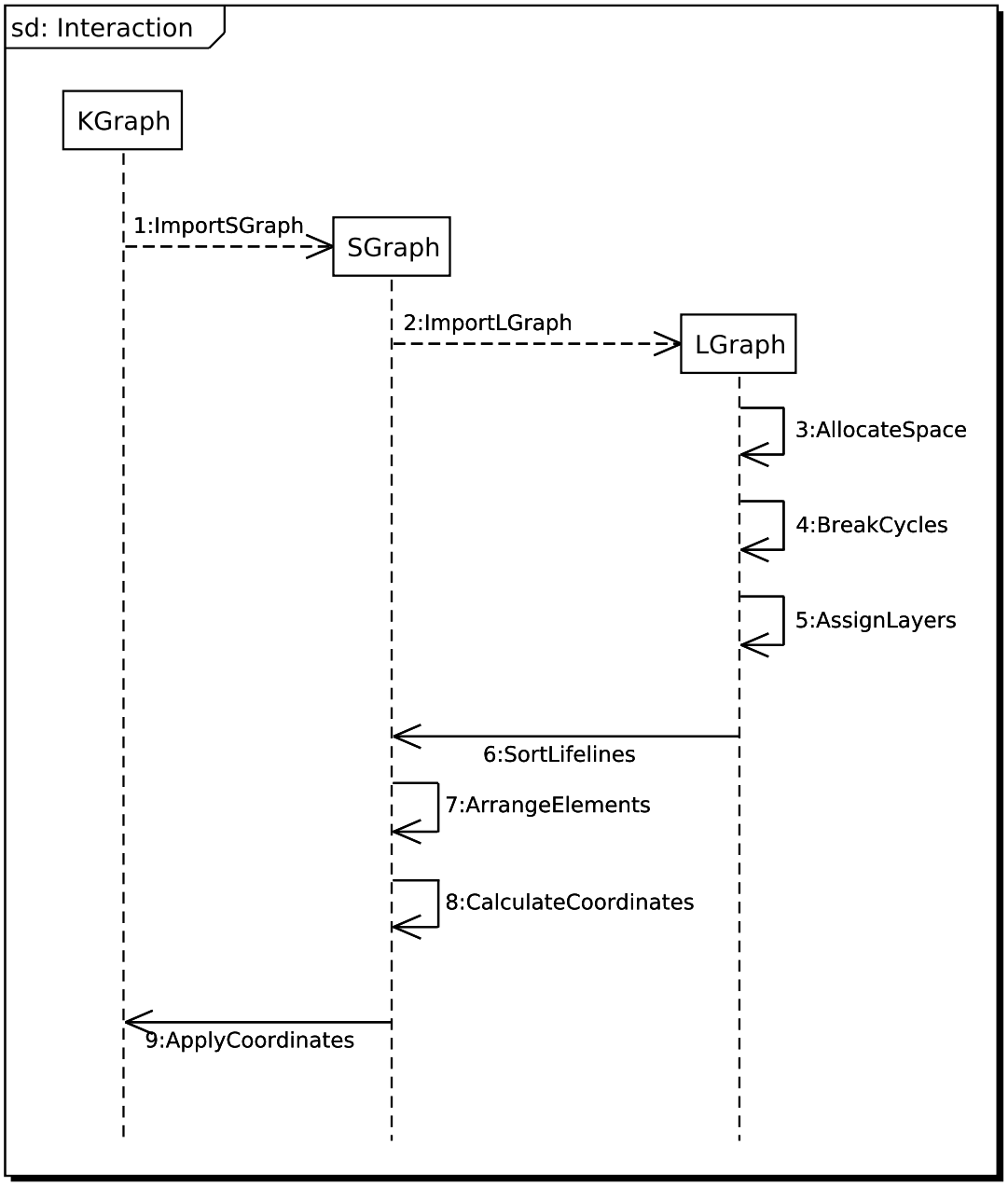


Figure 4.11. The different phases of the layout algorithm. Only the part that is independent of the concrete editor is shown. Three different graph structures are involved: the `KGraph`, the `SGraph` and the `LGraph`.

4.7.6 Sort Lifelines

The assignment of the message layers is the precondition for the lifeline sorting in general. Some of the lifeline sorting strategies, such as the layer-based lifeline sorter, rely on the results of the layering. The lifeline sorting is done with the user-chosen strategy.

4.7.7 Calculate Coordinates

With all the preliminary work done, the main part of the layout, namely the calculation of the actual coordinates for the different objects, can be started. These coordinates are calculated in several steps again, which are described in the following.

- *Calculate message y-coordinates*: from the layering of the messages, the actual vertical coordinates are calculated. In some cases, the layering has to be adapted if the lifeline sorting produced overlapping messages in the same layer.
- *Arrange connected comments*: those comments that are connected to a lifeline or another element have to be placed in the space between the messages and lifelines. They are placed relative to the nearest message.
- *Calculate lifeline positions*: lifelines are placed one after another according to the user-defined spacing and dependent on the with of comments and labels that are located between them. In addition to that, the horizontal positions of the connected messages are set here.
- *Calculate position for unconnected comments*: comments that are not connected to any object in the diagram are placed on the right side of the diagram.
- *Calculate positions for areas*: the positioning of the areas is calculated according to the position of the messages and lifelines that are contained in the area.

4.7.8 Apply Layout Coordinates

In a last step, the calculated coordinates are applied to the diagram. This is done by applying the coordinates of every object to the layout of the corresponding element in the `KGraph`. Label placement is done in this step too since it does not need any prior calculations or adjustments to the other elements.

After the explanation of the structure and the order of the different steps in the layout process in this chapter, Chapter 5 looks a little deeper. The implementation of the single algorithms is explained in detail and the data structures are introduced more precisely.

Chapter 5

Implementation

This chapter deals with the technical aspects of the algorithms described in the previous chapter. First, a look at the data structures is taken in Section 5.1 before Section 5.2 introduces the available layout options. Section 5.3 then shows how the actual layout algorithm is implemented. After that, Section 5.4 describes the integration of the algorithm into the Papyrus framework.

5.1 Data Structures

To be able to do automatic graph layout, a representation of graphs is necessary. Even though sequence diagram layout is a special case of graph layout, it needs a specialized graph representation. Several data structures are used in the algorithm to represent the different graphs and graph-like structures.

5.1.1 KGraph

The `KGraph` is the main graph representation of KIELER. Every diagram that is laid out is passed to the layout algorithm in this format. To be able to handle all kinds of graphs and graph-like structures, the `KGraph` is a very general purpose graph format. Special requirements of the different graph structures can be satisfied with the properties that can be attached to every element of the `KGraph`. A class diagram of the `KGraph` format is provided in Figure 5.1. There are four main types defined in the `KGraph`:

- `KNodes` are used for every element that has a shape except for labels, which are covered below.
- `KPorts` can be attached to `KNodes`. They are used to represent dedicated connection points for the connected edges.
- `KEdges` are used for everything that connects different elements of a graph. They connect different `KPorts` or `KNodes` and may have bend points.
- `KLabels` may be attached to every one of the above elements. Several types of labels are supported (head labels, tail labels, center labels). Several labels may be attached to a single element.

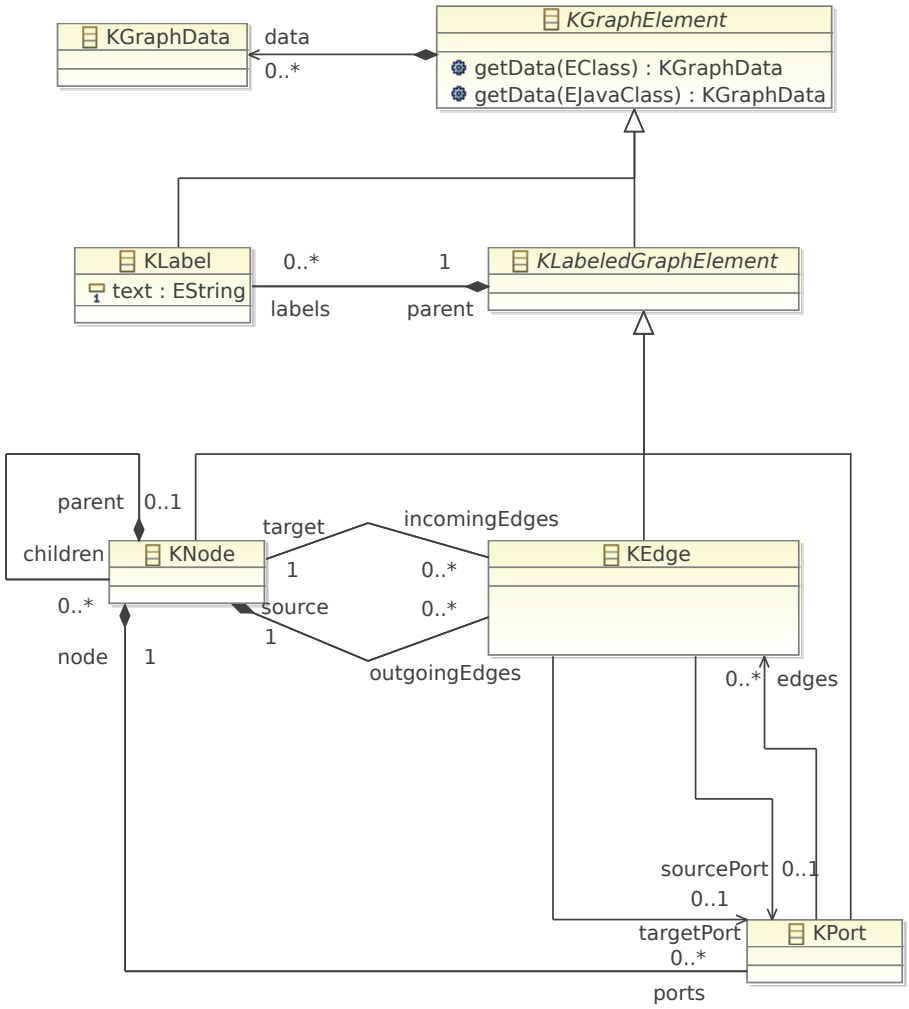


Figure 5.1. A class diagram of the `KGraph` structure. The `KGraph` can be divided into the actual graph shown here and the graph data that stores properties of the graph elements, shown in Figure 5.2

Each of these elements implements the interface `KGraphElement` which enables the elements to store different kinds of properties. Each of the properties has a specific data. A class diagram of the classes involved is shown in Figure 5.2.

`KNodes` may be nested in the `KGraph` format, enabling hierarchical graph structures (different graph-like structures that are mapped to the `KGraph` using nested `KNodes` for various reasons). The parent `KNode` can be accessed from each node as well as the list of its children.

A list of incoming `KEdges` is maintained for each `KNode` as well as a list of outgoing `KEdges`. Another way of accessing the connections of a `KNode` is through the list of attached ports. Each of the ports stores the owning node and has a list of the connected edges. The `KEdges`, in turn, store their source and target nodes as well as their source and target ports, if applicable.

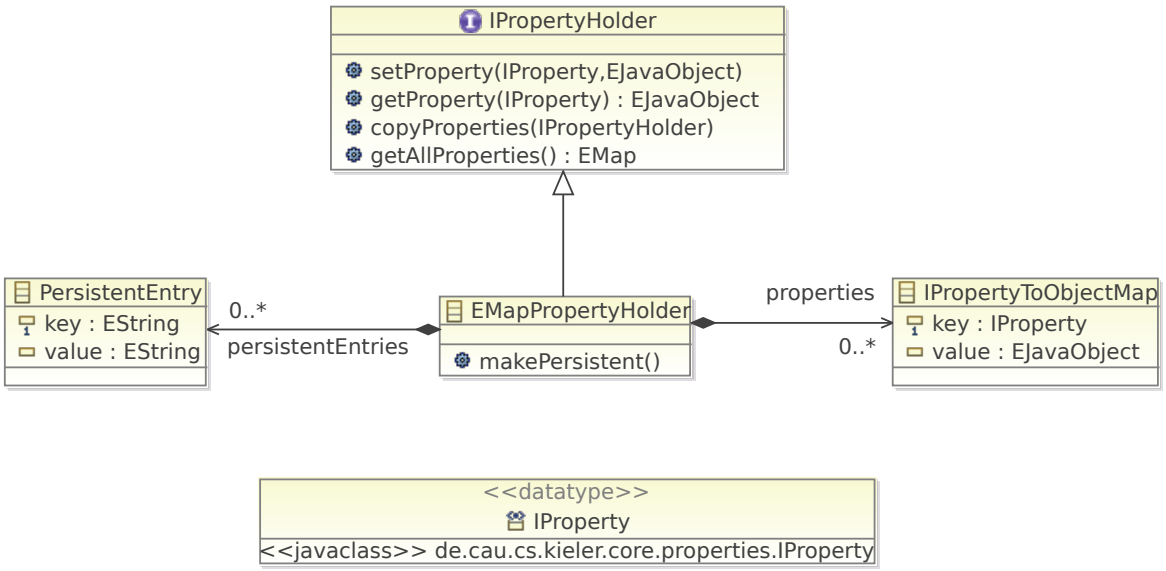


Figure 5.2. A class diagram of the *properties* structure. Several properties can be attached to every instance of a class that implements the `IPropertyHolder` interface.

5.1.2 SGraph

The `KGraph` structure is a very general purpose graph structure. However, it is not very suitable for sequence diagrams since they cannot be easily mapped to graphs as explained in Section 2.2. Lifelines can be mapped to nodes and messages can be mapped to edges, but areas such as interactions cannot be mapped easily since they may partly overlap lifelines. Activations may be mapped to sub nodes but this is not very elegant either.

To be able to work with sequence diagrams, a new format has to be introduced. The `SGraph` structure (see Figure 5.3) is build to meet the requirements of sequence diagrams. It is the sequence diagram representation that is used throughout the layout process. The `SGraph` consists of three kinds of elements: `SLifelines` represent lifelines, `SMessages` represent messages, and `SComments` are used for comments and constraints. Each of these subclasses the `SGraphElement`, which enables the elements to store the same kind of properties as the ones available for `KGraphElements`.

The `SGraph` stores a list of the contained `SLifelines` as well as a list of the `SComments` that are part of the diagram. Each of the `SLifelines` maintains a list of connected `SMessages` and a list of the `SComments` that are connected to that lifeline. The list of messages can be filtered for incoming or outgoing messages. `SMessages` store their source and target lifelines as well as a list of comments that are attached to that message.

All the other elements of a sequence diagram are handled through the use of properties attached to the various elements. There are data structures for activations / execution specifications (`SequenceExecution`) and interactions / combined fragments (`SequenceArea`) too,

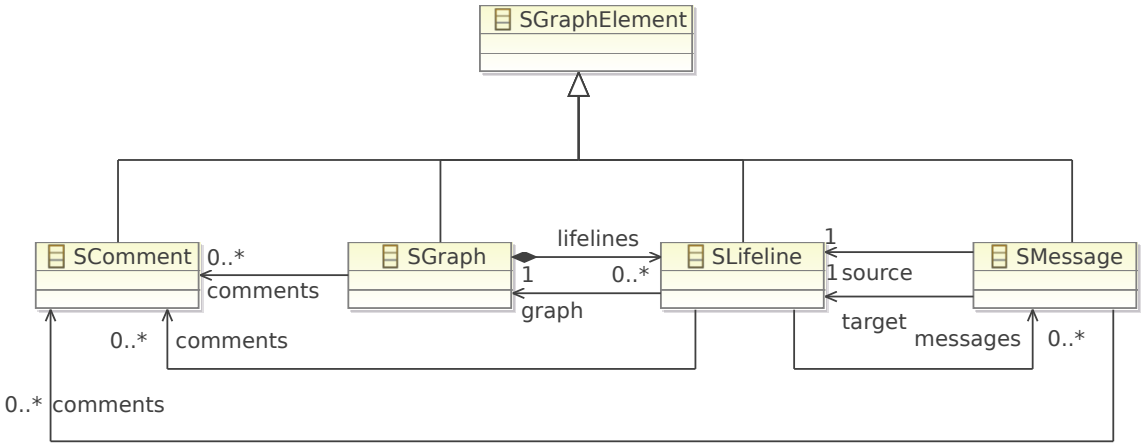


Figure 5.3. A class diagram of the `SGraph` structure. The `SGraph` mainly consists of *lifelines*, *messages* and *comments*.

but they are not part of the `SGraph` structure. This part of the design was chosen because of the plug-in structure of the related projects which is explained in Section 5.4. `SequenceExecution`s and `SequenceAreas` are already created when the `KGraph` is extracted from the diagram information.

5.1.3 LGraph

The `LGraph` is a data structure from the KLay Layered algorithm of the KIELER framework that is dedicated to layered graphs. Although a sequence diagram is not a layered graph, the concept of layers is applied to the messages in this thesis in order to find the best vertical positions for the messages. The `LGraph` is composed of the following elements (see Figure 5.4):

- `Layers` are collections of nodes that should be drawn in the same horizontal or vertical position.
- `LNodes` represent nodes in the `LGraph`.
- `LPorts` represent ports of a `LNode` in the `LGraph`.
- `LLabels` represent labels that can be attached to nodes, ports, and edges.
- `LEdges` represent the edges in the `LGraph`.

All of these elements inherit the possibility to store properties from the abstract class `LGraphElement`. Since `Layers` are collections of nodes, they just store the layer position and a list of `LNodes` contained in it. If an `LNode` is not contained in any layer it is stored in the `LGraph`'s list of `layerlessNodes`. If it is assigned to a layer, it stores the owning layer in its field `owner`. In difference to the `KGraph`, the `LNode` does not store any data regarding the connected edges. `LEdges`, `LPorts`, and `LNodes` store a possibly empty list of attached `LLabels`.

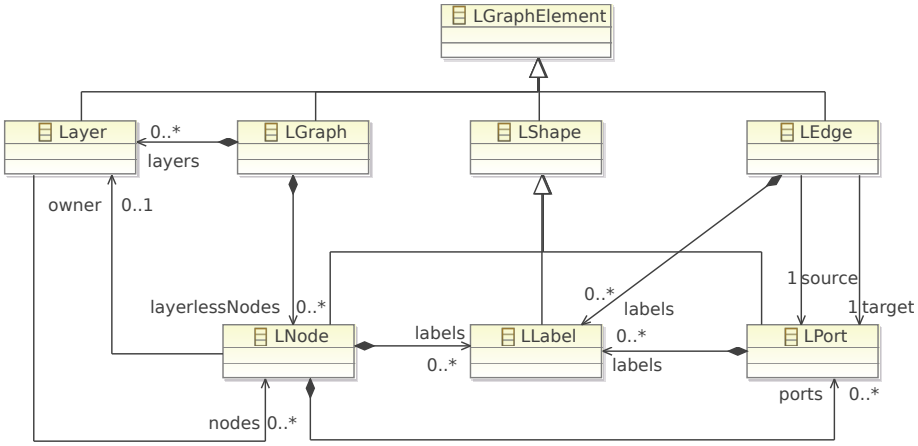


Figure 5.4. A class diagram of the `LGraph` structure.

5.2 User Defined Layout

With KIML, the KIELER framework provides an infrastructure that provides several services for diagram layout. One of its features is the layout view. It provides different layout options specific to the diagram or element that is currently displayed or selected. These options can be numeric values, booleans, or an enumeration (string values are valid too, but not in use so far). The options that are chosen by the user for a specific element of the diagram are passed to the corresponding `KNode` when the layout process is triggered. The layout algorithm then calculates a layout according to these options. The following layout options are available for sequence diagrams:

- *Border spacing* is a numeric option that is used to resize the margin around the diagram.
- *Lifeline spacing* is a numeric option that determines the horizontal spacing between two neighboring lifelines.
- *Message spacing* is a numeric option that determines the minimal vertical spacing between two neighboring messages.
- *Lifeline sorting strategy* is an enumeration of possible strategies for the sorting of the lifelines (see Section 4.5).
  - *Group according to areas* is a boolean option that is only applicable when the *short message lifeline sorter* is chosen as the lifeline sorting strategy (see Section 4.5.4). It determines if messages contained in areas should be given a higher priority for the minimization of the message length.
- *Label alignment* is an enumeration of different label alignment strategies (see Section 4.6).

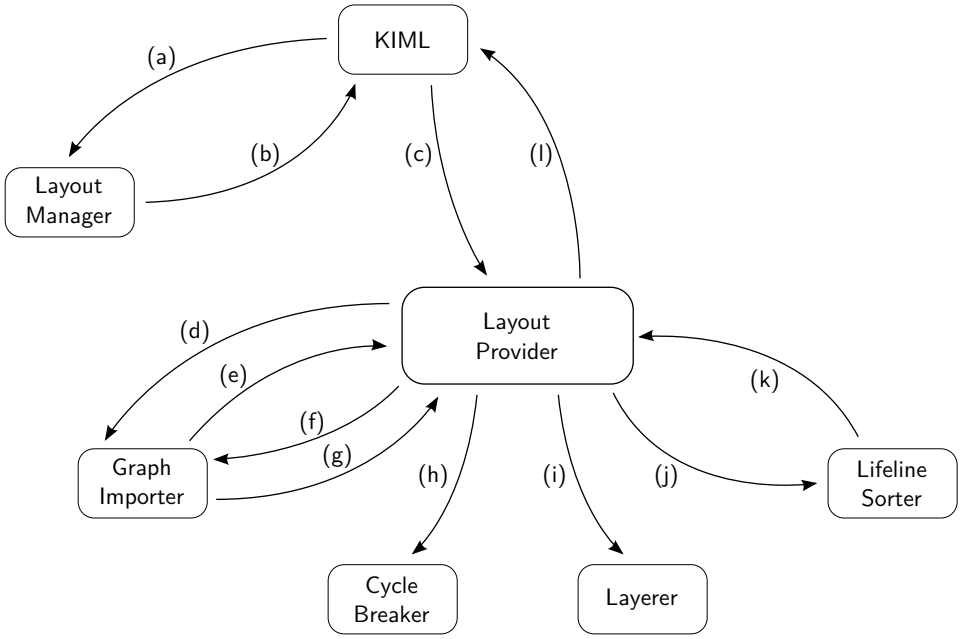


Figure 5.5. Overview of the main components that are involved in the layout process.

Layout options are usually provided for every shaped element in the diagram. This is very useful for graph like diagrams in order to have different styles of layout for different hierarchical nodes. For sequence diagrams, however, this is not very applicable since they do not contain hierarchical structures. Additionally, different layout options for different elements in the diagram may be confusing for the user. In this algorithm, the layout options are disabled for the surrounding `PackageEditPart` that is not part of the actual sequence diagram.

5.3 The Algorithm

The sequence diagram layout algorithm is divided into several phases. These phases were already introduced in Section 4.7. Figure 5.5 shows an overview of the invocation of the different parts. KIML requests the layout manager to convert the diagram to be laid out into KIELER's general purpose graph format `KGraph` (a). The layout manager then processes all the elements and returns a `KGraph` representation (b). In the next step, KIML passes this `KGraph` to the layout provider that calculates a layout (c).

The layout provider splits its work into several steps that are done by different classes. First, the graph importer is activated to transform the `KGraph` into the `SGraph` (d) which is returned to the layout provider (e). Another temporary graph format, the `LGraph`, is created for the layering of the messages (f) and is returned to the layout provider as well (g). After allocating space for various objects internally, the layout provider invokes the cycle breaker to break the cycles in the `LGraph` (h). The `NetworkSimplexLayerer` of the KLay Layered algorithm

then calculates a layering for the nodes in the `LGraph` (i). Finally, the layout provider chooses a lifeline sorting strategy and invokes the corresponding lifeline sorter (j) that returns the horizontal order of the lifelines (k). Before returning to KIML (l), the layout provider internally calculates the final coordinates for all the elements of the diagram and stores them in the `KGraph`.

This section describes the actual implementation of the layout algorithm and therefore does not cover the interaction between KIML and the layout manager (see Section 5.4 for a description of that). Instead, this section focuses on the interaction between KIML, the Layout Provider, and its utility classes. An overview over the steps of the `SequenceDiagramLayoutProvider`, which is the main class of the layout algorithm, is given in the following:

- Import `SGraph` (Section 5.3.1)
  - Create lifelines
  - Create messages and comments
  - Handle empty areas
- Create `LGraph` (Section 5.3.2)
- Add dummy nodes to allocate space (Section 5.3.3)
  - Add dummy nodes for combined fragments
  - Add dummy nodes for comments
  - Add dummy nodes for empty areas
- Break cycles in `LGraph` (Section 5.3.4)
  - Depth first search
  - Split nodes
- Calculate layering of `LGraph` (Section 5.3.5)
- Sort lifelines (Section 5.3.6)
  - Interactive lifeline sorting
  - Layer-based lifeline sorting
  - Equal distribution lifeline sorting
- Calculate coordinates (Section 5.3.7)
  - Calculate message coordinates
  - Calculate other coordinates
- Apply layout (Section 5.3.8)

Each of these steps is described in detail in the remainder of this section.

5.3.1 Import Sequence Diagram Structure

The sequence diagram is passed to the algorithm in KIELER's KGraph format. This format is a general purpose graph format that is not specialized on sequence diagrams at all. Therefore, the data has to be extracted from that KGraph into a special sequence diagram format to be able to work with it in a proper way. The SGraph proposed in Section 5.1.2 matches the requirements of the layout algorithm since it is built for that purpose.

The SGraphImporter builds a new SGraph from a given sequence diagram in the KGraph format. It maintains two hash maps for mapping KNodes to SLifelines as well as for mapping KEdges to SMessages. In addition to that, it stores a list of the areas (such as *interactions* and *combined fragments*) that belong to the sequence diagram.

First, the importer walks through the top-level elements and creates a SLifeline object for each of them which is a lifeline. For each of the created lifelines, the original name and the layout information is copied. The originating KNode is stored too and the node-lifeline mapping is stored in the corresponding hash map. Additionally, the SLifeline stores a list of related executions and whether it has an attached destruction event.

In a next step, the top-level elements are traversed again. This time, SMessages are created for every outgoing KEdge and for every KEdge that comes from something else than a lifeline (*found messages*, messages from the surrounding interaction). If an edge came from something else than a lifeline, it would not be reached through the other lifelines and, therefore, no SMessage would be created for that KEdge. If the current KNode is no lifeline, it is assumed to be a comment, constraint or a time observation. In this case, an SComment object is created and initialized.

After iterating over the lifelines and messages, the algorithm checks for empty areas. Empty areas are difficult to handle since they do not contain any information about their neighbored lifelines and messages. Therefore, the neighboring elements have to be detected in advance by comparing their positions with the position of the empty area. More precisely, the handleEmptyArea method searches for the message just below the area and stores this message in the nextMessage field of the area. With that, the space required for the empty area can be allocated as described in Section 5.3.3. In the following, the creation of SMessages and SComments is explained in detail.

The createMessages and createIncomingMessages methods only differ very little since they basically do the same thing - they create SMessage objects from KEdges. Similar to that, the createIncomingMessages method scans incoming instead of outgoing messages and skips all the messages that have a lifeline as source. This has to be done to create SMessage objects for those messages that do not have a source lifeline (messages from the surrounding interaction or *found* messages).

For each message, the SLifelines corresponding to the connected KNodes are looked up in the nodeMap. If the source or the target is not a lifeline, a dummy lifeline is created before the SMessage object is created. The original layout information and the originating KEdge is stored before the length of the widest label is determined and stored, too.

Since the SequenceExecution objects are already built in the previous step, they contain

references to the KEdge objects instead of the SMessage objects. This is corrected by replacing each of them with the corresponding SMessage.

In order not to work with the String-valued message types that are provided by the Papyrus framework, the MESSAGE_TYPE property is used within the algorithm. It is an enumeration of seven different message types: ASYNCHRONOUS, SYNCHRONOUS, REPLY, CREATE, DELETE, LOST, and FOUND. These properties are set on the messages according to the String values provided by Papyrus.

Next, the message's position is compared to the positions of all areas. If the message fits completely into the bounding box of an area, it is added to the area's list of contained messages. With that, all the messages that are contained in an area can be directly accessed through the area.

We now turn to the creation of the SComment objects. Comments, constraints, and time observations are handled equally in general. Therefore, we will refer to all of them as comments in the following. Their placement is determined by their connections. This, however, is not as easy as it seems. Comments may be attached to lifelines or execution specifications as well as to messages. It is generally desirable to place comments near to the object that they are attached to.

The createComment method first creates the SComment object and copies all the stored properties of the KNode to it. The originating KNode is stored along with its type (comment, constraint, or time observation) and the element (lifeline, message, or execution specification) that the comment is attached to. A reference to the connecting KEdge is stored as well, in order to be able to modify its position later. In addition to the ATTACHED_ELEMENT property, a list of elements called ATTACHED_TO is kept. The list stores those elements in the diagram that are closest to the comment. This information is necessary in order to find the best position in the diagram to place the comment at. The layout information of the KNode is copied to the comment after that.

The last step would be a special handling for time observations since they are not connected to messages or lifelines, but to the *send* or *receive event* of a message. They have to be placed near to that event. This, however, is omitted here because of a limitation in the Papyrus framework: there is no way to get the connections of the time observations programmatically. Thus, no placement can be computed.

5.3.2 Layering Messages

In order to divide the messages into several vertical layers, an LGraph is created. This *layered graph* is the graph format that is used with KIELER's KLay Layered layout algorithm. The concept of this was already shown in Figure 4.3: for every message in the diagram, an LNode is created and LEdges connect the LNodes at each neighbored pair of messages at every lifeline, in order to preserve the relative order of the messages at the lifeline.

At first, an LNode is created for every message in the diagram by iterating over the outgoing messages of every lifeline. The incoming messages have to be scanned as well, to create LNodes for all *found* messages. For the mapping of LNodes to their corresponding SMessages,

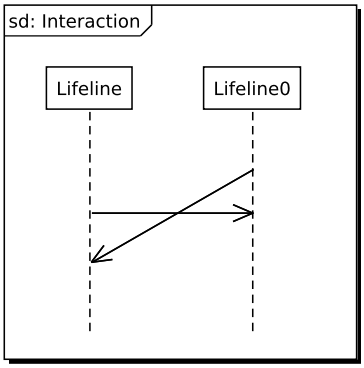


Figure 5.6. Cyclic dependencies of two asynchronous messages that introduce cycles in the `LGraph`.

the `origin` property of the `LNode` stores a reference to the `SMessage`, and the message has a property `LAYERED_NODE` that references the node. Finally, the `LNode` has to be added to the list of layerless nodes which are the nodes that have not been assigned a layer yet.

Thereafter, the `LNodes` that have just been created are connected by `LEdges`. To do so, every pair of vertically neighbored messages at every lifeline is found and an `LEdge` is created that points from the upper message's `LNode` the the lower message's `LNode`. Since the `LGraph` is a graph format that uses ports, source and target `LPorts` have to be created for each of the `LEdges`.

The `LEdge` stores a property `BELONGS_TO_LIFELINE` that stores a reference to the `SLifeline` object that the `LEdge` belongs to. This property becomes necessary when there are cycles in the `LGraph`. These cycles occur when a set of messages have cyclic dependencies as illustrated in Figure 5.6. This can only be handled by drawing at least one message at an angle. In this case, the corresponding `LNode` has to be split into two `LNodes` that will be placed in different layers.

5.3.3 Allocating Space for Various Objects

Apart from lifelines and messages, some other objects in sequence diagrams need additional space. Comments and constraints may be placed between the lifelines and messages. The headers of combined fragments should not overlap messages either. In addition to that, empty areas may occur in the diagram. These areas do not have any contained message, which prevents them from being considered in the earlier steps.

As the overlapping of any objects is undesirable, space for these objects has to be allocated. This is done in an early step by inserting dummy nodes into the layered graph. These dummy nodes reserve a certain amount of space between two vertically neighbored messages, enabling the mentioned objects to fit into that space.

Depending on the parameter `beforeNode`, the dummy node is inserted before or after the given `LNode`. In case of placing the dummy before the given `LNode`, all the incoming `LEdges` are stored and their target is reset to the dummy node's `LPort` for incoming edges. The

additional `LEdge` is then set to connect the dummy node with the given `LNode`. If the dummy is placed after the given node, the procedure is executed with outgoing edges and source ports, respectively. Finally, the dummy node is added to the `LGraph`.

The first object that is considered for the allocation of additional space is the *combined fragment*. Its header does not contain any messages and therefore the additional space has to be allocated here. Combined fragments contain different regions that are represented by `subAreas`. Therefore, the list of areas is scanned for those that have `subAreas` to find all the combined fragments. The uppermost contained message of each of them is determined by simply comparing their vertical positions. A dummy node is then inserted above the node that represents this message.

A comment may be attached to different kinds of elements in the diagram. During the import of the `SGraph` every comment object is assigned an `SMessage` that is the nearest to the comment's position. In order to place the comment above this message, dummy nodes are inserted before the corresponding `LNode` in the layered graph. First, the list of attached elements is scanned for an `SMessage`. Next, the number of dummies is determined from the height of the comment, taking into account the message spacing. The dummies are then created one after another. At last, the comment stores the nearest `SMessage` in a separate field and the message adds the comment object to its list of connected comments.

Similar to comments, empty areas need some space. Since empty areas do not contain any message that would allow the area to be placed around it, this space must be allocated. The list of areas is filtered for those areas that do not contain any messages. Each of them stores a `nextMessage`, which holds a reference to the message that is closest below the area. Two dummy nodes are then created before (above) the corresponding `LNode` in the layered graph.

5.3.4 Breaking Cycles

If there are cycles in the layered graph, which occurs when asynchronous messages have cyclic dependencies in the diagram (see Figure 5.6), these cycles have to be broken in order to find a layering of the nodes. Cycles are usually broken by inverting at least one edge in the graph. If there is more than one cycle, a minimal set of edges to be inverted is computed which makes the graph acyclic again.

In the special case of this algorithm, the procedure was adapted a little. Instead of inverting edges, a minimal set of nodes is split into two pieces each, one for each of the lifelines that the corresponding message connects. With that, the message may run diagonally and span more than one layer, hence breaking the cyclic dependencies.

A depth-first-search (dfs) is performed in order to find the nodes that have to be split. Within the dfs, nodes can be in one of three states:

- 0 if the node was not visited yet,
- 1 if the node was visited before, but not in the current path, and
- 2 if the node was visited in the current path.

The `node.id` field is used to store the current state of the node. First, all states are set to zero. The dfs is then performed for every node that was not visited yet. This is necessary to handle graphs with different connected components. Within the dfs, some nodes may be added to the list of nodes that have to be split which is done in the last step.

The dfs recursively searches for nodes that have already been visited in the current path. If a cycle is found, the algorithm searches for the one node in the cycle with the uppermost position in the diagram. This node is then added to the set of nodes that have to be split in the last step. If the dfs does not find a node that was already visited, it just marks the current node visited and recursively searches its successor nodes before resetting the `node.id` value.

The `addUppermostNode` method searches for the node in the current path with the uppermost position in the diagram. The one with the uppermost position is added to the set of nodes that have to be split in the final step.

Finally, every node to be split is split into two nodes, one for each of the connected lifelines. The new `LNode` is created and added to the list of nodes of the layered graph. The source and the target lifeline of the corresponding `SMessage` are looked up before the list of connected edges is traversed. Each of those that belong to the target lifeline of the corresponding `SMessage` is reassigned the new node as source or target. Afterwards, the `BELONGS_TO_LIFELINE` property is set for both of the nodes, which marks them as split nodes for subsequent steps of the algorithm.

5.3.5 Layering the Messages

With the cycles broken, the layering can be calculated. There are several common implementations for node layering available. This algorithm uses one of the KLay Layered algorithms to find a layering of the messages. More precisely, the *Network Simplex Layerer* is chosen. With that, each non-split message is assigned a vertical layer. Each part of the split nodes is assigned a different layer since the nodes are handled independently.

5.3.6 Sorting the Lifelines

Rearranging the horizontal order of the lifelines is the only step in sequence diagram layout where the layout algorithm has a real degree of freedom to do optimizations. Therefore, this thesis provides different algorithms for this step that are described in the following.

Interactive lifeline sorting. The *interactive lifeline sorting* strategy simply keeps the order of the lifelines as they were before the layout process. The lifelines are sorted by their horizontal position.

Layer-based lifeline sorting. As a brand new approach to lifeline sorting, *layer-based lifeline sorting* sorts the lifelines according to the layer of their outgoing messages. The theoretical concepts of this approach are described in Section 4.5.1. The basic algorithm, too, is already introduced in Listing 4.1.

Long message avoiding lifeline sorting. The *long message avoiding lifeline sorting* strategy that is described in Section 4.5.2 follows an approach of McAllister for the *Linear Arrangement Problem* that is equivalent to the lifeline sorting problem with respect to message-lifeline crossings [McA99]. The approach is divided into two phases. First, the leftmost lifeline is calculated before each of the following lifelines is calculated iteratively. As an improvement over the original approach, this algorithm has an optional feature that respects the presence of areas in the diagram. Those messages that are contained in an area are given a higher priority to be drawn as short as possible.

The algorithm uses yet another temporary graph format that is very lightweight and specialized for the purpose of the algorithm. The graph simply consists of a list of `EDLSNodes`. These nodes are named according to the original name of the approach, *equal distribution lifeline sorting*. For each lifeline, one such `EDLSNode` is created. Each of these nodes stores a hash map of the connected nodes and the weight of the connecting edge. In addition to that, it stores the *tl* value as described in Section 4.5.2. It holds the weighted sum of edges connected to nodes that were already placed and is used to compute the next node to be placed. The method `incrementNeighborsTL()` updates the *tl* value of every neighbored node; it is called whenever a node was chosen to be the next one to be placed.

The `EDLSNodes` are created and initialized first. A `HashBiMap` is initialized to be able to map these nodes to their corresponding lifelines. For each of the messages that connect two lifelines, the weight of the corresponding edge is increased by one. If the optional feature of *area-based lifeline sorting* is chosen, the weight of each message that is contained in an area is doubled. Messages that connect a lifeline with the surrounding interaction are considered with an increasing or decreasing *tl* value depending on the direction of the message.

The actual algorithm starts by calculating the first node to be placed. Two strategies were implemented for that purpose: The degree-based strategy as proposed originally by McAllister and a layer-based strategy that places the lifeline with the uppermost outgoing message first. Both strategies are part of the algorithm; however, only the latter is used since it is more specific to sequence diagrams.

Next, the iterative search for the next nodes to be placed is started. This step chooses the node with the smallest selection factor which is calculated as proposed in Section 4.5.2. If the selection factor is equal for two nodes, the one with the smaller *tl* value is chosen. As the last step, the order of the nodes is traversed and the corresponding lifelines are returned in that order.

5.3.7 Calculating Coordinates

Message coordinates are calculated according to the messages' layers. For each layer of `LNodes`, the corresponding messages should be drawn at the same vertical coordinates since their vertical order is not prescribed. However, it is not always possible to draw these messages at the same vertical position. As shown in Figure 5.7, a horizontal overlapping of two messages in the same layer yields a conflict. Therefore, every pair of messages in the same layer has to be checked for horizontal overlapping before fixing their vertical coordinates.

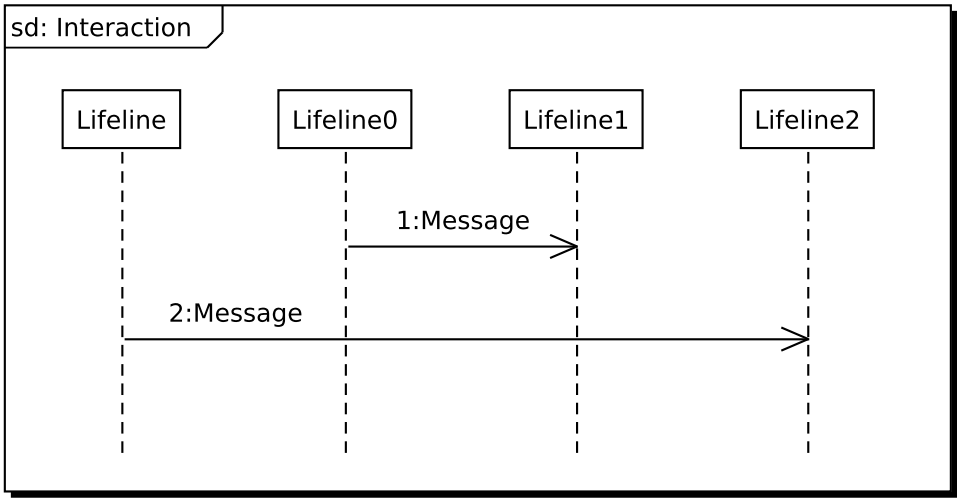


Figure 5.7. Conflicting messages: Both of the messages are placed in the same layer at first, since no relative order can be determined. In spite of this, the messages cannot be placed at the same vertical position with the given order of the lifelines.

Overlapping can only occur if at least one of the messages spans several lifelines - more precisely, if the position of the source lifeline and the target lifeline of a message differs by more than one in the list of lifelines. If both messages connect neighbored lifelines, they could only share the same horizontal space if they connected the same lifelines. This, in turn, already prevents the messages from being in the same layer. Therefore, only messages that span several lifelines are checked for overlapping horizontal sections.

If such an overlapping is detected, which is rare for sequence diagrams, one of the messages has to be moved into another layer. Since moving the message to the next layer may introduce new conflicts, a new layer is inserted after the current layer. Messages store if their vertical position is already fixed. If the conflicting message's position was already set, the current message has to be moved to the next layer. If not, and the other message does not span several lifelines, it is necessary to move the current message too since the other message is not checked for overlappings. The other message then has to be placed in the current layer. If the other message spans several lifelines itself, nothing is done in the current iteration since the other message will detect the overlapping in its own iteration (and the current message will already be fixed to the current layer). With that, as many messages as possible are placed in the upper one of the two layers.

After calculating the vertical position of the messages, the other elements can be positioned. At first, comment objects are arranged according to their connection to a lifeline or message. For each of the comment objects in the diagram, the algorithm checks if it is connected to a lifeline or to a message. As described in Section 5.3.1, every comment object that is connected to a lifeline stores the next message too. The comment is placed above that message. To find the best horizontal positioning, the comment's attached lifeline (if existing) is compared to the source and the target of the message. The comment will be placed right of this lifeline

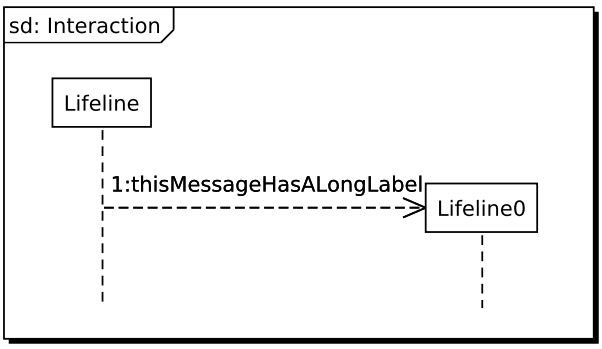


Figure 5.8. A long label at a *create message* may overlap the lifeline's header if the lifeline was not shifted to the right.

if possible. If the lifeline already is the rightmost lifeline in the range of the message, the comment will be drawn left of that lifeline. The actual coordinates are not set in this step since the lifelines' coordinates are not fixed yet, which is done in the next step.

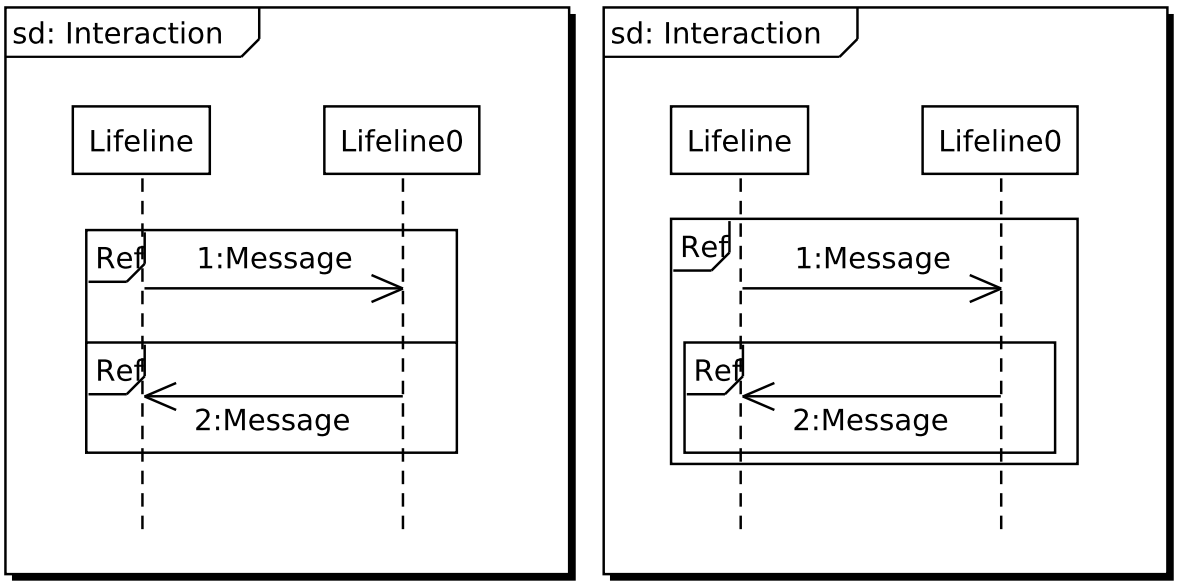
To calculate the lifelines' coordinates, the relative positions of the comments have to be known, since wide comments may increase the spacing between the neighboring lifelines. Message labels may also increase the spacing when they are wider than the normal lifeline spacing. At the beginning, the predefined lifeline spacing is assumed to be sufficient, before checking for reasons to increase that spacing. The length of the message labels is checked first. If a label does not fit into the space between the lifelines, the spacing is increased such that the long label fits, along with a small margin. *Create messages* need to have a special handling since they do not point at the lifeline but at its header, which decreases the available space for the message label (see Figure 5.8).

Next, all the comments attached to the current lifeline are traversed. If one of them is wider than the spacing calculated so far, the spacing is increased to the width of that comment. Comments are not supposed to enter the space below the lifeline header for aesthetic reasons.

After checking the comments' width, their exact position is determined. The horizontal position is determined by the calculated lifeline spacing. Comments are centered between the neighbored lifelines. The vertical placement may be a little more complicated. In general, comments should be drawn above the message with the normal message spacing between them. However, several comments may be attached to the same message. In that case, the comments have to be drawn above each other. Since observations are closely related to a certain message, they should be drawn nearer to the message if in doubt. The other comment is drawn above that with the normal message spacing.

Finally, the current lifeline's horizontal position is set to the position that is implied by the spacing calculated in the last iteration. The vertical position is set to the uppermost position in the diagram for all the lifelines. Different positions for lifelines that should be moved because of an incoming create message are reset while applying the layout to the `KGraph` (see Section 5.3.8).

After traversing all the lifelines of the diagram, its current width is calculated. Since



(a) Ambiguous drawing of two hierarchical areas.

(b) The ambiguity is resolved by a small margin that is attached to the outer area.

Figure 5.9. Hierarchical areas: an offset is attached for the outer area(s).

unconnected comments can not be attached to any object, they have to be placed at a different location. The place right of the last lifeline was chosen in this algorithm. The unconnected comments are placed one below the other in this spot. To do this, the list of comments is scanned for those comments that are not connected to any object. They are placed one below the other. The same vertical spacing as for messages is applied to these comments.

In order to calculate the position of the different areas, the hierarchy of these areas has to be determined first. If areas have any kind of hierarchy, they cannot be drawn with their normal spacing in some cases (see Figure 5.9a). Instead, the outer one of the hierarchical areas has to get a small margin (see Figure 5.9b) that resolves the ambiguity that may otherwise occur. For each of the areas placed, the depth of the hierarchy is checked first. A small margin is added for every additional level of hierarchy contained in the current area.

In a next step, the final coordinates of the area are calculated according to the contained messages. The normal margin around the contained messages is set to half of the lifeline spacing in horizontal matters and half of the message spacing in vertical matters. A potential extra margin as described above is added to that margin.

As a last step, the peculiarities of combined fragments are handled by increasing the size and placing the interaction operands. Combined fragments have a header that does not contain any messages. The space required for that header was already allocated in an earlier step as described in Section 5.3.3. The vertical position and size, thus, have to be adapted accordingly. Since combined fragments are divided into several interaction operands, each of

them has to be placed separately. They are placed one below the other within the surrounding combined fragment.

5.3.8 Applying Layout Results Back to the Diagram

The last step of the layout process is to apply the calculated coordinates back to the diagram elements. This is done by KIML once the layout provider has finished its work. KIML applies the coordinates that it finds in the `KGraph`; therefore, all the coordinates in the `SGraph` have to be transferred to the `KGraph` by the sequence diagram layout algorithm.

The elements in the `SGraph` are traversed again by iterating over the list of lifelines. For each of them, the connected messages' coordinates are copied first. The lifeline's position and height is modified in this step whenever a *create* or *delete* message points at the lifeline. In addition to that, the vertical position and size of execution specifications attached to the lifeline is set while iterating over the connected messages.

Message coordinates are given relative to the element that the message is connected to. However, they are stored in a strange way by the Papyrus framework: in order to get the desired results, the vertical coordinates must be scaled by a non-static number greater than one. It took a long time to figure out that the number depends on the size of the connected element and its parent element.

The position has to be scaled as if the element had the height of its parent. Namely, to place the source or target of a message at the lower end of a lifeline, the height has to be set to the height of the diagram instead of the height of the lifeline. Similarly, if a message should be attached to the vertical center of an execution specification, half of the height of the corresponding lifeline has to be used as the vertical position of the message.

In order to work around that behavior, a specific factor is calculated for each lifeline. Let y be the vertical coordinate relative to the connected element. Figure 5.10 shows the scales of different execution specifications and the lifelines. The source point of message (ii), for example, is calculated as follows: First, the pixel-based coordinate y relative to the execution specification is normalized to the range $[0, 1]$ by dividing by the height of the execution specification d.

$$y_{norm} = \frac{y}{d}$$

Then that value is scaled to match the height of its parent element (*Lifeline0*) a.

$$y_{parent} = \frac{y}{d} \cdot a$$

This value can then be applied to the `KEdge` that represents the message. In order to simplify the calculations, the equation is modified slightly.

$$y_{parent} = y \cdot \frac{a}{d}$$

This procedure is executed analogical for every message. The factor $f = \frac{a}{[b,c,d]}$ can be precalculated for each parent element. All these calculations are simplified for comprehensibility: offsets and margins are ignored.

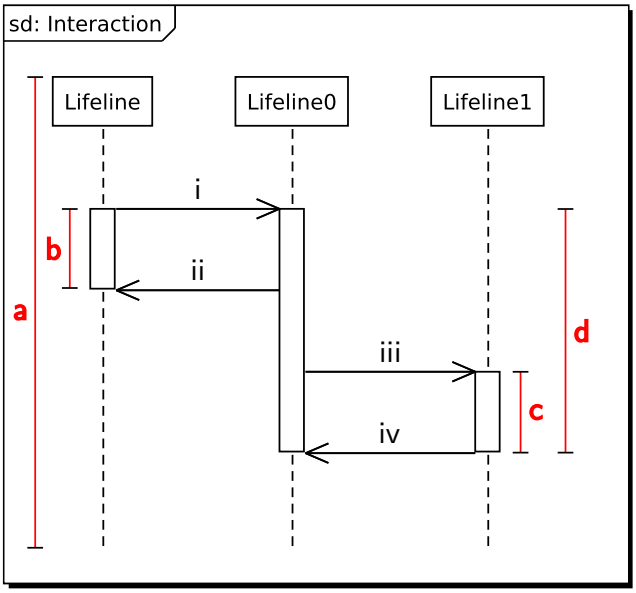


Figure 5.10. Illustration of the different scales involved into the calculation of the vertical coordinates of messages. (a), (b), (c), and (d) show the heights of the lifelines and the different execution specifications.

Once the message coordinates of a lifeline are set, the horizontal coordinates of the attached execution specifications have to be determined. Since execution specifications may be attached to a lifeline in a hierarchical way too, this has to be checked for before finally setting the horizontal coordinates. If this happens, the contained execution specification is drawn with a displacement of half times their width to the right. If no message is attached to an execution specification, a minimum height is used.

For each message that is connected to an execution specification, the vertical position of the connection point is reset in order to work around the behavior described above. A factor that is specific to the current execution specification is applied to the original coordinates.

As the last steps, the current lifeline's position and height are copied to the corresponding `KNode` and a destruction event is placed at the lower end of the lifeline, if such an event exists for that lifeline.

The only thing left now is to copy the coordinates of the comment objects which are not fixed to any lifeline or message. In addition to simply copying the stored coordinates to the `KGraph`, the connections of the comments are routed in this step. Connections to lifelines or their execution specifications that are located left or right of the comment are drawn horizontally, connections to messages are drawn vertically.

Table 5.1 and Table 5.2 give an overview over the different properties used within the algorithm. `SequenceDiagramProperties` are used in the actual layout algorithm for different purposes. Their name and description are specified, as well as the phases that make use of it.

`PapyrusProperties` are used while importing the `KGraph` from the editor (see Section 5.4). Their name and description is provided in Table 5.2.

Table 5.1. Overview over the `SequenceDiagramProperties`: their name, description, and the phases that rely on these properties are given.

Property	Description	Affected Phases
`MESSAGE_TYPE`	The type of a message	`SGraph` import, lifeline sorting, coordinate calculation, layout application
`BELONGS_TO_LIFELINE`	The lifeline to which an element of the SGraph belongs	`LGraph` import, cycle breaking, coordinate calculation,
`LAYERED_NODE`	The node in the layered graph that corresponds to a message	`LGraph` import, space allocation
`DESTRUCTION_EVENT`	The node in the KGraph that corresponds to the destruction event of a lifeline	`SGraph` import, layout application
`COMMENT_CONNECTION`	The KEdge that connects the comment to another element of the diagram	`SGraph` import, layout application
`LIFELINE_HEADER`	The height of the lifeline's header	coordinate calculation, layout application
`LIFELINE_Y_POS`	The vertical position of lifelines	coordinate calculation, layout application
`AREA_HEADER`	The height of the header of combined fragments	coordinate calculation
`TIME_OBSERVATION_WIDTH`	The width of time observations	`SGraph` import
`CONTAINMENT_OFFSET`	The offset between two nested areas	coordinate calculation
`LIFELINE_SPACING`	The horizontal space between two neighbored lifelines which may be set by the user	coordinate calculation, layout application
`MESSAGE_SPACING`	The vertical space between two neighbored messages which may be set by the user	space allocation, coordinate calculation, layout application
`LABEL_ALIGNMENT`	The alignment of message labels which may be set by the user	layout application
`LIFELINE_SORTING`	The lifeline sorting strategy that should be used in the algorithm which may be set by the user	lifeline sorting
`GROUP_AREAS`	If messages in areas should be grouped together which may be set by the user if the `SHORT_MESSAGES` lifeline sorter is chosen	lifeline sorting

Table 5.2. Overview over the PapyrusProperties: their name and description is given.

Property	Description
MESSAGE_TYPE	The numeric identifier of a message
NODE_TYPE	The numeric identifier of a node
EXECUTIONS	The list of execution specifications of a lifeline
EXECUTION	The SequenceExecution object that is attached to a KNode that represents an execution specification
AREAS	The list of areas in the diagram
ATTACHED_TO	The list of objects, a comment is attached to
ATTACHED_ELEMENT	Indicates to what kind of element a comment is attached
DESTRUCTION	The destruction event of a lifeline

5.4 Integration into Papyrus

In order to use the layout facilities of KIELER, the data from the editor has to be transformed to match the KGraph structures of KIELER. There is an existing implementation for GMF-based diagrams that had to be adapted to Papyrus and especially for sequence diagrams since they are different to other graph-like diagrams. The MultiPartDiagramLayoutManager was designed to import the KGraph for all kinds of Papyrus diagrams. A sequence diagram specific import has to be executed whenever the importer detects that the current diagram is a sequence diagram. The MultiPartDiagramLayoutManager had to be extended to match the peculiarities of these diagrams since some sequence diagram specific elements cannot be identified by the original version.

The main task of the adapted importer, however, is to annotate the KGraph elements. All kinds of shaped elements are transformed into KNodes and every connection is transformed into a KEdge. These elements have to be distinguished throughout the layout algorithm since their semantics are differing largely. Some relations may be determined from the hierarchy of the elements, but most of the context would be lost if the type of the different elements was unknown. Therefore, the numeric type of each element is stored for the KNodes and KEdges. The properties feature of the KGraph that was already introduced in Figure 5.2 is used to store the NODE_TYPE and MESSAGE_TYPE.

Another task that is performed by the MultiPartDiagramLayoutManager is the initialization of areas and execution specifications. These elements are handled different since their relations can not be determined from the structure of the KGraph. The KNode of an area does not have any information about the messages that are contained in the area. Similarly, the messages that are connected to an execution specification are determined by their position. Therefore, the sequence diagram specific representations of these elements are already built while creating the KGraph. Since there is no dependency to the sequence diagram layout plug-in in the Papyrus layout plug-in, the SequenceArea and SequenceExecution classes are not integrated into the SGraph structure. Instead, they are located in the Papyrus layout plug-in.

The actual import that creates the KGraph is started from the top-level element in the diagram. The GMF *EditParts* are traversed recursively starting at the PackageEditPart that represents the diagram editor. At first, only the KNodes are created. The connections between the corresponding elements are stored in a reference2EdgeMap that is traversed when all the shaped elements are already created. This procedure ensures that the source and the target KNodes are already existing when the connections are created.

In the following, the mapping of the diagram elements to the KGraph elements is described more detailed:

- The *Surrounding interaction* is transformed to the top level KNode. It is the first element that is traversed while importing the KGraph from the diagram. Every shaped element in the diagram is a descendant of this KNode.
- *Lifelines* are transformed to KNodes that are children of the surrounding interaction. They are created in a first step when iterating through the children of the top level element.
- Each *Interaction Use* as well as *Combined Fragments* are transformed to KNodes that are children of the surrounding interaction. These areas are created together with the lifelines since they share the same layer of hierarchy in the KGraph.
- *Interaction Operands* are transformed to KNodes that are children of the KNode of the combined fragment that they belong to.
- *Action Execution Specifications* and *Behavior Execution Specifications* are transformed to KNodes that are children of their lifeline's KNode.
- *Comments* and *Constraints* are transformed to KNodes that have the same hierarchical layer as the lifelines.
- *Destruction Events* are transformed to KNodes that are children of their lifeline's KNodes.
- *Duration Observations* as well as *Time Observations* are transformed to KNodes that are children of their lifeline's KNode.
- *Duration Constraints* and *Time Constraints* are transformed to KNodes that are children of the surrounding interaction.
- All types of *Messages* are transformed to KEdges. These KEdges are not part of the hierarchy of the shaped elements. Instead, they may connect elements from different hierarchical layers.

An overview of these elements and their hierarchy is shown in Table 5.3.

Coordinates for these elements are calculated by the layout algorithm. These coordinates have to be applied to the diagram when the calculations are finished. The shaped elements that are transformed to KNodes store their coordinates relative to their parent elements. The coordinates of messages are stored relative to the elements that the message is connected to.

Table 5.3. The different elements of the diagram and their hierarchy: For each *Diagram Element* that may occur in the diagram, the corresponding *EditPart* is specified as well as the *Type* that is used to identify the element. The diagram element is transformed into the `KGraph` element that is specified in the column `KGraph`. Column *Parent* shows the parent element in the `KGraph` if existing.

Diagram Element	EditPart	Type	`KGraph`	Parent
Surrounding Interaction	InteractionEditPart	2001	`KNode`	-
Lifeline	LifelineEditPart	3001	`KNode`	2001
Interaction Use	InteractionUseEditPart	3002	`KNode`	2001
Combined Fragment	CombinedFragmentEditPart	3004	`KNode`	2001
Interaction Operand	InteractionOperandEditPart	3005	`KNode`	3004
Action Execution Specification	ActionExecutionSpecification EditPart	3006	`KNode`	3001
Behavior Execution Specification	BehaviorExecutionSpecification EditPart	3003	`KNode`	3001
Comment	CommentEditPart	3009	`KNode`	2001
Constraint	ConstraintEditPart	3008	`KNode`	2001
Destruction Event	DestructionOccurrenceSpecificationEditPart	3022	`KNode`	3001
Time Constraint	TimeConstraintEditPart	3019	`KNode`	3001
Time Observation	TimeObservationEditPart	3020	`KNode`	2001
Duration Constraint	DurationConstraintEditPart	3021	`KNode`	3001
Duration Observation	DurationObservationEditPart	3024	`KNode`	2001
Synchronous Message	MessageEditPart	4003	`KEdge`	-
Asynchronous Message	Message2EditPart	4004	`KEdge`	-
Reply Message	Message3EditPart	4005	`KEdge`	-
Create Message	Message4EditPart	4006	`KEdge`	-
Delete Message	Message5EditPart	4007	`KEdge`	-
Lost Message	Message6EditPart	4008	`KEdge`	-
Found Message	Message7EditPart	4009	`KEdge`	-

More detailed, the coordinates of the message's source point are relative to its source element, the coordinates of the target point are relative to the target element.

The next chapter evaluates the aesthetic criteria specific to sequence diagrams and applies these criteria to the produced results of the algorithm before evaluating the execution time of the algorithm and comparing it to different approaches.

Chapter 6

Evaluation

In this chapter, we evaluate the performance and quality of the developed layout algorithm. In Section 6.1, the aesthetic criteria used to measure the quality of layouts are introduced and motivated. Section 6.2 uses these criteria to evaluate the produced layouts. After these, the performance of the algorithm is evaluated in Section 6.3. Finally the algorithm is compared to other approaches in Section 6.4.

6.1 Aesthetic Criteria for Sequence Diagram Layout

This section gives an overview of common aesthetic criteria for graph layout and chooses those that are of interest for sequence diagrams. It also introduces known aesthetic criteria specific to sequence diagrams.

6.1.1 General Aesthetic Criteria

Fundamental work was done by Helen C. Purchase who published several papers on various topics in graph drawing aesthetics. These include papers on the effect of graph layout on human performance when dealing with automatically laid out graphs [Pur98] and on the question of how much different aesthetic criteria affect the readability of graphs [PCJ96]. She also developed metrics for the readability of these layouts [Pur02].

Her work covered the evaluation of several common aesthetic criteria for graphs as described in Section 2.4: the number of *edge crossings*, the *length* of the edges, the *area* consumed by the drawing, the number of *edge bends*, the *angle* between neighboring edges of a node, the *aspect ratio* of the drawing, and the *balance* of the nodes in the drawing.

However, sequence diagrams do not correspond to our definition of a graph. One may reason that they share the concept of nodes connected by edges, but they differ in central aspects. Lifelines, that may be interpreted as the sequence diagram's nodes, do not behave like nodes in general graphs. They can be interpreted as the nodes that are connected by the messages, but these messages are not connected to the node-like part of the lifeline at the top. Instead, messages are connected to the dotted lines below the lifeline's head. Additionally, messages are connected to these lines in a predefined order.

Lifelines, in turn, cannot be freely placed on the diagram. They have to be placed at the top of the drawing, side-by-side (except for lifelines that have a create message). The horizontal order of the lifelines, however, may be changed for the purpose of a good layout. This reduces the 2-dimensional problem of node placement in general graphs to a 1-dimensional problem

that is more like "node sorting". There are further differences, namely the integration of areas like interactions that may cover parts of lifelines and messages.

Due to these differences, it is not obvious whether the common aesthetic criteria can be applied to sequence diagrams. In the following, we will examine each criterion and discuss if it should be applied to sequence diagrams.

The minimization of *edge crossings* is a criterion that cannot be adopted to sequence diagrams without some modifications. Messages usually don't cross each other in sequence diagrams. However, messages do cross lifelines when connecting two lifelines that are not positioned next to each other. It is generally preferable to have as few crossings as possible in any kind of graph, so this criterion is of interest for sequence diagrams too.

Another optimization goal that is desirable for any kind of graph is the minimization of the *edge length*. Both edge crossings and edge length are of course important for a good layout of sequence diagrams since the ability of following the messages is crucial for the understandability of the diagram.

The *area* that is consumed by a sequence diagram is a criterion that the layout algorithm has very little influence over. Since lifelines are placed side-by-side, the horizontal size of the layout only depends on the width of the lifelines and the spacing between them. Vertical size is mainly influenced by message spacing and message distribution.

Edge bends are obstacles to following the flow of an edge easily. However, messages are not supposed to have bends at all; they should normally be drawn horizontally instead. In seldom cases, messages may be drawn sloping downwards; but even in this case they are drawn as straight lines. Hence, the edge bends criterion is not applicable to sequence diagrams.

The same holds for the *angle* criterion, which measures the angle between two edges connected to the same node. Messages are drawn horizontally, so the angle will always be zero. Furthermore, it is not even clear what "nodes" in sequence diagrams are with respect to this criterion.

For the *aspect ratio* criterion, as for the area criterion, it is important to mention that the width of the diagram is influenced only by spacing of the lifelines and their individual width. This reduces the problem to the height of the diagram, which, as described above, is not very variable, thus rendering this criterion useless.

Since lifelines are placed side-by-side in sequence diagrams, the *balance* criterion may only be interesting for diagrams that contain very long message labels. These labels could cause neighboring lifelines to be separated from each other if they are connected by a message with such a label. In all other cases, lifelines are distributed equally over the horizontal spread of the diagram.

In a nutshell, there are the following criteria left that can reasonably be applied to sequence diagrams:

- Edge crossings
- Edge length
- (Area)

▷ (Aspect ratio)

▷ (Balance)

Edge crossings and edge length however are strongly related for sequence diagrams. For equally distanced and sized lifelines, they are even linear dependent. Every edge that connects two neighboring lifelines has an edge length of lifeline width plus the horizontal lifeline spacing. Every crossing with a lifeline adds the same amount of edge length to the equation. A layout without any edge crossings therefore also is a layout with minimum edge length. This correlation ceases to hold only for big differences in width or horizontal distance between the lifelines. This conforms to former work of Purchase [Pur97], who published a study that tested which aesthetic criterion was the most important for the understandability of graphs in general. It states that minimizing the number of edge crossings is by far the most important optimization goal with respect to aesthetics.

6.1.2 Aesthetic Criteria Specific to Sequence Diagrams

In addition to the criterion of edge crossings that turned out to be the most important one for general graphs, one can develop other criteria that may be more specific to sequence diagrams. In order to find such criteria, Wong and Sun [WS05] started from scratch and had a look at basic cognitive science. The theory of perception had a particularly remarkable impact on their development of aesthetic criteria for the layout of UML diagrams. While most of their theoretical work served to prove the criteria above, the law of proximity was the key to develop another requirement for a good layout: it states that objects close to each other are regarded as a group. For sequence diagrams, as for graphs in general, this implies that nodes which are strongly connected should be drawn in close proximity.

With that, the *slidability* property defined by Poranen et al. [PMN03] is shown to be an interesting property as well. If the drawing of the graph does not fit on the screen, it is necessary to scroll the diagram. If this happens, slidability becomes an interesting property: it measures if it is possible to view all the relevant information in a window of fixed size. More precisely, slidability is the property that measures if it is possible to slide the fixed size window over the whole diagram without losing the context of the information that is visible inside the window at any given time.

Common practice implies another goal too. For sequence diagrams it is desirable to have the *source lifeline on the left side* of the drawing [PMN03]. The source lifeline here is the one lifeline where interaction starts. More precisely, it is the lifeline that is the source lifeline of the uppermost message in the diagram. In some cases, this message may not be unique; this is rather seldom in common diagrams and may be handled in different ways.

Finally, there is another interesting aspect of sequence diagram layout. Messages may be grouped together by several types of areas in sequence diagrams (e.g. interactions, combined fragments). The lifelines that are affected by these messages should be drawn in direct neighborhood in order to highlight the strong correlation of these lifelines by keeping the area as compact as possible. This property can be combined with the edge length minimization

property by modifying edge weights. With that, at least the three stated goals may be satisfied as well as heuristically possible.

6.2 Quality of Produced Layouts

A collection of real world sequence diagrams from various areas and a few randomly generated sequence diagrams are used to evaluate the quality of the layout. The real world diagrams are collected from different sources and remodeled in the Papyrus sequence diagram editor. The random diagrams are just a collection of lifelines and messages that were arbitrarily connected. The diagrams contain three to eleven lifelines and up to 38 messages which covers the range that is used for typical sequence diagrams [PMN03].

The quality of the produced layouts is evaluated in terms of the aesthetic criterion of message-lifeline crossings in this section. Each of the diagrams has a prescribed order of the lifelines that was determined by the respective creator of the diagram. With that, the human preferences can be compared to the computer-measurable aesthetic criteria.

Automatic layout as developed in this thesis is applied to each of the diagrams. Both lifeline sorting strategies, *layer-based* lifeline sorting and *long message avoiding* lifeline sorting, are used. The number of message-lifeline crossings is compared for the three different lifeline sorting methods. In addition to that, the minimal number of crossings is determined and compared to the results of the lifeline sortations. Table 6.1 shows the results of this evaluation.

The main fact in Table 6.1 is that the original lifeline order produces fewer message-lifeline crossings in the sum than both of the lifeline sorting strategies of the algorithm. At first sight, this could imply that there is no need for automatic lifeline sorting. However, most of the original diagrams were optimized manually. Claiming automatic layout to produce better results than thoroughly manually optimized diagrams would be a very optimistic goal especially for heuristic algorithms. The real advantage of automatic layout is the time saved. Optimizing layout by hand may take hours depending on the complexity of the diagram. Automatic layout on the contrary is done within a click of the mouse button (see Section 6.3 for details).

In spite of the worse sum of the message-lifeline crossings, the *layer-based* lifeline sorter produces exactly the same result as the original drawing in many cases. This holds especially for the real world diagrams that were modeled by hand. For random diagrams that do not represent a real use-case, the *layer-based* lifeline sorter produces results that are worse in general.

The higher number of crossings of the *long message avoiding* lifeline sorter in comparison to the *layer-based* lifeline sorter is remarkable. It is especially remarkable since the *long message avoiding* lifeline sorter was implemented following an approach of McAllister which is dedicated to minimizing the number of these crossings [McA99]. The greater sum of the crossings, however, arises from few outliers that produce very poor results (see Figure 6.1 for an example). For most of the diagrams, the number of crossings is equal or smaller than for the *layer-based* lifeline sorter.

Table 6.1. The number of message-lifeline crossings in the diagram with different lifeline sorting strategies. The original number of crossings is given as well as the minimal possible number. These are compared to the crossing number of the *layer-based* and the *long message avoiding* (LMA) lifeline sorter. An asterisk in the column of these lifeline sorters denotes an equal lifeline order as in the original diagram.

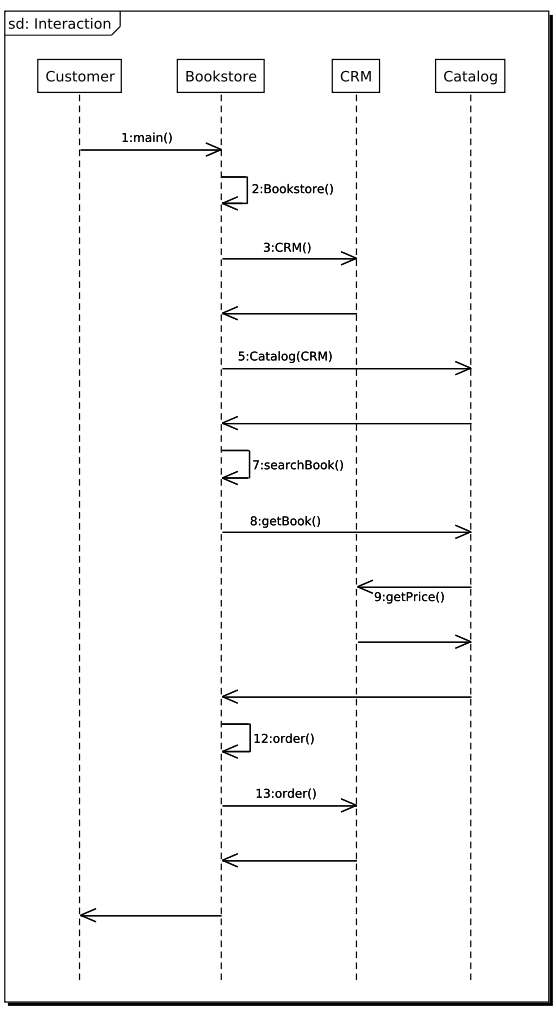
Diagram (Lifelines / Messages)	Original	Minimal	Layer-Based	LMA	Reference
Internet Shopping (5 / 12)	8	5	8*	5	Figure A.1
Stock (4 / 16)	4	4	4*	4	Figure A.2
Bookstore (4 / 15)	4	4	4*	8	Figure A.3
Bookstore Create (4 / 16)	2	2	4	2*	Figure A.4
Book Journey (5 / 16)	4	4	4*	4*	Figure A.5
Snow Clearing System (4 / 18)	4	4	4*	4*	Figure A.6
Restaurant (3 / 8)	0	0	0*	0*	Figure A.7
Movie Rental (5 / 19)	4	4	4*	7	Figure A.8
KiVi (7 / 20)	13	8	13*	11	Figure A.9
Unfulfilled Orders (5 / 8)	3	2	2	2	Figure A.10
Flirt (4 / 9)	3	2	3	3*	Figure A.11
Model-View-Controller (4 / 16)	6	6	8	8	Figure A.12
Loan (7 / 17)	4	2	5	4	Figure A.13
Random I (4 / 8)	2	2	6	4	Figure A.14
Random II (6 / 14)	10	9	22	15	Figure A.15
Big Random (11 / 38)	18	18	19	32	Figure A.16
Sum	89	76	110	113	

When analyzing the poor results of the *long message avoiding* lifeline sorter (Figure A.3, Figure A.8, and Figure A.16), those diagrams with a source lifeline that has a small degree of connection seem to be very error-prone when the target lifeline of the first message has a high degree. This arises from the calculation of the selection factor sf. It is calculated from the lifeline's degree and weighted sum of the connected messages, the tl value:

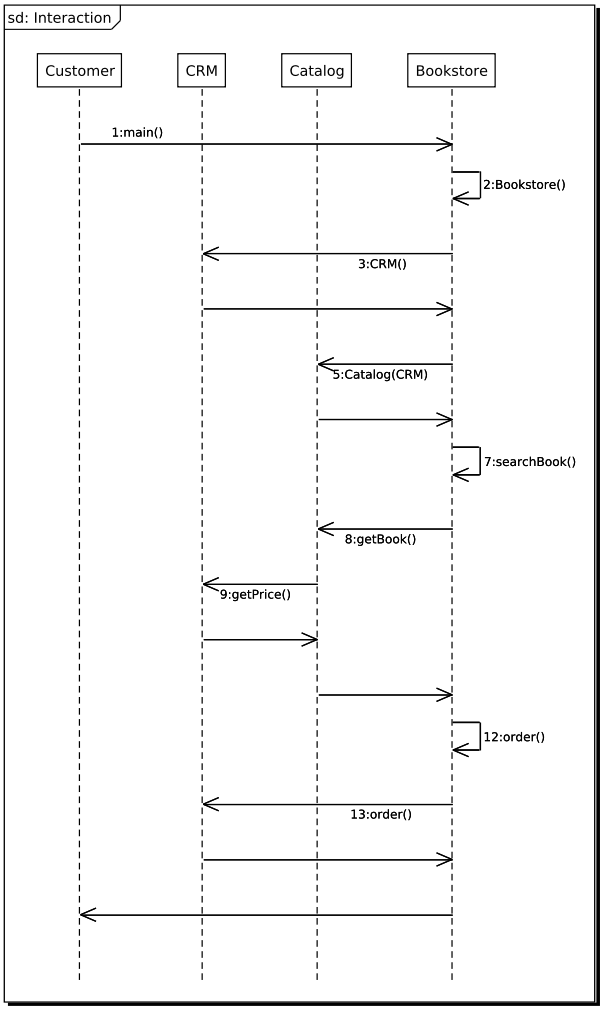
$$sf = degree(lifeline) - 2 \cdot tl(lifeline).$$

The lifeline with the smallest selection factor sf is chosen to be placed next. A high degree of a lifeline therefore decreases the chance to be placed next especially when the connectivity to the source lifeline (represented by the tl value) is small. Experimenting with the weight of the degree and the tl value may change that behavior.

Poor results are computed for the big diagram (Figure A.16) that contains many messages as well. The reason for that is not as obvious as above. The selection of a different starting lifeline, however, is noticeable here. Since the diagram is created randomly, two messages can be the starting message here. The *long message avoiding* lifeline sorter chooses the one with the higher degree in that case. Choosing the lifeline with the lower degree could be a better choice if the messages are connected to different lifelines.



(a) The bookstore diagram with layer-based lifeline sorting, which yields the same result as the original lifeline sorting (4 message-lifeline crossings).



(b) The bookstore diagram with long message avoiding lifeline sorting (8 message-lifeline crossings).

Figure 6.1. An example of a diagram where the layer-based lifeline sorter yields exactly the same results as the manual layout of the author. The long message avoiding lifeline sorter fails to optimize this diagram.

6.3 Performance Evaluation

Evaluating the performance of an algorithm usually compares the algorithm to other implementations. Examining the asymptotic performance is another common evaluation of an algorithm's performance. Both, however, are not very convenient in the special case of this layout algorithm. Sequence diagram layout is not covered very well in academic research and therefore, no data is available for comparison. The asymptotic running time may be computed by extensive evaluation; it is not very important for normal sequence diagrams, though. Typical sequence diagrams consist of less than 10 lifelines and 20 messages. Even very big diagrams do not exceed 20 lifelines.

In spite of this, we measured the execution time of the layout algorithm and its sub-tasks. The evaluation of the execution time was done with the same diagrams that were already used for the analysis of the layout quality in the last section. As for the quality analysis, every diagram was laid out with every lifeline sorting strategy.

For each of these layout runs, the exact execution time was measured for the whole run as well as for the single layout phases (such as *graph importing*, *cycle breaking*, and *lifeline sorting*). Table 6.2 shows the complete results of the performance evaluation. Even for the big diagram (Figure A.16), the execution was very fast. The maximum of the time consumed for the whole algorithm was 20.138 milliseconds. The execution time stayed below 25 milliseconds for all diagrams. The average execution time for the whole algorithm was 11.253 milliseconds with several instances being laid out in less than 8 milliseconds. The fastest recorded layout was measured to be 4.551 milliseconds for the *Random II* diagram (Figure A.15) with the *layer-based* lifeline sorter.

These execution times are short enough to be perceived as *instant* by users of the layout algorithm.

Since each of the different lifeline sorting algorithms was used for each of the diagrams, the execution times of them can be compared to each other in a reliable way. A clear ranking could be established between these algorithms.

The *interactive* lifeline sorter obviously is the fastest algorithm since it simply compares the horizontal positions of the lifelines and returns the order of the lifelines as it was. It takes less than 0.02 milliseconds to compute this sortation even for the biggest diagrams in the test.

The *long message avoiding* lifeline sorter, on the contrary, is the one that needs most execution time. It builds its own lightweight graph representation before computing the actual order of the lifelines. This causes the *long message avoiding* lifeline sorter to take up to 0.4 milliseconds for its calculations in worst case.

The *layer-based* lifeline sorter, finally, is located between the previous ones with respect to execution time. It does not have to build its own graph representation, but uses both of the existing graph representations (`SGraph` and `LGraph`) instead. The maximum calculation time was 0.15 milliseconds for the *layer-based* lifeline sorter.

These execution times are interesting with respect to their relative duration. Their absolute values, however, are not crucial when interacting with human users at all.

Table 6.2. The execution time of the layout algorithm measured in milliseconds. All sample diagrams were laid out with each of the three lifeline sorting strategies.

Diagram (Lifelines / Messages)	Interactive	Layer-Based	LMA	Reference
Internet Shopping (5 / 12)	19.864	19.892	13.104	Figure A.1
Stock (4 / 16)	14.210	15.650	8.838	Figure A.2
Bookstore (4 / 15)	11.264	11.052	11.378	Figure A.3
Bookstore Create (4 / 16)	14.534	15.498	12.525	Figure A.4
Book Journey (5 / 16)	13.467	14.712	12.139	Figure A.5
Snow Clearing System (4 / 18)	12.636	12.914	12.125	Figure A.6
Restaurant (3 / 8)	7.320	6.421	6.653	Figure A.7
Movie Rental (5 / 19)	13.728	15.851	15.471	Figure A.8
KiVi (7 / 20)	12.167	12.397	13.246	Figure A.9
Unfulfilled Orders (5 / 8)	6.999	5.963	6.302	Figure A.10
Flirt (4 / 9)	7.747	7.227	8.198	Figure A.11
Model-View-Controller (4 / 16)	11.688	11.054	7.110	Figure A.12
Loan (7 / 17)	11.195	11.376	11.313	Figure A.13
Random I (4 / 8)	5.009	5.062	4.938	Figure A.14
Random II (6 / 14)	7.415	4.551	7.645	Figure A.15
Big Random (11 / 38)	20.138	11.780	18.346	Figure A.16
Average	11.838	11.337	10.583	
Minimum	5.009	4.551	4.938	
Maximum	20.138	19.892	18.346	

6.4 Comparison to Different Approaches

Several tools and editors provide automatic layout for sequence diagrams. The capabilities of these layout algorithms, however, differ a lot.

Some sequence diagram layout algorithms are integrated into a graphical editor whilst others produce graphical sequence diagrams as a result of a certain kind of input, such as textual sequence diagram representations. The Papyrus framework provides a full graphical editor that is equipped with drag and drop features and live validation.

Since there is only little degree of freedom for most of the layout tasks, most of the algorithms concentrate on these tasks. With that, askew messages and lifelines are aligned and straightened, which saves a lot of manual work. However, the layout is not optimized in any way. Graph-like diagrams are usually optimized whenever automatic layout is applied. Since sequence diagrams do not match a common graph structure, optimizing its layout is not investigated as well as for normal graphs.

With this layout algorithm we tried to go new ways. The vertical positioning of the messages was reinvented inspired by the layering of nodes in a layered graph. With that, several messages could share the same vertical position if they are not ordered explicitly. This saves vertical space and highlights possible parallelism that is hidden by an arbitrary

ordering of these messages.

In addition to that, the horizontal order of the lifelines may be modified automatically if desired by the user. Several lifeline sorting strategies were implemented that are tailored to optimize different aesthetic criteria. Each of them is dedicated to a better readability of the diagram. User defined lifeline sorting is still available through the interactive lifeline sorter.

After evaluating the results and capabilities of the developed sequence diagram layout algorithm in this chapter, now is the time for a look at the whole thesis again. The next chapter concludes the thesis by summarizing the contributions and taking a look at future work.

Chapter 7

Conclusion

The previous chapters gave insight into the theoretical ideas as well as the implementational details of the proposed sequence diagram layout algorithm. This chapter at last takes a step back and gives an overview of the contributions of this thesis in Section 7.1 before Section 7.2 describes what kind of work on this topic may be done in future.

7.1 Summary

In this thesis, a layout algorithm for sequence diagrams was developed. The algorithm was integrated into the Papyrus project. Some concepts and data-structures were adopted from the KIELER project. In particular, the layered graph concept was adapted to handle the vertical positioning of messages in sequence diagrams. Their layering is done by the `NetworkSimplexLayerer` of KIELER's KLay Layered algorithm.

As a major improvement over existing sequence diagram layout algorithms, several strategies for lifeline sorting were implemented. The horizontal order of the lifelines may be changed by the algorithm which aims at improving different aesthetic criteria. Message-lifeline crossings are minimized by the *long message avoiding* lifeline sorting strategy. Within this strategy, the importance of messages contained in an interaction or combined fragment can be emphasized by *grouping according to areas*. The flow of the messages should be easy to follow when the *layer-based* lifeline sorter is chosen. Finally, the *interactive* lifeline sorter passes the power of lifeline sorting to the user.

Label placement is another feature that was adapted to sequence diagrams in this thesis. Some label placement strategies known from common graph layout could be adopted, others had to be adapted. Centered label placement as originally integrated in the Papyrus editor, as well as placement near the source lifeline are common label placement strategies. The placement centered between the two first lifelines regarding the direction of the message, however, is a new approach that was developed during this thesis.

The layout algorithm was connected to the Papyrus sequence diagram editor that provides GMF-based sequence diagram editing. These diagrams can be created and modified using the drag and drop feature of Papyrus. Diagrams are instantly validated, preventing semantically incorrect sequence diagrams. The Papyrus sequence diagram editor had to be connected to KIELER and its layout infrastructure KIML.

Sequence diagram specific layout options were integrated into KIML, enabling customizable layout for these diagrams. Different options were implemented to modify spacings and the

chosen strategies for lifeline sorting and label placement.

The execution time of the layout algorithm is short enough to be perceived as *instant layout* by human users which is very important for the acceptance of such an algorithm. No run of the layout algorithm with normal-sized diagrams took more than 0.05 seconds during the performance analysis.

7.2 Future Work

The broken label placement feature is obviously the first place to start when talking about future optimizations of the existing implementation. Further investigation of the coordinate transformations within the Papyrus framework could explain, how the calculated coordinates have to be transformed in order to compensate for the obscure behavior in that special case.

Since the Papyrus framework is under continuous development, several changes had to be made during the implementation of the layout algorithm. Some of these changes affected the results of the layout algorithm which forced us to modify some parts of the algorithm. Therefore, the results of the layout algorithm should be checked from time to time in order to adapt the algorithm to further changes.

Due to obscure coordinate transformations when applying coordinates to the diagram, the label placement feature is broken for Papyrus' most recent version when execution specifications are involved. It is not evident what kind of coordinate system the coordinates of such labels should be given in. Currently, the labels of such messages are spread all over the diagram since the coordinate system could not be determined.

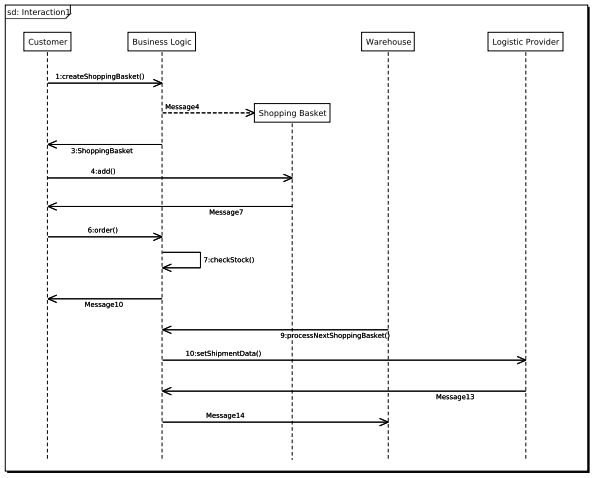
So far, the proposed layout algorithm is only applicable to sequence diagrams that are modeled with the Papyrus sequence diagram editor. However, since the actual algorithm is a general purpose sequence diagram layout algorithm, it could be integrated into different sequence diagram editors. A more general interface could be established without too much effort to be able to apply the algorithm to a wide range of sequence diagram editors.

Since graph layout is all about optimization of different aesthetic criteria, this is a major goal for every automatic layout algorithm. Aesthetic criteria have been established for different graph-based structures in the past. For sequence diagrams, however, many common criteria are not applicable. Several aesthetic criteria were examined in this thesis but there is still a lot of work to do. Starting from demanding the first lifeline to be the one that starts the communication, new criteria may be developed that may further improve layout of sequence diagrams.

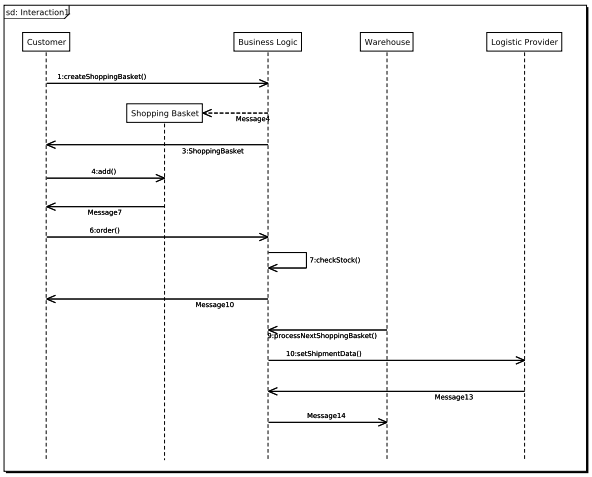
The lifeline sorting strategies that were developed and implemented in this thesis are based on different optimization goals. According to new aesthetic criteria, further lifeline sorting strategies may be developed. These strategies may even combine several different goals, improving the readability of the laid out sequence diagrams further.

Appendix A

Sample Sequence Diagrams

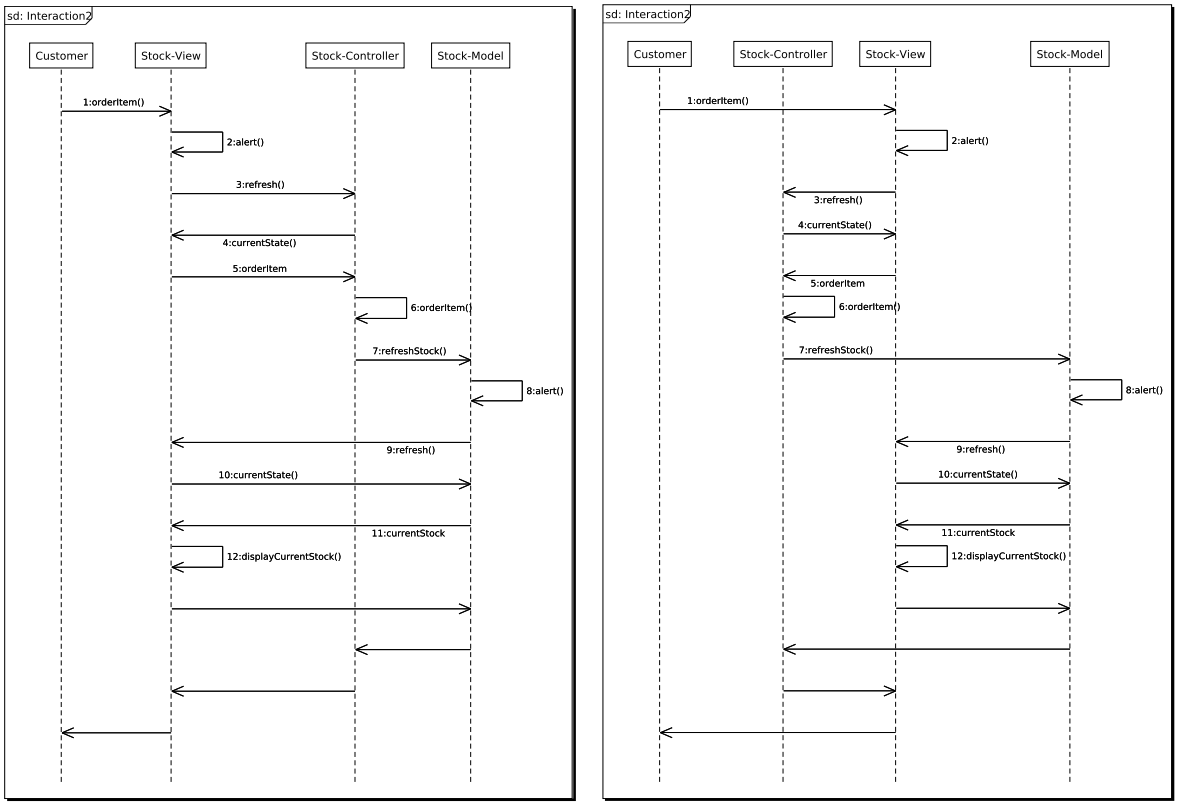


(a) *Original* and *layer-based* lifeline sorting.



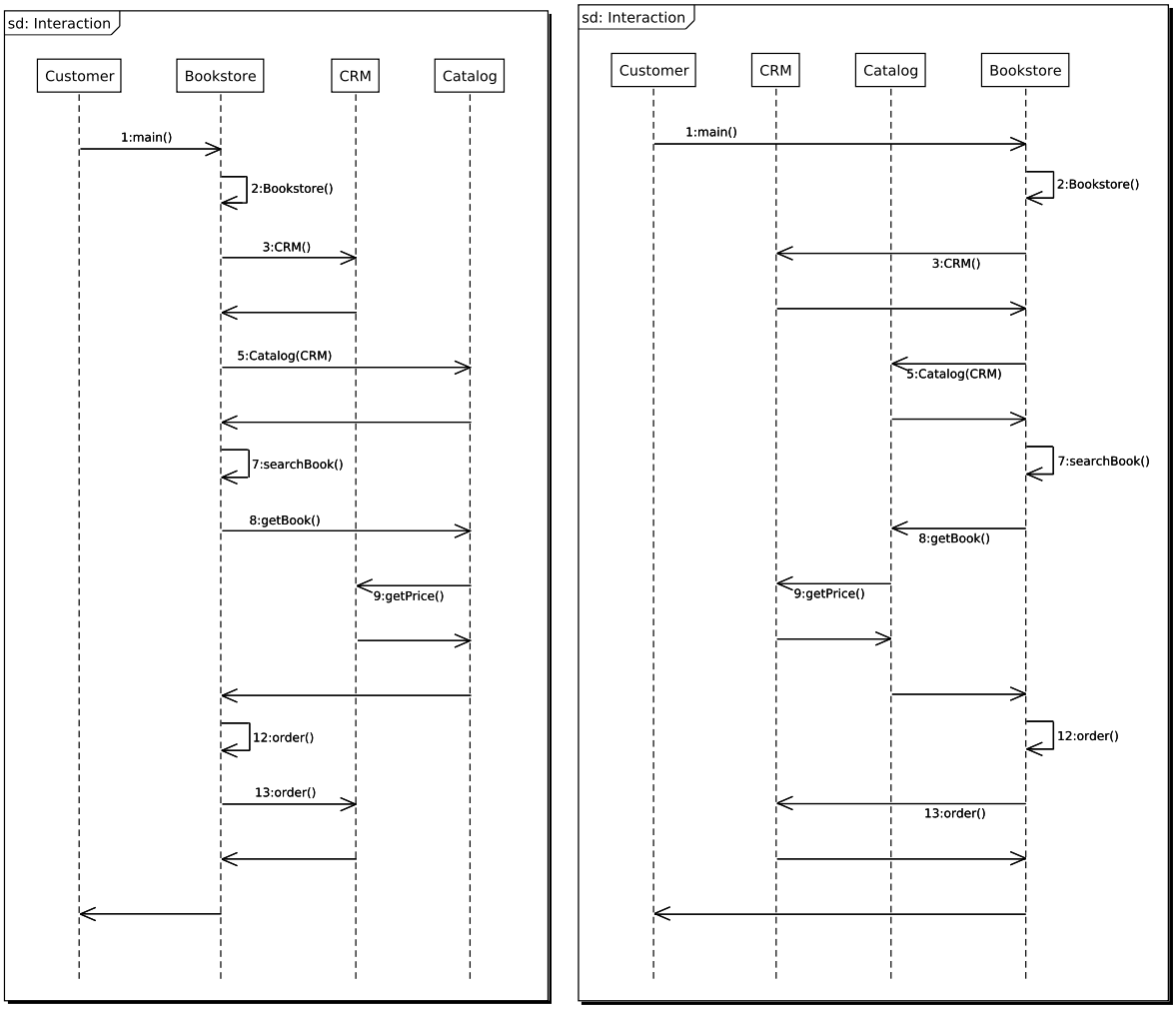
(b) *Long message avoiding* lifeline sorting.

Figure A.1. The *Internet Shopping* diagram.



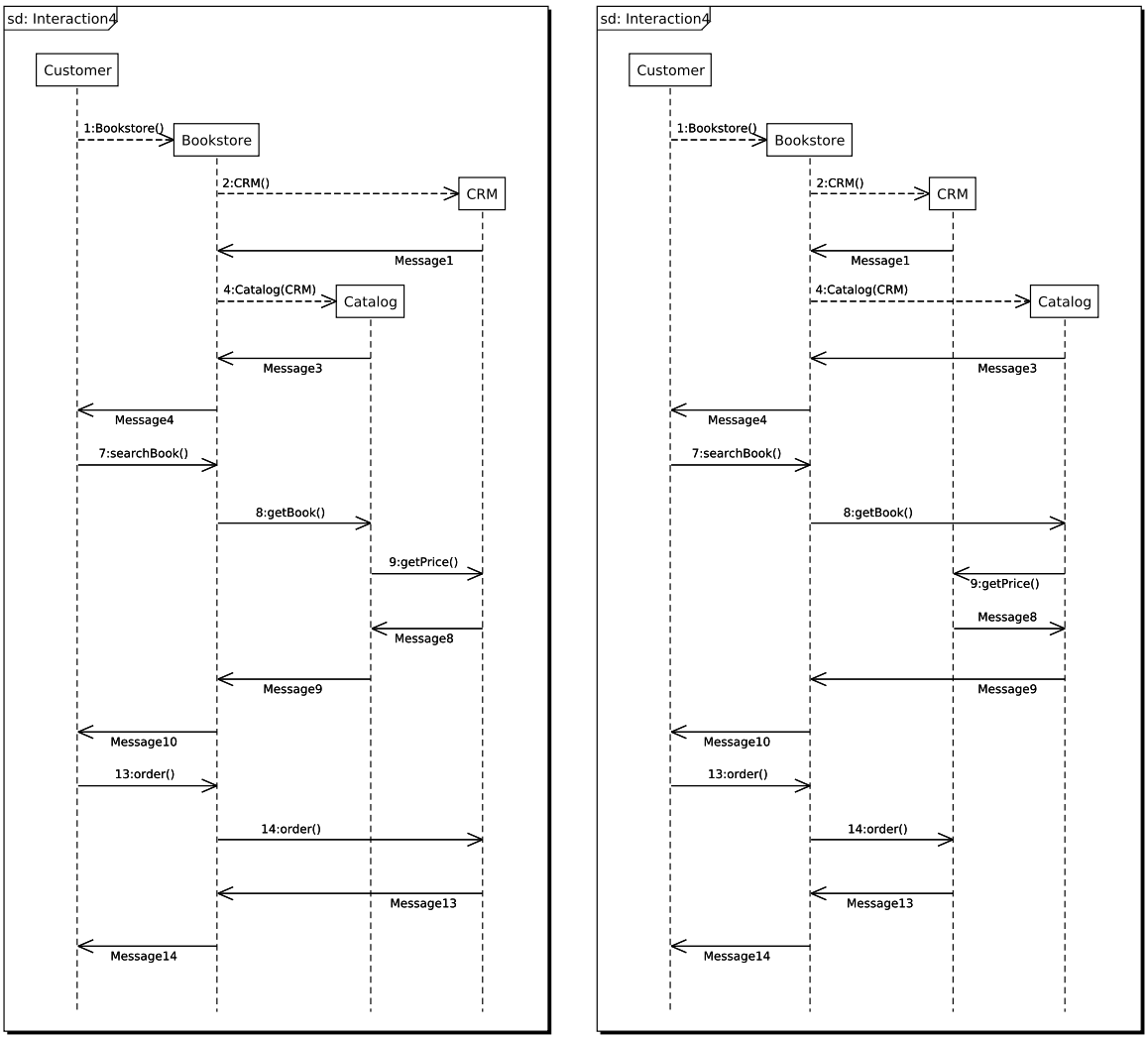
(a) *Original* and *layer-based* lifeline sorting. **(b)** *Long message avoiding* lifeline sorting.

Figure A.2. The *Stock* diagram.



(a) *Original* and *layer-based* lifeline sorting. **(b)** *Long message avoiding* lifeline sorting.

Figure A.3. The *Bookstore* diagram.



(a) *Original* and *long message avoiding* lifeline sorting.

(b) *Layer-based* lifeline sorting.

Figure A.4. The *Bookstore Create* diagram.

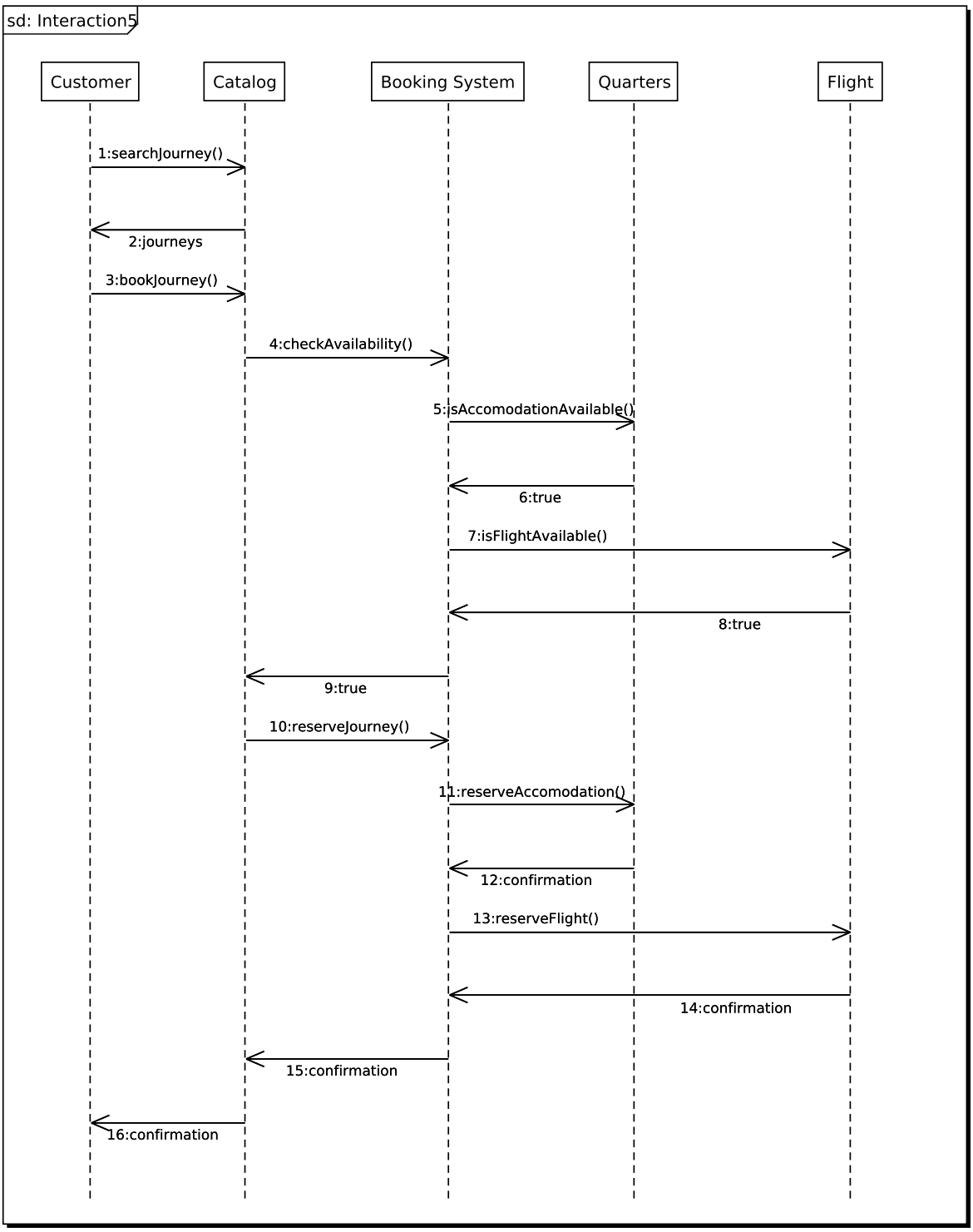


Figure A.5. The *Book Journey* diagram. All lifeline sorters yield the same result for this diagram.

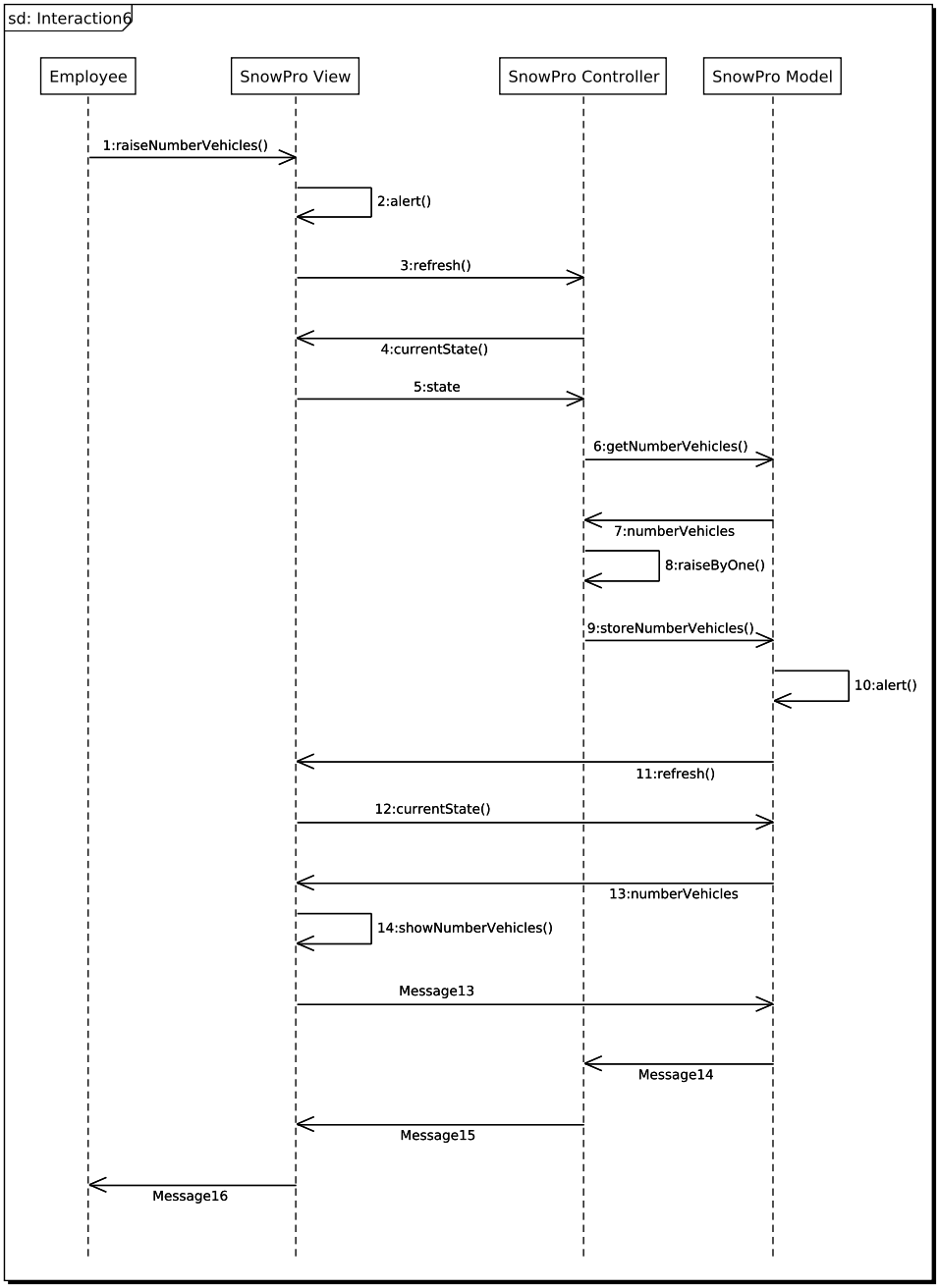


Figure A.6. The *Snow Clearing System* diagram. All lifeline sorters yield the same result for this diagram.

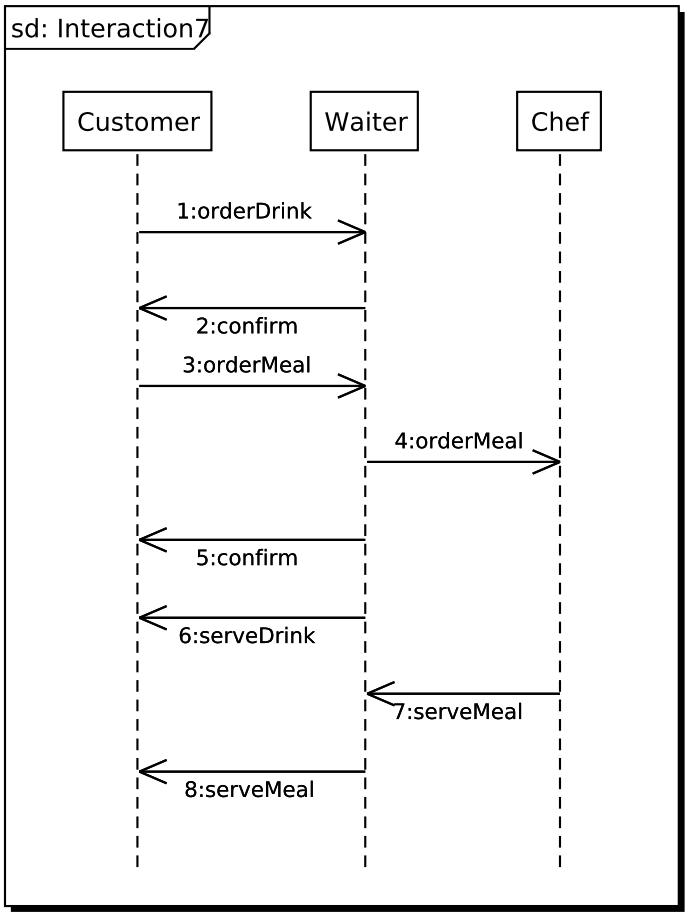
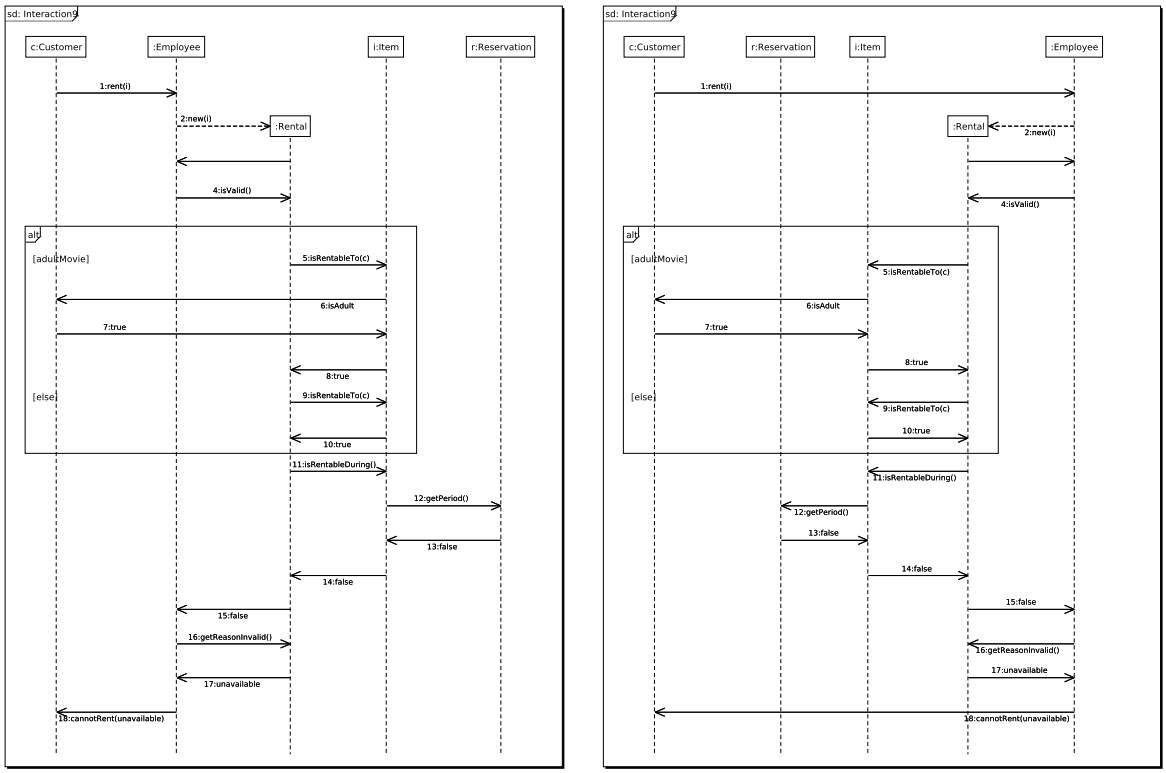
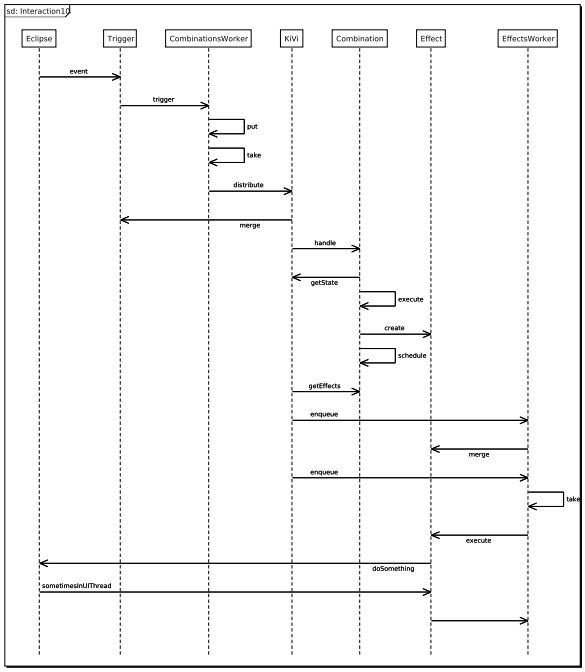


Figure A.7. The *Restaurant* diagram. All lifeline sorters yield the same result for this diagram.

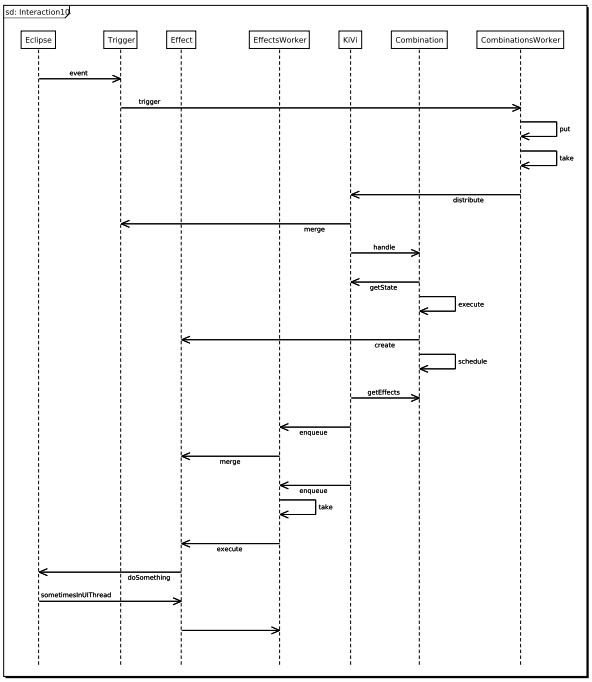


(a) *Original* and *layer-based* lifeline sorting. **(b)** *Long message avoiding* lifeline sorting.

Figure A.8. The *Movie Rental* diagram.

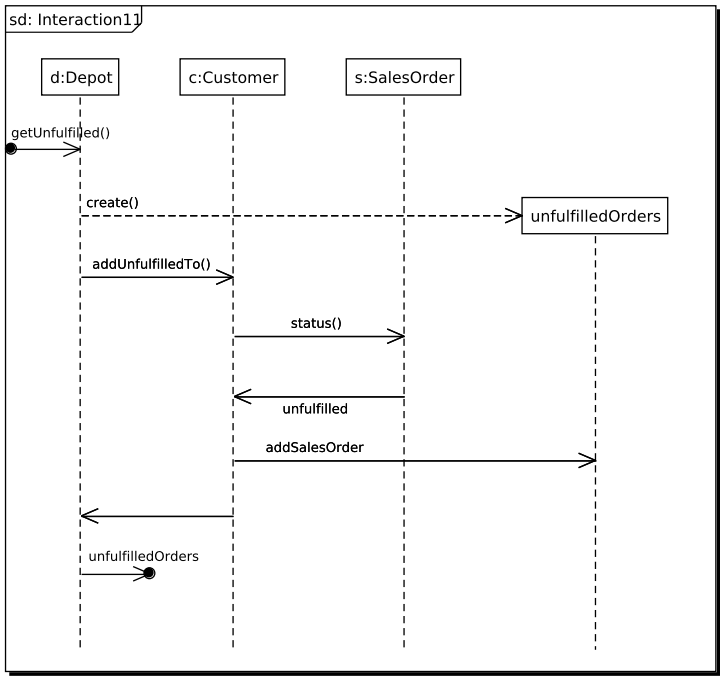


(a) *Original* and *layer-based* lifeline sorting.

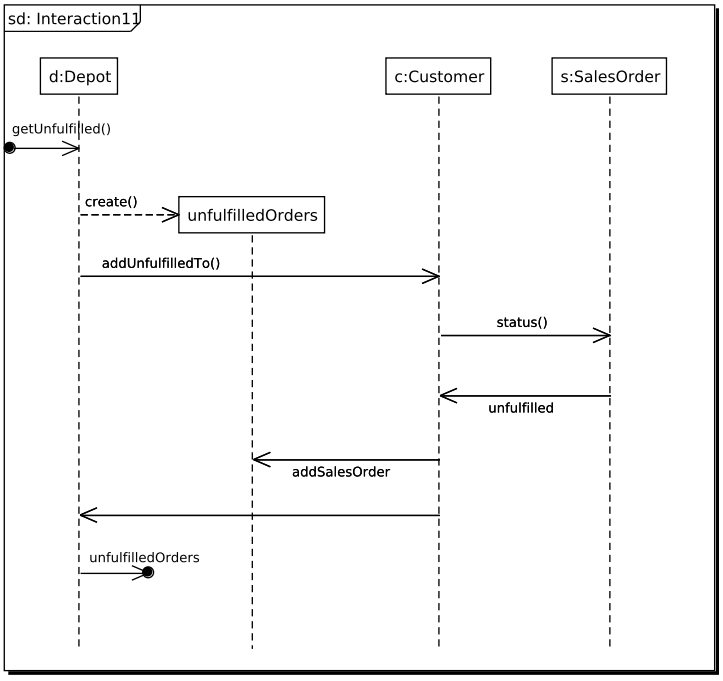


(b) *Long message avoiding* lifeline sorting.

Figure A.9. The *KiVi* diagram.

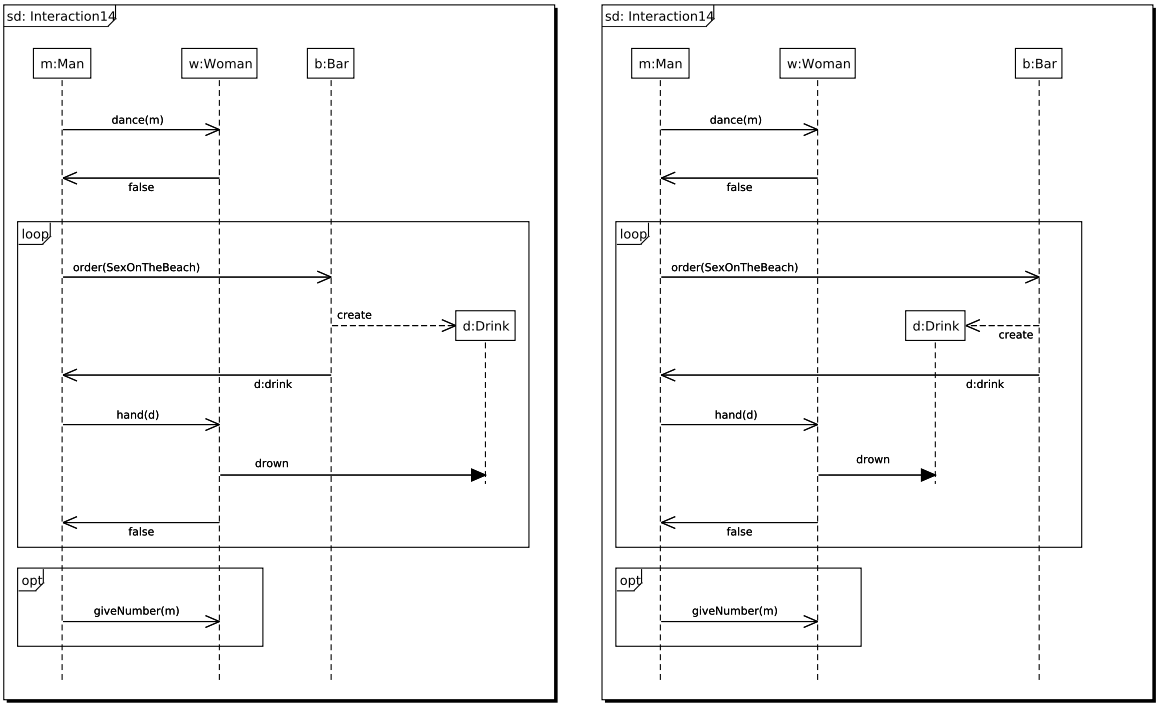


(a) *Original* lifeline sorting.



(b) *Layer-based* and *long message avoiding* lifeline sorting.

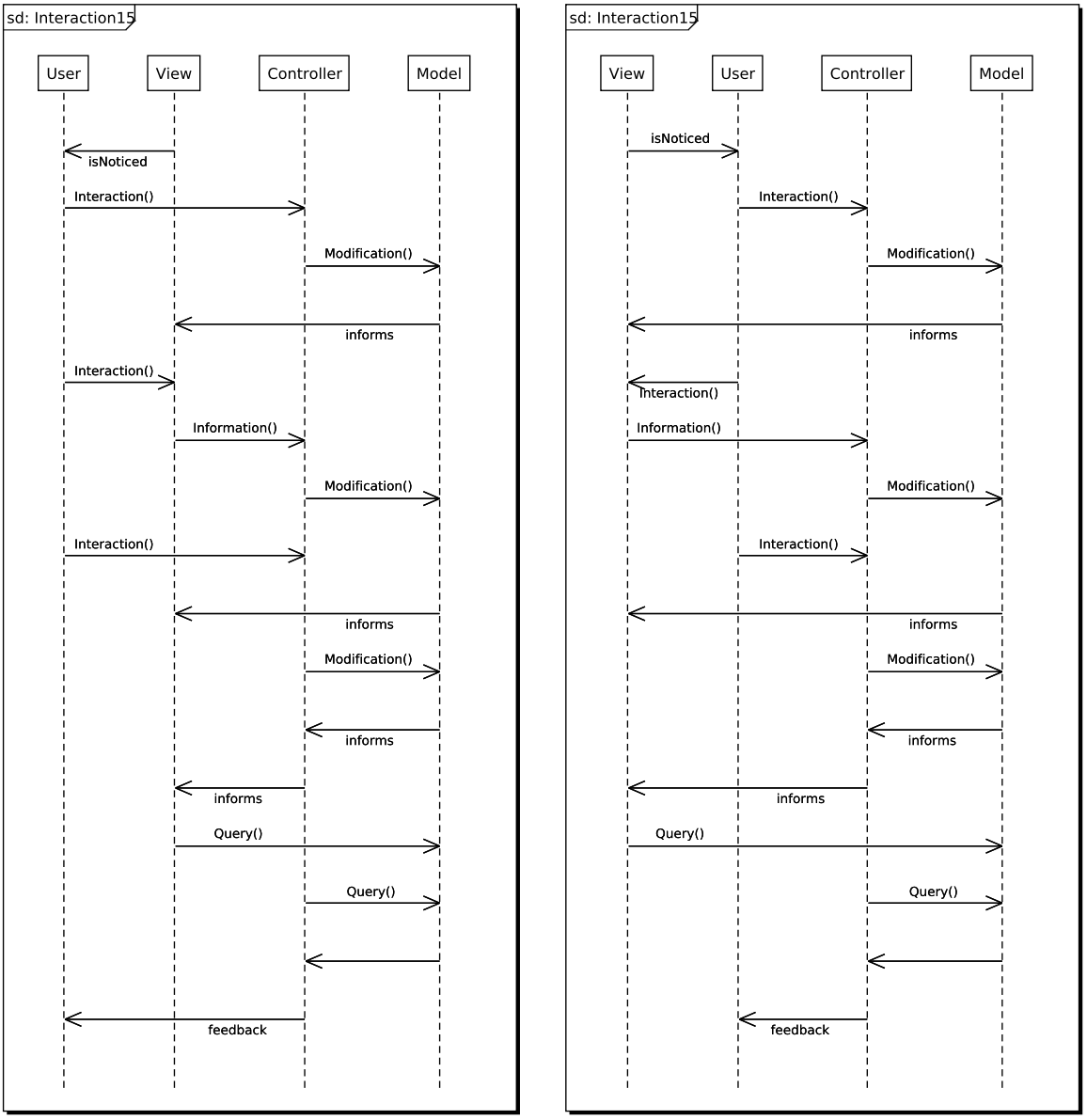
Figure A.10. The *Unfulfilled Orders* diagram.



(a) *Original* and *long message avoiding* lifeline sorting.

(b) *Layer-based* lifeline sorting.

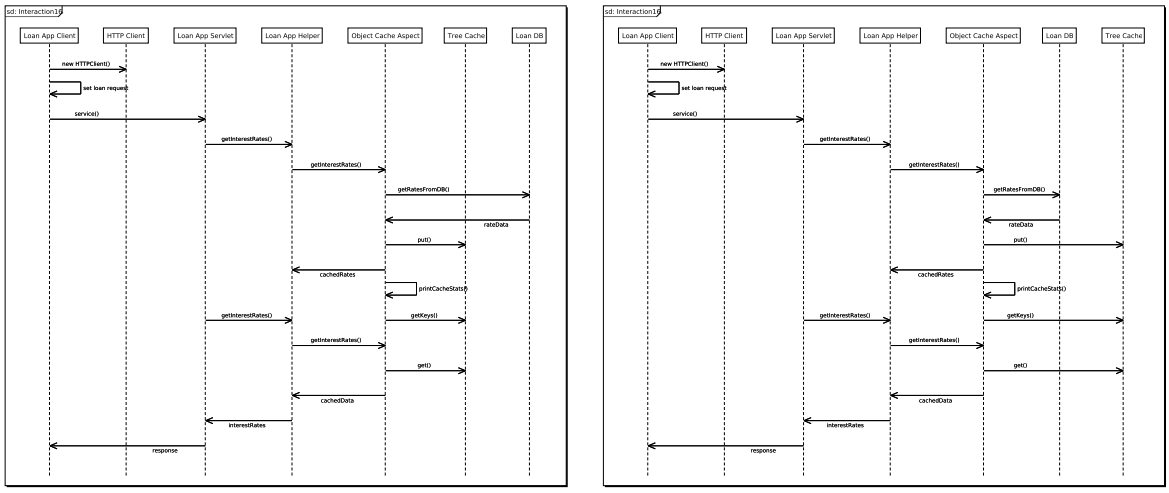
Figure A.11. The *Flirt* diagram.



(a) *Original* lifeline sorting.

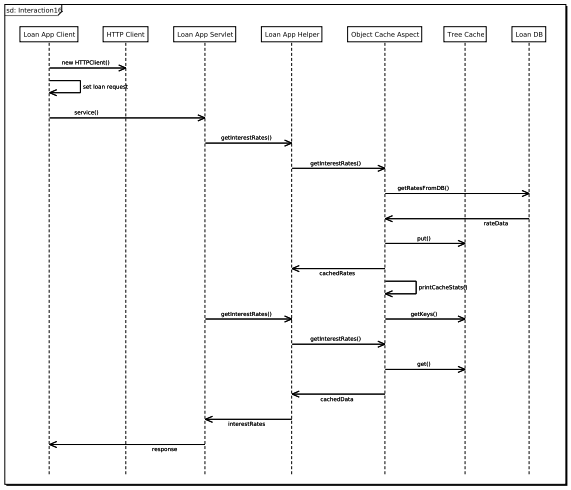
(b) *Layer-based* and *long message avoiding* lifeline sorting.

Figure A.12. The *Model View Controller* diagram.



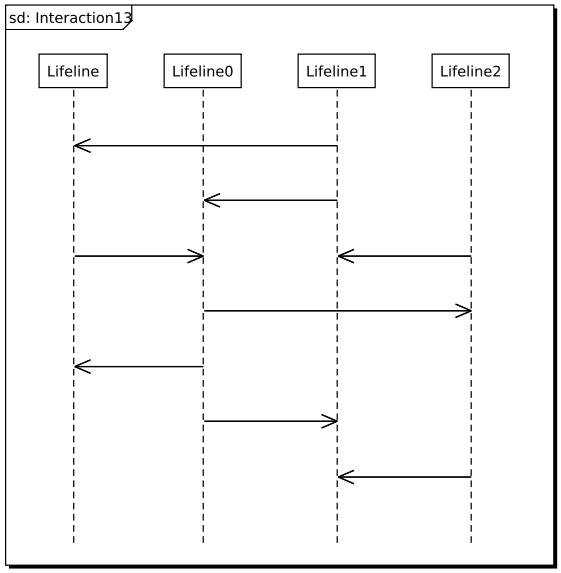
(a) *Original* lifeline sorting.

(b) *Layer-based* lifeline sorting.

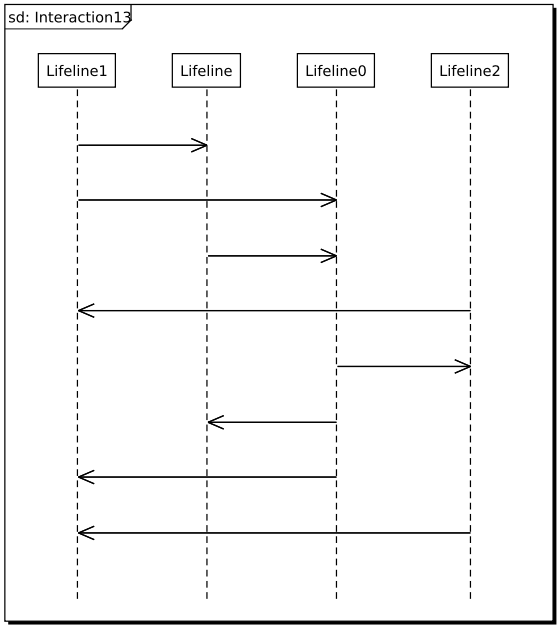


(c) *Long message avoiding* lifeline sorting.

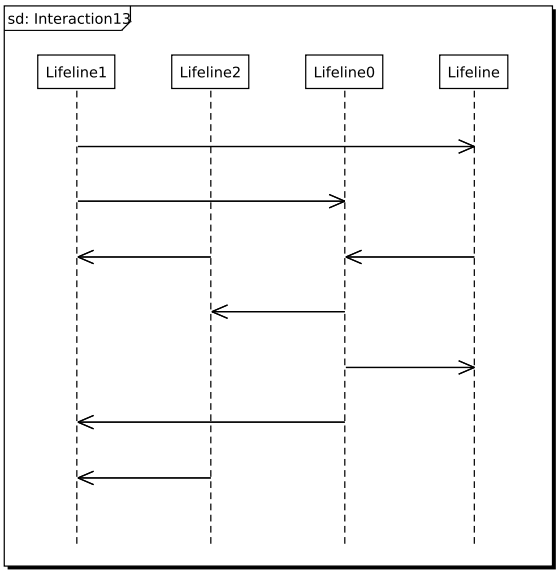
Figure A.13. The *Loan* diagram.



(a) *Original* lifeline sorting.

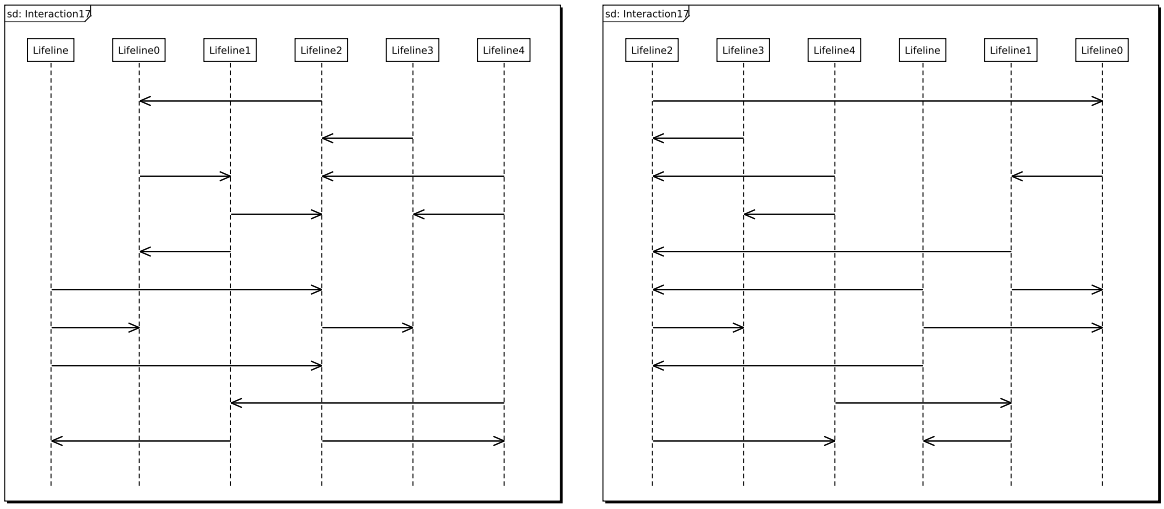


(b) *Layer-based* lifeline sorting.

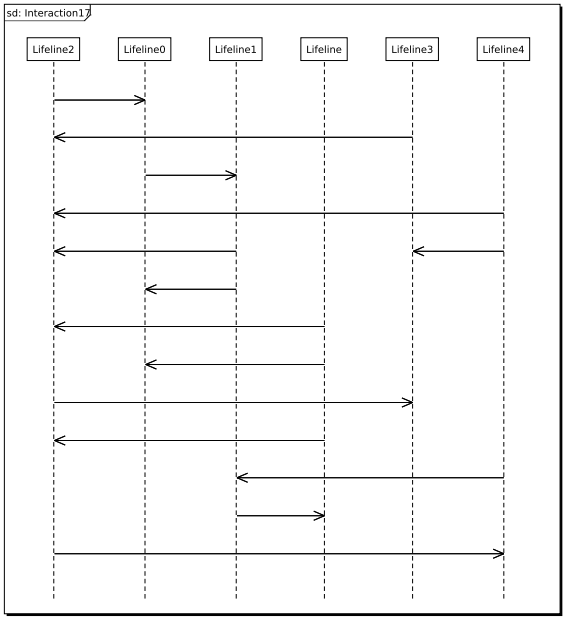


(c) *Long message avoiding* lifeline sorting.

Figure A.14. The *Random* diagram.

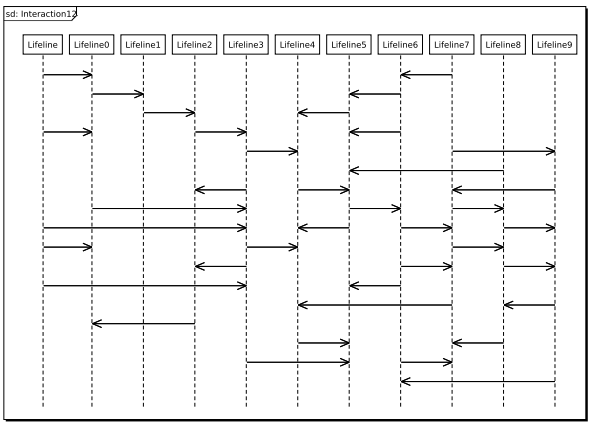


(a) *Original* lifeline sorting. **(b)** *Long message avoiding* lifeline sorting.

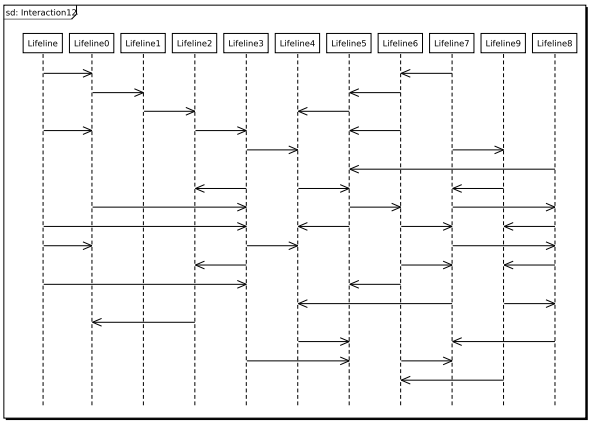


(c) *Layer-based* lifeline sorting.

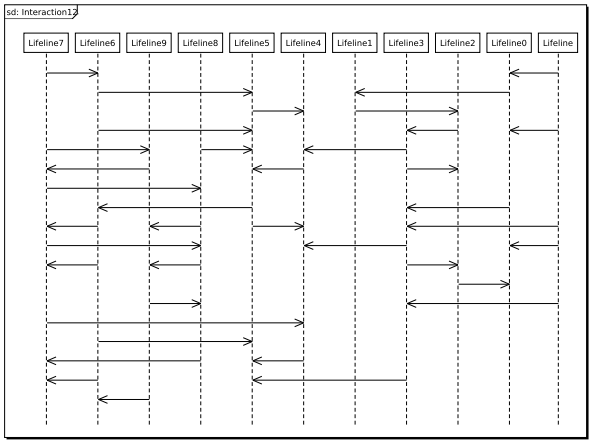
Figure A.15. The *Random II* diagram.



(a) *Original* lifeline sorting.



(b) *Layer-based* lifeline sorting.



(c) *Long message avoiding* lifeline sorting.

Figure A.16. The *Random Big* diagram.

Bibliography

[BDLN02] Carla Binucci, Walter Didimo, Giuseppe Liotta, and Maddalena Nonato. Computing labeled orthogonal drawings. In Michael T. Goodrich and Stephen G. Kobourov, editors, *Graph Drawing*, volume 2528 of *Lecture Notes in Computer Science*, pages 66–73. Springer Berlin Heidelberg, 2002.

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

[CGMW10] Markus Chimani, Carsten Gutwenger, Petra Mutzel, and Hoi-Ming Wong. Upward planarization layout. In *Proceedings of the 17th International Symposium on Graph Drawing (GD'09)*, volume 5849 of *LNCS*, pages 94–106. Springer, 2010.

[DETT94] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. Algorithms for drawing graphs: An annotated bibliography. *Computational Geometry: Theory and Applications*, 4:235–282, June 1994.

[DETT98] Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1998.

[Ecl08] Eclipse Software Foundation. Eclipse homepage, 2008. http://www.eclipse.org/.

[ESK04] Markus Eiglsperger, Martin Siebenhaller, and Michael Kaufmann. An efficient implementation of Sugiyama's algorithm for layered graph drawing. In János Pach, editor, *12th International Symposium on Graph Drawing (GD'04)*, volume 3383 of *LNCS*, pages 155–166. Springer-Verlag, 2004.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. Graph drawing by force-directed placement. *Software—Practice & Experience*, 21(11):1129–1164, 1991.

[FvH09] Hauke Fuhrmann and Reinhard von Hanxleden. The Kiel Integrated Environment for Layout for the Eclipse RichClientPlatform (KIELER) Homepage, 2009. http://www.informatik.uni-kiel.de/rtsys/kieler/.

[GJS76] Michael R. Garey, David S. Johnson, and L. Stockmeyer. Some simplified np-complete graph problems. *Theoretical Computer Science*, 1(3):237 – 267, 1976.

[GN98] Emden R. Gansner and Stephen C. North. Improved force-directed layouts. In SueH. Whitesides, editor, *Graph Drawing*, volume 1547 of *Lecture Notes in Computer Science*, pages 364–373. Springer Berlin Heidelberg, 1998.

[JM03] Michael Jünger and Petra Mutzel. *Graph Drawing Software*. Springer, October 2003.

[Kan96] Goos Kant. Drawing planar graphs using the canonical ordering. *Algorithmica*, 16(1):4–32, 1996.

[KD10] Lars Kristian Klauske and Christian Dziobek. Improving modeling usability: Automated layout generation for Simulink. In *Proceedings of the MathWorks Automotive Conference (MAC'10)*, 2010.

[KM98] Gunnar W. Klau and Petra Mutzel. Quasi-orthogonal drawing of planar graphs. Technical Report MPI-I-98-1-013, Max-Planck-Institut für Informatik, Saarbrücken, 1998.

[KM99] Gunnar W. Klau and Petra Mutzel. Optimal compaction of orthogonal grid drawings. In *Proceedings of the 7th International IPCO Conference on Integer Programming and Combinatorial Optimization*, volume 1610 of *LNCS*, pages 304–319. Springer-Verlag, 1999.

[KSSvH12] Lars Kristian Klauske, Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. Improved layout for data flow diagrams with port constraints. In *Proceedings of the 7th International Conference on the Theory and Application of Diagrams (DIAGRAMS'12)*, volume 7352 of *LNAI*, pages 65–79. Springer, 2012.

[KW01] Michael Kaufmann and Dorothea Wagner, editors. *Drawing Graphs: Methods and Models*. Number 2025 in LNCS. Springer-Verlag, Berlin, Germany, 2001.

[McA99] Andrew J. McAllister. A new heuristic algorithm for the linear arrangement problem. Technical report, University of New Brunswick, 1999.

[OSG08] OSGi Alliance. Homepage, 2008. `http://www.osgi.org/`, retrieved 2008-12-10.

[Pap76] Ch.H. Papadimitriou. The np-completeness of the bandwidth minimization problem. *Computing*, 16:263–270, 1976.

[PCJ96] Helen C. Purchase, Robert F. Cohen, and Murray James. Validating graph drawing aesthetics. In F. Brandenburg, editor, *Proceedings of Graph Drawing Symposium*, volume 1027 of *LNCS*, pages 435–446. Springer Verlag, 1996.

[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.

[Pur97] Helen C. Purchase. Which aesthetic has the greatest effect on human understanding? In *Proceedings of the 5th International Symposium on Graph Drawing (GD'97)*, volume 1353 of *LNCS*, pages 248–261. Springer, 1997.

[Pur98] Helen C. Purchase. The effects of graph layout. In *Computer Human Interaction Conference, 1998. Proceedings. 1998 Australasian*, pages 80 –86, nov-4 dec 1998.

[Pur02] Helen C. Purchase. Metrics for graph drawing aesthetics. *Journal of Visual Languages and Computing*, 13(5):501–516, 2002.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, February 1981.

[Tam87] Roberto Tamassia. On embedding a graph in the grid with the minimum number of bends. *SIAM Journal of Computing*, 16(3):421–444, 1987.

[TDB88] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. Automatic graph drawing and readability of diagrams. *IEEE Transactions on Systems, Man and Cybernetics*, 18(1):61–79, 1988.

[TT86] Roberto Tamassia and Ioannis G. Tollis. A unified approach to visibility representations of planar graphs. *Discrete and Computational Geometry*, 1(1):321–341, 1986.

[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)*, pages 317 – 326, may 2005.