INSTITUT FÜR INFORMATIK

Automatic Layout and Label Management for UML Sequence Diagrams

Christoph Daniel Schulze, Gregor Hoops,
and Reinhard von Hanxleden

Bericht Nr. 1804
July 2018
ISSN 2192-6247

CHRISTIAN-ALBRECHTS-UNIVERSITÄT

ZU KIEL

Institut für Informatik der
Christian-Albrechts-Universität zu Kiel
Olshausenstr. 40
D – 24098 Kiel

Automatic Layout and Label Management for UML Sequence Diagrams

Christoph Daniel Schulze, Gregor Hoops,
and Reinhard von Hanxleden

Bericht Nr. 1804
July 2018
ISSN 2192-6247

e-mail: E-mail: {cds,nbw,rvh}@informatik.uni-kiel.de

An abridged version of this work is published at the
IEEE Symposium on Visual Languages and Human-Centric Computing
(VL/HCC),
Lisbon, Portugal, October 2018.

Abstract

Sequence diagrams belong to the most commonly used types of UML diagrams. There is research on desirable aesthetics, but to our knowledge no published layout algorithms, although several have been developed. This might be due to the rigid specification of sequence diagrams that seems to make laying them out quite easy. However, as we argue here, naive algorithms do not always produce desirable solutions.

We present a layout algorithm that can compute the order of lifelines according to different optimization criteria. We also look at the problem of diagram size by introducing vertical compaction to sequence diagrams and by applying label management to compact them horizontally. We evaluate our methods with 50 real-world sequence diagrams.

1 Introduction

Sequence diagrams, a type of diagram defined by the Unified Modeling Language (UML), specify an *interaction* between a number of entities. At their most basic, sequence diagrams consist of *lifelines* that each represent an entity and are connected by arrows which represent the exchange of a message. The order of messages at each lifeline is meaningful: if a message a connects to a given lifeline above a message b, a temporally occurs before b.

In a study on the use of UML among 50 professional software developers [11], Petre found sequence diagrams to be among the top three most commonly used diagrams (along with activity diagrams and, of course, class diagrams), making them interesting objects to study. In this report, we will look at automatic layout of sequence diagrams, a topic which has received little research attention.

1.1 Contributions

We will present and evaluate an automatic layout algorithm for sequence diagrams which focusses on the following problems:

- *Height.* As the number of messages in an interaction increases, sequence diagrams grow taller. We present *vertical compaction* as a means to decrease their height by allowing messages to share y coordinates.
- *Width.* Not just the number of lifelines affects a diagram's width, the labels of messages do so as well. If we want to avoid impairing legibility due to lifelines that cross message labels, these need to fit into the space between lifelines, quite possibly pushing them apart. This problem is severe enough for companies to have developed guidelines for technical writers on how to draw sequence diagrams that fit into the available space [2]. We apply *label management*, first introduced in the context of another visual language [15], to sequence diagrams to improve this situation.
- *Lifeline order.* The order of lifelines affects the flow of communication through the diagram and the number of crossings between messages and lifelines. We build upon concepts introduced by Poranen et al. [13] to have the layout algorithm compute an order based on different criteria.

The scope of this paper is limited to what we believe are the most frequently used features of UML diagrams. Adding more features, however, would be straightforward.

1.2 Related Work

As opposed to class diagrams, sequence diagrams have received comparatively little research attention. A paper by Wong and Sun [17] on desirable aesthetics of class and sequence diagrams is a case in point: while the authors reference four papers just on the layout of class diagrams, sequence diagrams are represented only by two papers [2, 13] which do not even describe layout algorithms, but merely general aesthetics. One reason for this situation might be that compared to class diagrams, sequence diagrams offer rather less freedom when it comes to their layout.

In fact, layout algorithms have been developed, but we are not aware of any having been published. Bennett et al. [1] have implemented a sequence diagram viewer as part of a tool used in a study. They did not, however, describe their layout algorithm. There are several sequence diagram editors available that turn specifications based on a domain-specific languages into diagrams. Examples are *Quick Sequence Diagram Editor*,[1] *WebSequenceDiagrams*,[2] and *SequenceDiagram.org*.[3] Again, details on their layout algorithms have not been published, but experiments showed that none change the order of lifelines to optimize for any aesthetic criteria, or allow messages to share vertical coordinates to reduce the height of the diagram. Both are features of our layout algorithm that we describe in chapter 3.

Poranen et al. [13] describe and partly formalize aesthetic criteria they believe to be desirable for the layout of sequence diagrams. Wong and Sun [17] pick up on those criteria and justify them with principles from perceptual theories. We will introduce the relevant criteria in Sec. 2.1.1.

Label management, first proposed by Fuhrmann [4], has since been successfully integrated into another visual language based on node-link diagrams [15]. Here, we will integrate it into sequence diagrams, which exhibit other characteristics and thus make for a valuable additional case study.

1.3 Outline

This paper is structured as follows. We start by explaining the foundations of sequence diagrams and label management in chapter 2. We then describe our layout algorithm in chapter 3, followed by the integration of label management in chapter 4. After an evaluation in chapter 5, we conclude in chapter 6.

[1] http://sdedit.sourceforge.net/
[2] https://www.websequencediagrams.com/
[3] https://sequencediagram.org/

2 Foundations

This chapter will introduce the foundations necessary to understand the rest of this report: sequence diagrams (along with their aesthetics) and label management.

2.1 Sequence Diagrams

The UML distinguishes two types of diagrams [9]: *structure diagrams* and *behavior diagrams*. Sequence diagrams belong to a sub-group of the latter type called *interaction diagrams* that describe the communication between parts of a system, each focusing on different aspects. Sequence diagrams focus on the messages exchanged between communicating parties as part of an *interaction* between them. In a study performed by Petre [12], sequence diagrams were among the three types of UML diagrams most commonly used by professional software developers.

Figure 2.1 shows a simple sequence diagram while Figure 2.2 shows a more comprehensive overview of some of the elements available in sequence diagrams. The frame around it represents the interaction described by the diagram. Inside the frame is a series of vertical lines with individual headings called *lifelines*, one for each of the interaction's participants. The lifelines are connected by arrows that represent messages being exchanged; in the realm of object-oriented languages, these might for example be method calls. We say that a message that leaves or arrives at a lifeline is *incident* to that lifeline. Lifelines can be created and destroyed by messages, just like objects can be created and destroyed during the execution of, say, a Java program. Usually, whenever a lifeline is "busy" sending and receiving messages, this is interpreted as specifying a particular execution of whatever the lifeline represents, for example the execution of a method on an object. Such *execution specifications* are visualized by drawing a box along the lifeline starting at the first and ending at the last message involved. Execution specifications can be nested.

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

Sequence diagrams make use of text in several ways. First, every graphical element

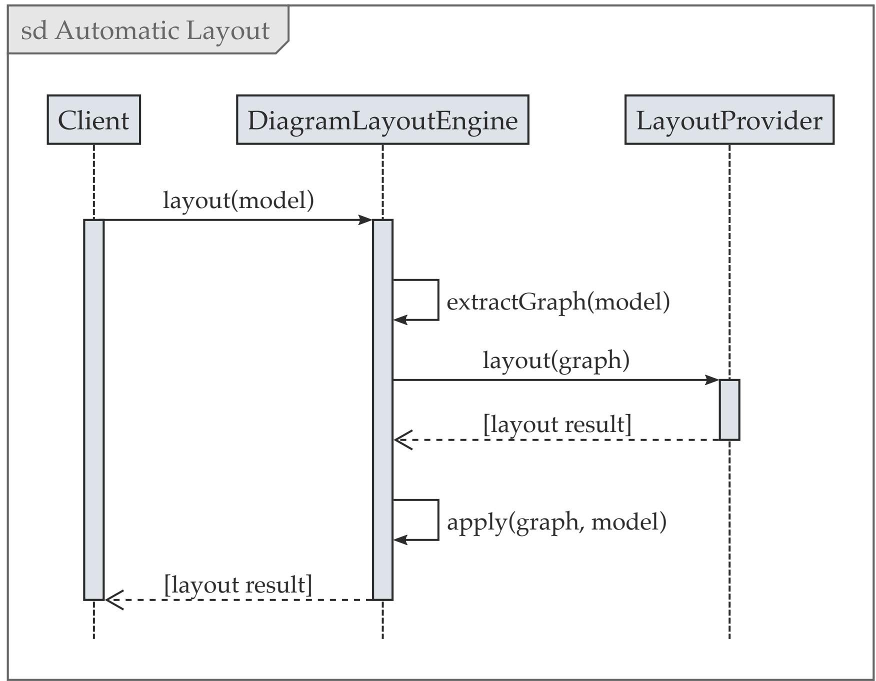


Figure 2.1: A simple example of a UML sequence diagram.

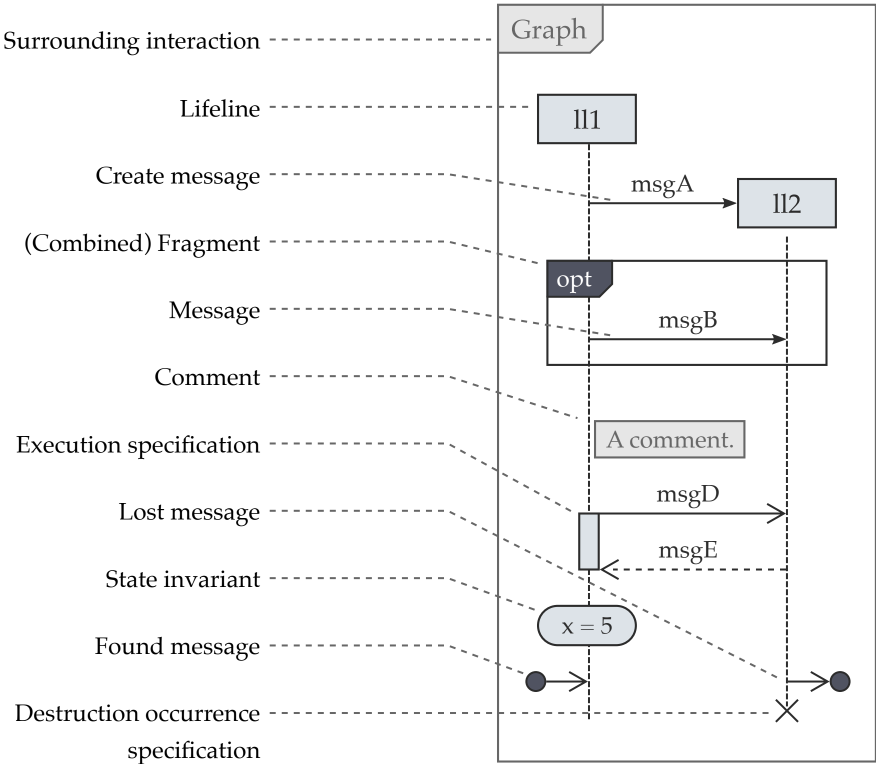


Figure 2.2: The different elements of a sequence diagram. The full specification includes yet more elements, but we constrain ourselves to this subset. Comments are not part of the official specification, but are included in virtually any sequence diagram editor.

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

Software for drawing sequence diagrams comes in two flavors: drag and drop editors (a popular example from the Eclipse world being the Papyrus[1] project), and textual editors based on custom Domain Specific Languages (DSLs) with a synthesized graphical view (WebSequenceDiagrams and SequenceDiagram.org for the browser, Quick Sequence Diagram Editor for the desktop).

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

To implement and test concepts proposed in this report, we have developed a textual editing environment that uses a new DSL called KIELER Sequence Diagram Language (KieSL), shown in Figure 2.3. The textual editor provides many features today's users would expect, such as syntax highlighting and content assist. In addition, the automatically updated graphical view can serve as a navigation aid in that clicking on an element reveals its definition in the text. The editing environment is available as part of the Open KIELER project.[2]

2.1.1 Aesthetics

Since sequence diagrams look entirely different than standard node-link diagrams, commonly accepted aesthetic criteria differ in applicability, as noted by Poranen et al. [13]. Edge crossings, for instance, should not be regarded as being just that, crossings between two edges, but as crossings between messages and lifelines. As per the UML standard [9], edges are always straight and are usually drawn horizontally. The length of an edge is closely related to the number of lifelines it crosses. A sequence diagram's size and aspect ratio are mostly functions of its structure, but its height in particular can be variable, as we shall see in chapter 3 (in this point we disagree with Poranen et al., who consider the size of a sequence diagram to follow entirely from its structure).

It is not just that existing aesthetic criteria must be adapted to sequence diagrams or do not apply in the first place, it is also that additional criteria may be called for.

[1] https://eclipse.org/papyrus

[2] https://github.com/OpenKieler

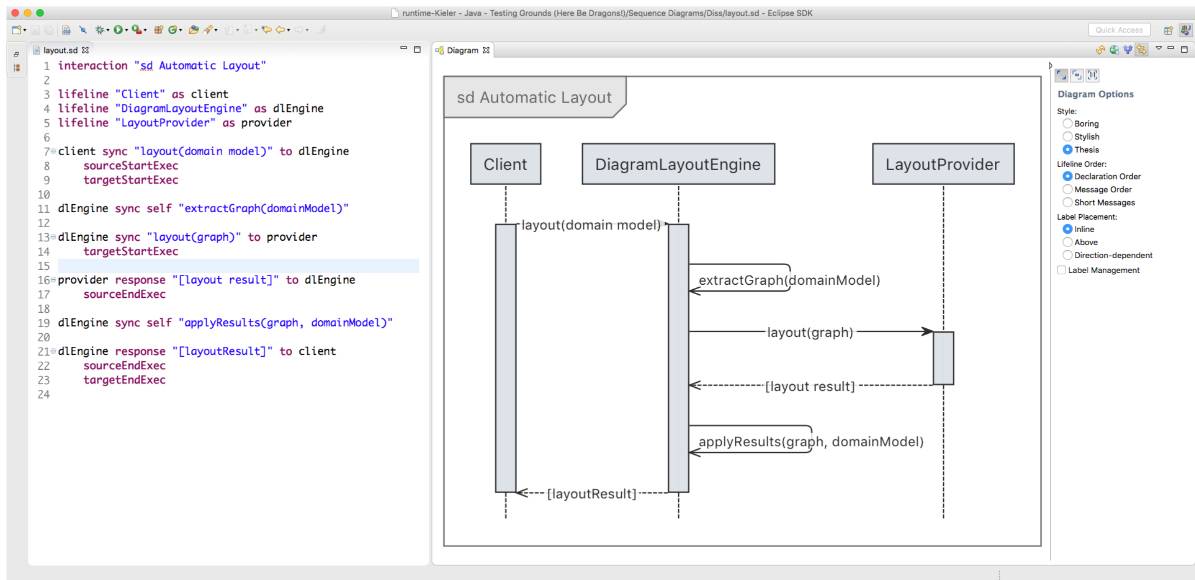


Figure 2.3: The sequence diagram from Figure 2.1 being edited in our sequence diagram editing environment. Sequence diagrams are defined through the textual KieSL language in the text editor to the left. Right next to it is the graphical view that shows the corresponding diagram, with the sidebar to the right allowing users to switch between different display options.

Poranen et al. introduce three such criteria: *subset separation*, the *number of long edges*, and *slidability* (which they call *sliding*).

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

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

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

One last, but important aesthetic is one which Poranen et al. even elevate to a constraint each sequence diagram must satisfy: the *starting object* criterion, which requires that the lifeline which initiates the interaction be the leftmost lifeline in the sequence diagram.

2.1.2 Label Management

Too much text in a diagram has two effects: first, it increases the width of a diagram to a point where, if it is supposed to be drawn on screen in its entirety, it needs to be scaled down too much to be readable; and second, it puts much information on screen that may not actually be relevant to the viewer at a given moment. Label management [15] solves this through *label management strategies* that take the original text and shorten or wrap it, optionally taking a desired *target width* into account. The latter can be provided by automatic layout algorithms since they know how long a label can be before it starts affecting the size of the diagram by pushing other elements apart.

Label management ties into the well-known *model-view-controller paradigm* [14] in that it is a solution that only affects the view, not the underlying model. Regarding the decision of whether to shorten a label or to leave it as is, *focus and context* is an important concept [3]. Herein, the diagram elements are divided into the set of elements the user is currently focussing on (which should probably be left untouched) and surrounding elements that provide context (which may well have their details reduced).

3 Laying Out Sequence Diagrams

Due to the rigorous visual structure of sequence diagrams it may seem that they do not make for a particularly interesting layout problem. After all, the order of messages is fixed within each lifeline, and the lifelines themselves are simply placed next to each other along a horizontal line. However, the order of lifelines is free and determines the number of crossings between messages and lifelines. Also, the exact coordinates of message end points, while constrained, is not fixed and can impact readability. Our layout algorithm aims to capitalize on these degrees of freedom.

In accordance with the *vertical distance* constraint defined by Poranen et al. [13], we divide the diagram into horizontal *communication lines*, a configurable amount of space apart, and restrict messages to run along these lines only. We allow messages to share a communication line, in contrast to other sequence diagram tools—at least ones based on automatic layout—which usually assign each message to its own communication line, ordered by their appearance in the original diagram (or a textual description thereof). This of course results in higher drawings, whereas allowing messages to share communication lines gives us more freedom to optimize the diagram's height. This is what we refer to as *vertical compaction*, which we allow to be switched on or off.

Our algorithm is structured into five phases, each with clearly defined responsibilities:

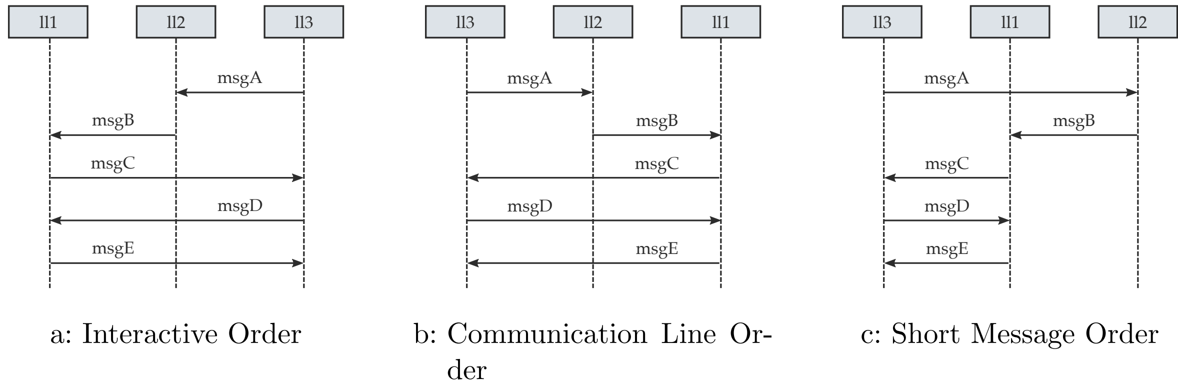
1. Lifeline ordering
2. Space allocation
3. Cycle Breaking
4. Communication line assignment
5. Coordinate assignment

The following sections will describe each phase.

3.1 Phase 1: Lifeline Ordering

The algorithm starts out by tackling what it has the most control over: determining the order of lifelines. It provides three different ordering strategies: *interactive order*, *communication line order*, and *short message order*. The results produced for an example diagram are shown in Figure 3.1.

The latter two strategies require further information about the vertical order of messages, which is why the first phase begins by calculating an *element ordering graph*.



a: Interactive Order

b: Communication Line Order

c: Short Message Order

Figure 3.1: The results produced by different lifeline ordering strategies. The interactive order corresponds to the original order as fed to the layout algorithm.

Therein, each message is represented by a node and an edge runs from node x to node y if the following conditions are met:

1. There is a lifeline both messages are incident to.

2. The message represented by x immediately precedes the messages represented by y on that lifeline.

We call this an *element ordering constraint*, and it must be adhered to in the final layout by assigning the involved messages to different communication lines.

If vertical compaction is disabled, further constraints are added to the graph to ensure that each message is assigned its own communication line. We order the messages by their vertical coordinates and introduce a constraint between each pair of adjacent messages, unless one already exists. Note that the coordinates are used only to establish an order among the messages and thus need not be based on any kind of existing drawing of the sequence diagram. For a diagram based on a textual language, it may for example be enough to simply use the line number a message was defined in as its vertical coordinate.

3.1.1 Interactive Order

The simplest of the three ordering strategies, this one simply adopts the lifeline order from an input graph.

3.1.2 Communication Line Order

The idea of this strategy is to order lifelines in a way that makes following the communication through the diagram as easy as possible. The main loop in Figure 3.2 runs until all lifelines are placed. Each iteration starts with a call to the `findUppermostMessage(...)` function (line 1). Among the outgoing messages of all unprocessed lifelines, it computes

Input: *lifelines*, the set of lifelines to be ordered
Output: Ordered list of lifelines

result ← empty list

while *lifelines* $\neq \emptyset$ **do**
    msg ← findUppermostMessage(*lifelines*)
    if *msg* is undefined **then**
        Add *lifelines* to *result* in arbitrary order
        lifelines ← $\emptyset$
    else
        Remove source(*msg*) from *lifelines*
        Add source(*msg*) to *result*
        repeat
            if target(*msg*) $\in$ *lifelines* **then**
                Remove target(*msg*) from *lifelines*
                Add target(*msg*) to *result*
            end
            msg ← findUppermostOutgoingMessage(target(*msg*), *lifelines*)
        until *msg* is undefined
    end
end

return *result*

Figure 3.2: The communication line lifeline ordering strategy.

the subset of messages whose corresponding nodes have no predecessor in the element ordering graph. Of this subset it returns the message whose source lifeline has the biggest difference of outgoing message count minus incoming message count, the idea being to have lifelines with lots of outgoing messages move to the left of the diagram. If there is no outgoing message left among the unprocessed lifelines, we add the remaining lifelines to our result in an arbitrary order (lines 1 to 1).

If we did find an uppermost message, we add its source lifeline to the end of our list (lines 1 and 1). If the message's target lifeline has not been placed yet, we do so (lines 1 to 1) and then proceed to find the uppermost message that connects it to a lifeline that has not been placed yet by calling the `findUppermostOutgoingMessage(...)` function (line 1).

To take the diagram in Fig. 3.1a as an example, we would start with all three lifelines unprocessed. The algorithm would find that *msgA* is the uppermost message and would thus add *ll3* to the as yet empty list of lifelines. It would then proceed to add the message's target, *ll2*, to the list as well and look for its uppermost outgoing message that is headed for an unprocessed lifeline. It finds *msgB* and adds *ll1* to the list, at which point the result is the one shown in Fig. 3.1b.

The results produced by this algorithm adhere to the starting object constraint which

requires the lifeline that initiates the interaction to be the leftmost one. Their criterion of slidability should not be confused with our goal here. While layouts with good slidability lend themselves well to being browsed through from top to bottom with only occasional horizontal shifts, the layouts produced by our algorithm will generally tend to be browsed through from left to right.

3.1.3 Short Message Order

None of the preceding algorithms minimizes the length of messages or the crossings they cause. The culprits in our example diagram are the three bottommost messages, which in both layouts so far span the whole diagram. Besides the message length itself, longer messages quite obviously also produce more crossings with lifelines [13], which may harm readability. The third and final algorithm thus aims to optimize for message length. Doing so, however, turns out to be a hard problem.

Let L be a set of $n \in \mathbb{N}$ lifelines and let $m(l_1, l_2)$ be the number of messages from l_1 to l_2 or vice versa for $l_1 \neq l_2 \in L$ (zero for $l_1 = l_2$). What we are looking for in our quest to order the lifelines is a bijective assignment $\mathcal{L} \colon L \to \{1, \dots, n\}$. The length of a message that connects l_1 and l_2 is exactly $|\mathcal{L}(l_1) - \mathcal{L}(l_2)|$, and the number of crossings it produces is $|\mathcal{L}(l_1) - \mathcal{L}(l_2)| - 1$.

Poranen et al. list two different goals one might want to optimize for [13]:

- The first would be to minimize the weighted sum of the length of all messages:

$$\sum_{\{l_1, l_2\} \subseteq L} m(l_1, l_2) \cdot |\mathcal{L}(l_1) - \mathcal{L}(l_2)|.$$

  This is a weighted version of the *linear arrangement problem*, which is NP-complete already in the unweighted case [6].

- The second would be to minimize the length of the longest message. This is equivalent to the *bandwidth minimization problem*, shown to be NP-complete by Papadimitriou [10].

We accept a few long edges to gain mostly short edges and thus optimize for the first goal. Our implementation is based on an algorithm proposed by McAllister [8], which, like many algorithms for bandwidth minimization, is split into two phases:

1. Selecting a start lifeline. This will end up being the leftmost lifeline in the diagram.

2. Iteratively selecting lifelines for the remaining $n - 1$ slots.

We make two adjustments to McAllister's algorithm to adapt it to the requirements of sequence diagram layout. First, we select the start lifeline as we did with the communication line order algorithm to have the leftmost lifeline be the one that starts the interaction. And second, we introduce an option to increase the weight of messages that are contained in combined fragments. The algorithm then tries harder to keep such messages short, resulting in smaller fragments.

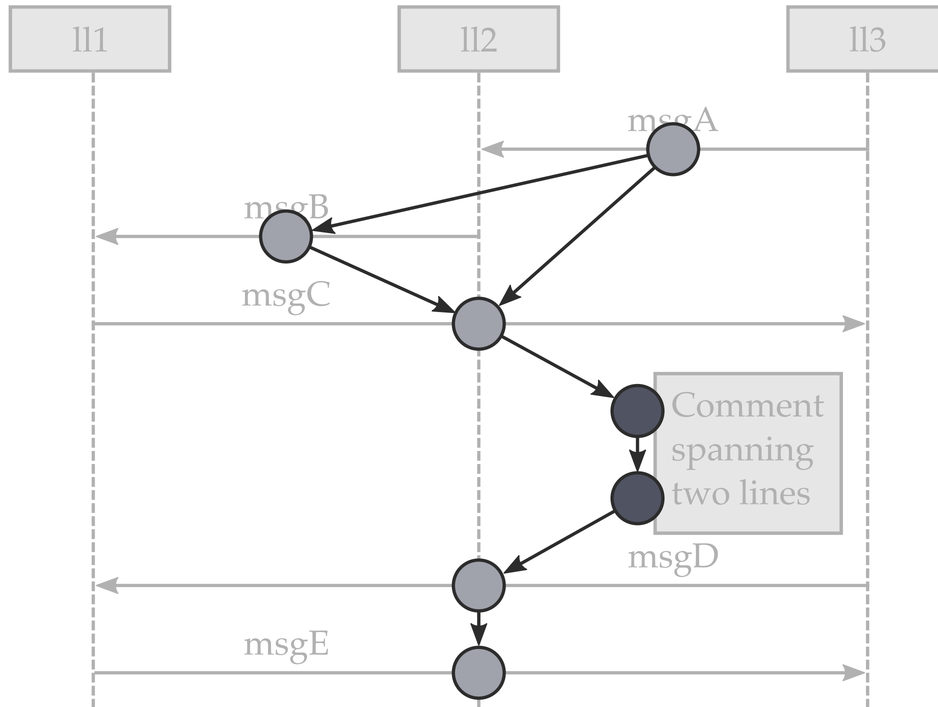


Figure 3.3: A diagram with the element ordering graph produced by the algorithm's second phase. Note how the successor constraint from *msgC* to *msgD* was broken to reserve space for the comment.

3.2 Phase 2: Space Allocation

The algorithm will soon use the element ordering graph to assign messages to communication lines such that no two messages with direct or transitive ordering constraints end up on the same communication line. To comply with the vertical distance constraint, communication lines will later be placed a uniform distance apart. To ensure that diagram elements that are not messages will have enough space available for their placement, Phase 2 inserts nodes into the element ordering graph to reserve space for them. The set of affected elements includes comments as well as headers of combined fragments.

Figure 3.3 shows a diagram with its element ordering graph overlaid. Note that the comment is so tall that it requires two communication lines of space, and is thus represented by two nodes in the graph.

Unless vertical compaction is switched off, there is yet another problem to be solved. Consider the sequence diagram in Figure 3.4. Without further provisions the layout algorithm may end up moving *msgD* into the combined fragment, since the element ordering graph does not prevent it from doing so.

Before we solve this problem we define two kinds of nodes in the element ordering graph. The *lowermost nodes* of a fragment are nodes that represent messages of the fragment that have no successors in the ordering graph that represent messages of the

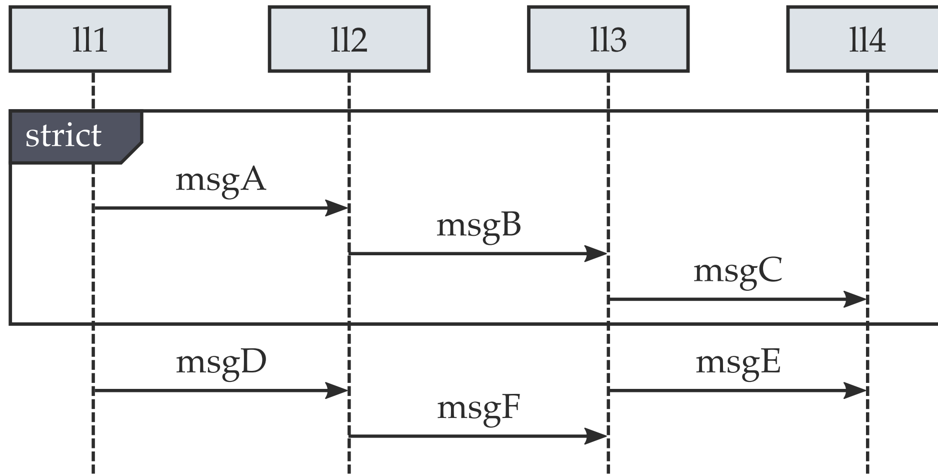


Figure 3.4: Without further precautions, the ordering constraints placed on *msgD* would allow it to creep into the combined fragment, sharing a communication line with *msgC*.

same fragment. In the example, the node that represents *msgC* is a lowermost node. Note that however large the fragment becomes, one of its lowermost nodes will always mark its bottom boundary.

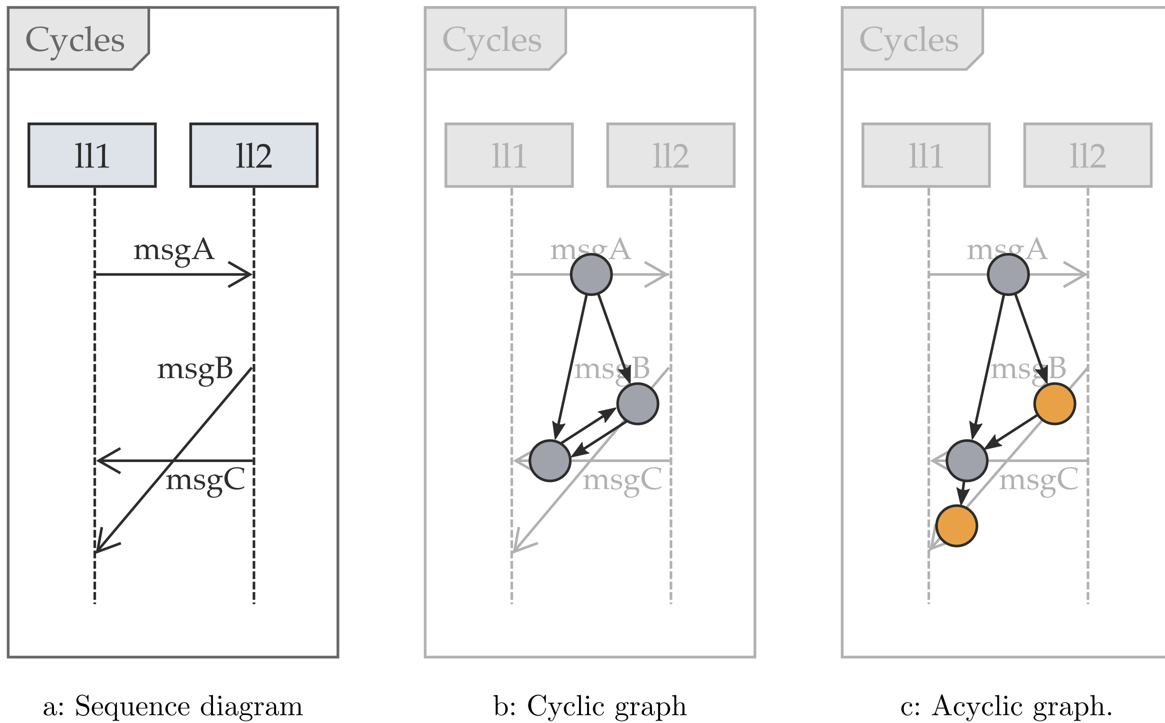
The second concept we need is that of *fragment successor nodes*: nodes that represent messages not part of a given fragment, but that have at least one predecessor which is. In the example, both *msgD* and *msgE* are fragment successor nodes, but *msgF* is not.

To solve our original problem, we introduce additional ordering constraints from all lowermost nodes to all fragment successor nodes. In a way, we force the algorithm to keep the messages that follow the fragment from creeping past those messages that will later determine the fragment's lower boundary.

Similar problems arise above a fragment. First, the *fragment predecessor nodes* must be kept from creeping past the fragment's upper boundary. And second, the fragment's *uppermost nodes* must not enter the fragment's header area. Both problems are solved by introducing additional successor constraints from the predecessor nodes to the dummy node that represents the fragment's header area, and from said dummy node to the fragment's uppermost nodes.

3.2.1 Phase 3: Cycle Breaking

The element ordering graph created by the preceding phase captures message ordering constraints that need to be met once messages are assigned to communication lines. For this to be possible, the graph needs to be acyclic. At first, this does not seem to be a problem, but there are valid sequence diagrams that violate this requirement, such as the one in Figure 3.5a, which results in the element ordering graph shown in Figure 3.5b. While it certainly seems debatable whether such a diagram makes sense, the fact is that such states can arise while users are in the process of building a sequence diagram, and thus need to be supported by the layout algorithm.



a: Sequence diagram

b: Cyclic graph

c: Acyclic graph.

Figure 3.5: The algorithm needs to be able to handle diagrams with cyclic element ordering graphs. **(a)** A sequence diagram with non-horizontal messages, inspired by the UML specification [9, Figure 17.3]. **(b)** Without processing, *msgB* causes the graph to be cyclic. **(c)** The node that represents *msgB* is split into two nodes to remove the cycle, each representing one of the message's end points.

To break a cycle, ELK Sequence splits one of its nodes, as shown in Figure 3.5c. Since most messages will end up being drawn horizontally, each node in the element ordering graph usually represents both the start and the end point of its corresponding message. Its incident edges consequently represent ordering constraints involving either the source or the target lifeline. This is the cause for cycles in the first place. Splitting a node makes the ordering constraints independent from one another, making it impossible for cycles to appear.

3.3 Phase 4: Communication Line Assignment

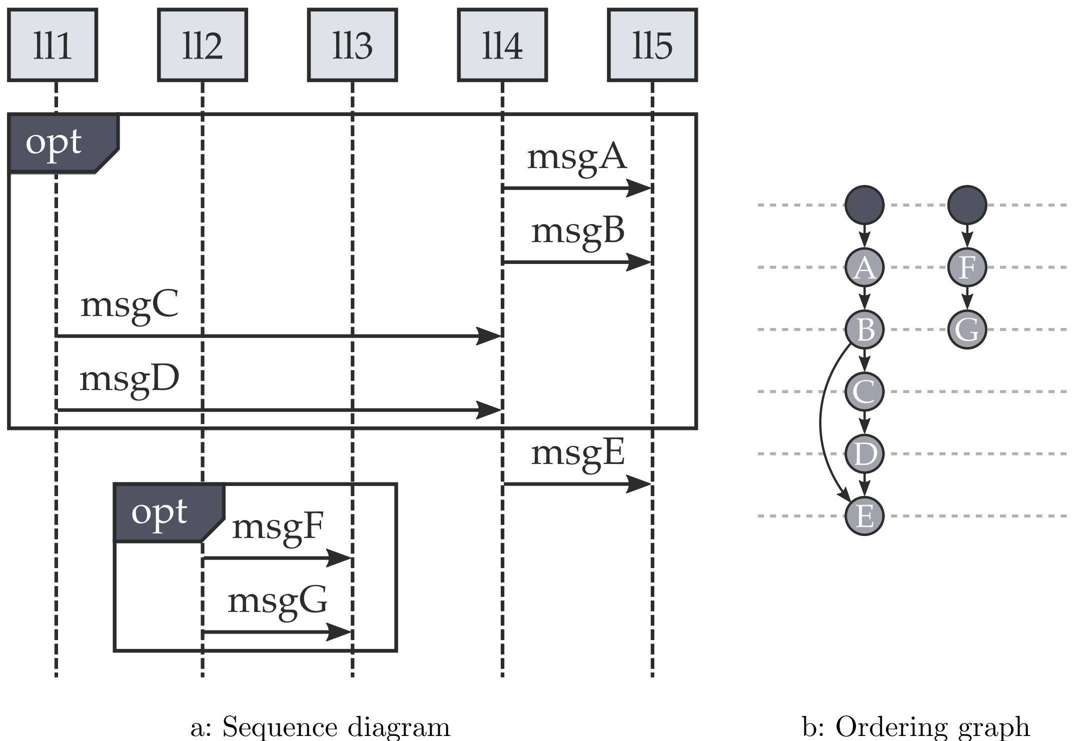
With the first three phases completed, we now have an acyclic element ordering graph that we can use to assign the graph's nodes to communication lines. This problem is equivalent to the layer assignment problem of the well-known layered approach by Sugiyama et al. [16]. To solve it, we use the network simplex algorithm by Gansner et al. [5] to produce results that tend to keep messages close together on each lifeline.

Similar to Phase 2, the fourth phase also has to solve an additional problem if vertical compaction is switched on. Phase 2 introduced constraints that keep messages from entering fragments off limits to them, but it was only able to catch those messages that it knew for a fact would have to precede or follow a fragment. It could not, however, prevent completely unrelated messages from wreaking havoc with the diagram's semantics.

Consider, for example, the sequence diagram in Fig. 3.6a. It has five lifelines of which *ll2* and *ll3* have no relation at all to the others, in particular not to anything contained in the combined fragment. Fig. 3.6b shows the original assignment of the element ordering graph's nodes to communication lines as produced by Phase 4, including the nodes added for fragment headers (darker nodes). Since Phase 2 completely missed the fact that *msgF* and *msgG* must not move into the big fragment, Phase 4 produces an assignment that would lead to an overlap of the two fragments unless we take additional steps to prevent that from happening.

Even if Phase 2 had noticed the problem, it could not have determined which additional constraints to add: should *msgF* and *msgG* be placed above or below the larger fragment? Such decisions can only be made once messages have been assigned to communication lines and possible overlaps can be detected; in other words, it becomes the responsibility of Phase 4.

We deal with this problem by post-processing the communication line assignment as follows. For each combined fragment, we compute the lifelines it spans by looking for the leftmost and for the rightmost lifeline incident to one of the fragment's messages. The big fragment in Fig. 3.6a, for example, spans all five lifelines whereas the smaller fragment only spans two. We then iterate over all communication lines and keep a list of fragments for each lifeline that are currently open (or *active*) there. For each communication line, we perform four steps of computation.



a: Sequence diagram

b: Ordering graph

Figure 3.6: Correct drawing of a sequence diagram with five lifelines, of which two (*ll2* and *ll3*) bear no relation to the others. Its element graph originally had both areas assigned to the same communication lines since there are no ordering constraints between the involved nodes.

3.3.1 Step 1

We iterate over the communication line's nodes, looking for nodes that represent the header of a combined fragment. Let x be such a node and f_x be the fragment it represents. The fragment can begin at the current communication line if there is no other fragment f_y for which all of the following conditions are true:

- f_y has already been marked as being active.
- The sets of lifelines spanned by f_x and f_y have a non-empty intersection.
- f_x is not contained in f_y and f_y is not contained in f_x (otherwise it would be perfectly fine for them to overlap).

If we find no such fragment f_y, f_x can begin at the current communication line and is thus marked as being active at all lifelines it spans.

Taking the ordering graph from Fig. 3.6b as an example, we would first encounter the node that represents the header of the large fragment and mark that as active at all lifelines. For the node that represents the smaller fragment, we would detect a conflict with the larger fragment at lifelines $ll2$ and $ll3$ and thus refrain from activating it.

3.3.2 Step 2

We iterate over the current communication line's nodes again, building an initially empty list of nodes that will have to be moved to the next communication line because they are in conflict with active fragments. If a node represents the header of a fragment that has not been marked active by Step 1, it is added to our list. If a node represents a message that would cross an active fragment it is not part of, that too is added to our list.

In our example graph, the smaller fragment's header node would be the only one added to the list while processing the first communication line.

3.3.3 Step 3

Each node in the list computed by the previous step is moved to the next communication line. If it has successors in the element ordering graph, that may of course invalide the communication line assignment by placing two nodes with ordering constraints on the same communication line. We thus allow the movement to propagate through the following communication lines to restore the assignment's validity.

In our example, we would move the smaller fragment's header node to the second communication line. Since it now shares a communication line with the node representing $msgF$, we would move that to the third line, and move $msgG$ to the fourth line for a similar reason.

3.3.4 Step 4

Finally, we look for active fragments that need to be deactivated. A fragment can cease to be active once we have encountered all of its bottommost nodes.

In the example, the first time this happens is when we encounter the node that represents *msgD* on communication line five. This is when we mark the larger fragment as not being active anymore, thereby allowing the smaller fragment to become active once we process the next communication line.

3.4 Phase 5: Coordinate Assignment

With a lifeline order having been decided upon and communication lines having been computed, the final phase computes actual coordinates for all elements of the sequence diagram. This is rather straightforward. The bulk of the coordinate assignment process can be divided into two stages: the *vertical sweep* and the *horizontal sweep*.

The vertical sweep iterates over the communication lines computed during the preceding phase and is only interested in those of the nodes of the element ordering graph that represent messages. At first glance it seems that we can simply assign all of the line's messages a y coordinate that is a configurable amount of space below the preceding line's messages. In doing so, however, we might end up with overlaps between messages with a non-empty intersection of the sets of lifelines they span. The solution of course is to introduce the possibility for a communication line to split into several lines to avoid message overlaps, but trying to get away with as few lines as possible. We use a simple greedy algorithm that iterates over the communication line's messages and splits off a new line if a message causes overlaps in all existing lines.

The horizontal sweep primarily iterates over the lifelines and assigns x positions to them. This is a straightforward process, yet one needs to be sure to reserve enough space for lost and found messages, execution specifications, message labels, and comments attached to messages. The x positions of the lifelines can not always be directly applied to their incident messages due to the possible presence of execution specifications. After placing these, x coordinates of incident messages need to be adjusted.

At this point we know how far to the right the lifelines extend. Any comments not explicitly attached to diagram elements can now be placed in a column to the right of the rightmost lifeline. Also, coordinates of combined fragments can be set according to the messages they contain.

4 Integrating Label Management

Sequence diagrams, prone to becoming large rather quickly [2], are a good candidate for label management. They use labels at several points, which we will go through in order to discuss if and how we apply label management to them.

First up is the title of a sequence diagram's interaction. This will usually be much shorter than the interaction's content is wide and does not contribute a lot to the diagram's clutter. We thus exempt it from label management.

More relevant are the titles of lifelines. We believe them to be important enough, though, for them not to be shortened. We thus exempt lifeline titles from label management as well.

Message labels are a lot more interesting. If they are placed between the message's source lifeline and one of its adjacent lifelines, they influence the amount of space between them, quickly pushing them apart and enlarging the diagram in the process. This is a prime candidate for label management.

We allow users to switch between two label management strategies. The first is a label management strategy which removes the arguments of method calls (*semantical abbreviation*). While this seems promising for sequence diagrams that model interactions at this level of abstraction, it will not work as well for diagrams which model, say, communication protocols. The second strategy thus simply cuts off the label text once it reaches the minimum amount of space available to it at the lifelines it is placed between (*syntactical abbreviation*), adding an ellipsis to indicate to users that this has happened. The amount of space is calculated by the horizontal sweep of the algorithm's fourth phase and supplied to label management as the target width.

Just like messages, comments can also push lifelines apart and thereby enlarge sequence diagrams. Unlike messages, they can even contain several lines of text. We thus start by removing all but the first five words, followed by syntactical abbreviation to ensure that what remains does not enlarge the diagram.

Besides offering tool tips that provide access to the original text, we can also make good use of focus and context by only applying label management to those elements that are part of the context and displaying focussed elements in full detail. If the user selects a comment, only that is focussed. If they select a message, its label and any associated comments are focussed. If they select a lifeline, any incident messages and associated comments are focussed.

This behavior led to an interesting discovery. Imagine that the labels of the comment and the messages in the sequence diagram shown back in Figure 2.1 were much longer than they are in that diagram, so much so that they push the two lifelines apart. Suppose further that label management is engaged, which promptly pulls the lifelines closer together again by shortening the text of all messages and comments. Finally, suppose

that the user selects a message. The message's original text is restored, pushing the lifelines apart again. While this does not come as a surprise, what can come as one is what happens to the comment. It is not part of the focus, but the additional space available between the lifelines may allow label management to show more of the comment's text as well.

When this happened during a presentation, someone in the audience commented that they found this behavior confusing. In this case, confusion might be reduced by always shortening a comment which is not focussed to its first few words, without the additional syntactical abbreviation step.

5 Evaluation

We evaluated the layout algorithm and our label management implementation with two aesthetics-based experiments. To do so, we used 50 sequence diagrams published on GitHub which we found among the first $5,000$ items of a list of real-world UML models published by Helbig et al. [7].[1] The sequence diagrams averaged 6.06 lifelines (for a total of 303) and 16.54 messages (827). The collected data and any scripts used to conduct the subsequent analysis are available online.[2]

5.1 Layout Algorithm

We wanted to answer the following questions:

1. Which effects do the lifeline ordering strategies have on the length of messages and their crossings with lifelines?
2. Which effect does vertical compaction have on the height of sequence diagrams?

5.1.1 Lifeline Ordering Strategies

Figure 5.1a shows the mean message length in each diagram produced by the three lifeline ordering strategies. The results look rather similar, so Figure 5.1b shows the effect of the non-interactive strategies on message length compared to the original ordering as determined by the diagram's designer. As expected, short message order will usually decrease the mean edge length in a diagram, which is true for 50% of all diagrams in our test set, as Table 5.1 shows.

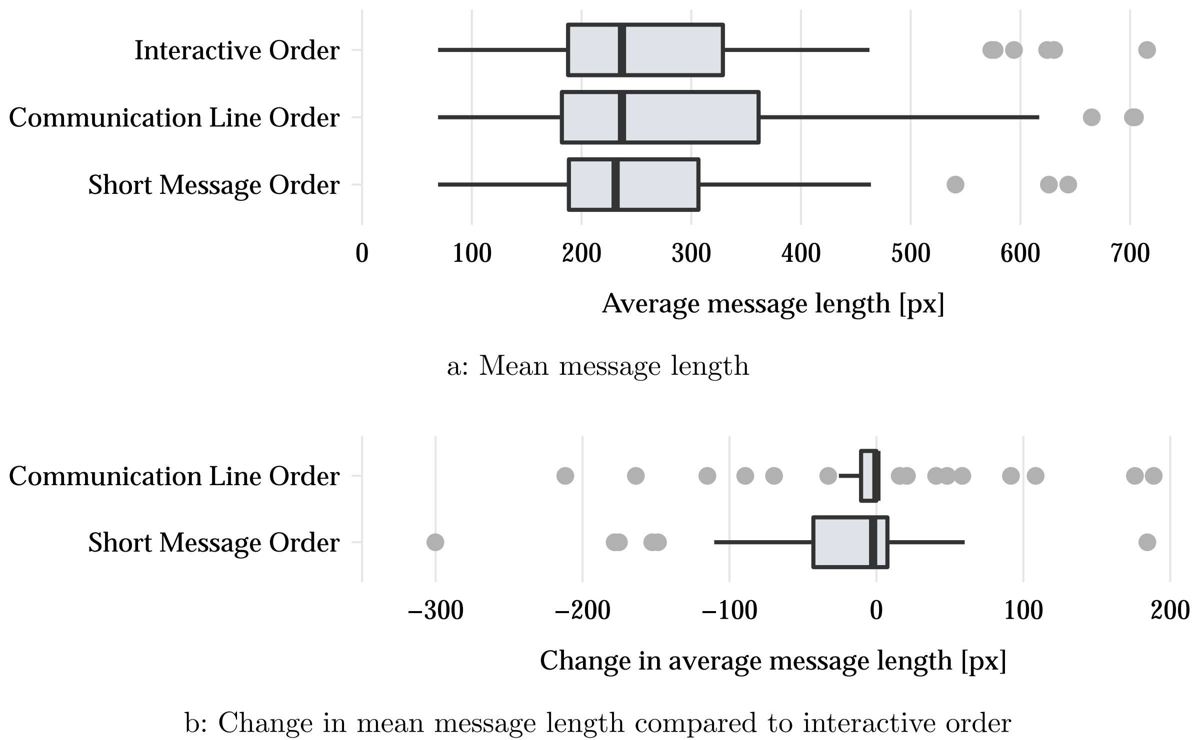
Figure 5.2a shows the number of crossings between messages and lifelines produced by the three strategies in each diagram. The way we counted crossings, neither the source nor the target lifeline of a message contributed to the total. In this metric, short message order has a considerably more pronounced effect, a conclusion confirmed Figure 5.2b, which shows the number of crossings relative to the original ordering. As Table 5.1 shows, short message ordering reduces the number of crossings in half of all diagrams.

5.1.2 Vertical Compaction

Let us turn to the second question: the influence of vertical compaction on diagram height. Figure 5.3a shows the height of diagrams with and without vertical compaction.

[1] `http://oss.models-db.com/`

[2] `https://rtsys.informatik.uni-kiel.de/~biblio/downloads/papers/report-1804-data.zip`



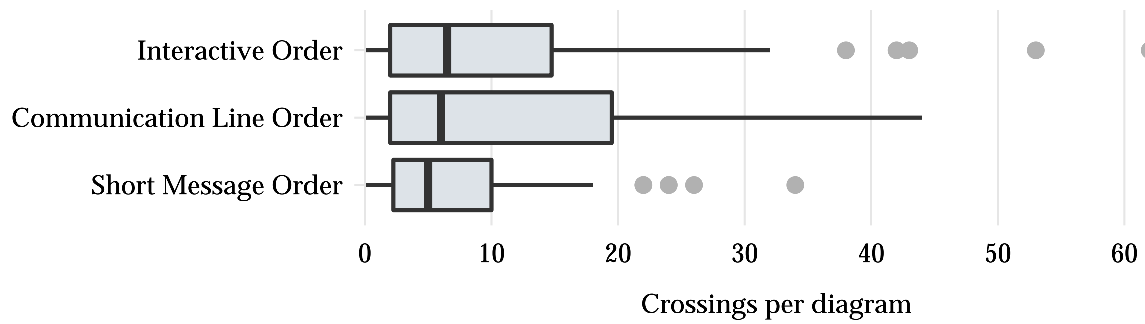
a: Mean message length

b: Change in mean message length compared to interactive order

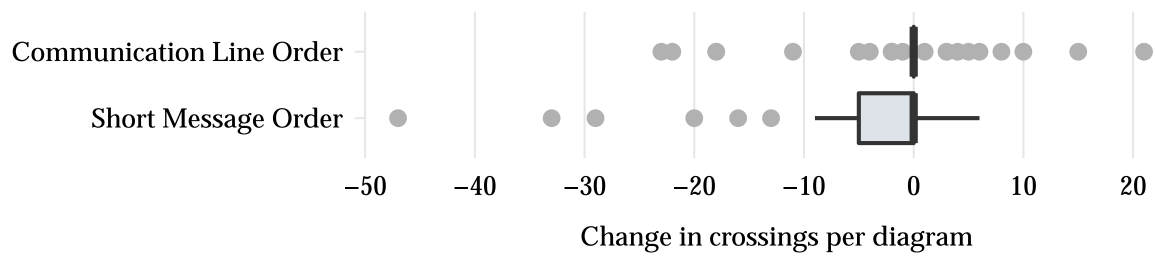
Figure 5.1: The mean message length produced by different lifeline ordering strategies for each sequence diagram, both overall and expressed as the change compared to the original lifeline order.

	Communication line order	Short message order
Mean message length		
Decreased	32	50
Unchanged	50	24
Increased	18	26
Message-lifeline crossings		
Decreased	32	50
Unchanged	50	24
Increased	18	26

Table 5.1: Percentage of diagrams whose message length or number of message-lifeline crossings have decreased, remained unchanged, or have increased subject to the lifeline ordering algorithm.



a: Message-lifeline crossings



b: Change in message-lifeline crossings compared to interactive order

Figure 5.2: The sum of message-lifeline crossings produced by different lifeline ordering strategies for each sequence diagram, both as absolute values and as the change compared to the original lifeline order.

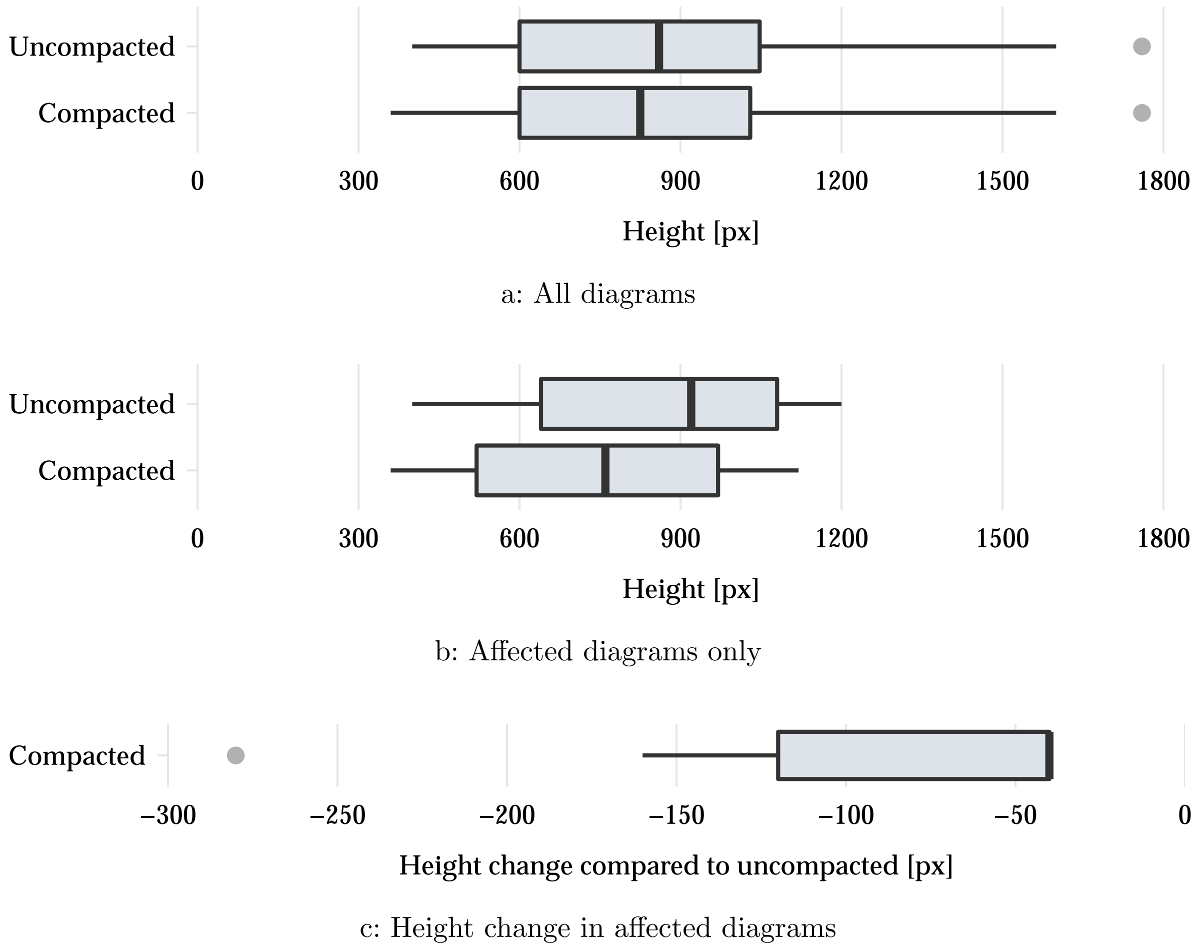


Figure 5.3: The height of drawings with and without vertical compaction, both for all diagrams and limited to those diagrams affected by compaction.

Both results are virtually identical, suggesting that vertical compaction does not have much of an effect. Indeed, closer inspection revealed that it affected only 18% of diagrams. If we limit our evaluation on those, we get the data shown in Figure 5.3b. If vertical compaction has a chance to do something, its influence is noticeable, as Figure 5.3c shows.

5.1.3 Discussion

It is hard to give final advice on how to configure ELK Sequence. As is often the case, the question of what is the "best" configuration depends on the situation. If one associates certain semantics with the order of lifelines, going for the short message order strategy might not be a good idea. The communication line strategy aims at finding a helpful order and will probably be the best choice if one does not yet have an idea of what the final diagram should look like, as is often the case while the user is still in the process of creating it. Since users may be irritated if the order of lifelines changes while editing,

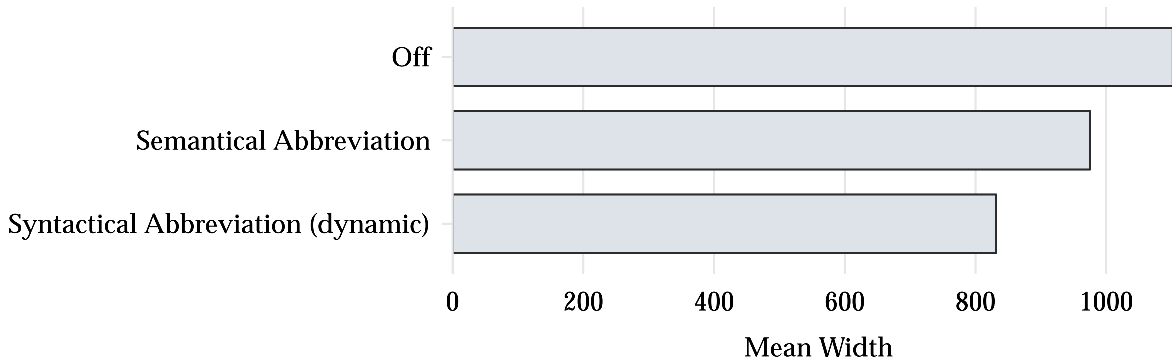


Figure 5.4: The mean width of sequence diagrams with different label management strategies as well as with label management turned off.

a sensible approach may be to keep the order they chose until they request it to be optimized.

Vertical compaction seems to be less ambivalent. While this may be different for other diagrams, it never had a negative effect on the diagrams in our test set. This leads us to believe that leaving it turned on will usually not be harmful.

5.2 Label Management

Every diagram was laid out once with every available label management strategy as well as with label management switched off. We measured each diagram's width and height and calculated the resulting aspect ratio (width divided by height). Since one of the goals of label management is to be able to increase the scaling a diagram is displayed with, we also calculated for each diagram how the scaling necessary to fit it on screen changes for each label management strategy compared to label management switched off.

The mean scaling factor increases where rather disappointing at first (1.03 for semantical abbreviation and 1.04 for syntactical abbreviation). The values do make sense, however: quite often, sequence diagrams are dominated by their height (true for 78% of diagrams in our data set), which dramatically reduces the impact of label management, a technique focussed on reducing the width of diagrams.

Which impact, then, do the label management strategies have on the width of our diagrams? Figure 5.4 shows that the impact is more dramatic here, with the mean diagram width reduced up to almost 25%.

If label management makes narrow diagrams narrower, why use it in the first place? First, not all diagrams are dominated by their height. And second, applying label management allows lifelines to move closer together. This helps when zooming into a diagram since more of it can be displayed on a screen, reducing the need for users to pan the view by increasing slidability.

6 Conclusions and Outlook

In this report we have demonstrated the two ways that the definition of sequence diagrams leaves for layout algorithms to exploit: computing the order of lifelines and allowing messages to share y coordinates in order to reduce their height. We have also shown how label management can be applied to reduce the width of sequence diagrams in interactive editing and viewing scenarios. How effective the methods are depends on the actual sequence diagram and the application. We thus believe that it is a good idea to allow users to experiment with different settings.

We currently plan for an implementation of the algorithm we presented to be included in the Eclipse Layout Kernel (ELK) project,[1] an open-source project that provides automatic layout algorithms and an infrastructure to support them. Until that has happened, preliminary versions can be made available upon request.

Regarding future work, while we have evaluated the methods in terms of their influence on aesthetics, a user study may be interesting in order to establish how well they are received in practice. We also think about allowing users to configure whether the space between communication lines should be uniform or not. If it need not be, that would—depending on the chosen distance between them—open up further avenues for vertical compaction.

[1] https://www.eclipse.org/elk/

Bibliography

[1] C. Bennett, D. Myers, M. Storey, D. M. German, D. Ouellet, M. Salois, and P. Charland. A survey and evaluation of tool features for understanding reverse-engineered sequence diagrams. *Journal of Software Maintenance and Evolution: Research and Practice*, 20(4):291–315, July 2008.

[2] G. Bist, N. MacKinnon, and S. Murphy. Sequence diagram presentation in technical documentation. In *Proceedings of the 22nd Annual International Conference on Design of Communication: The Engineering of Quality Documentation (SIGDOC '04)*, pages 128–133, New York, NY, USA, 2004. ACM.

[3] S. K. Card, J. Mackinlay, and B. Shneiderman. *Readings in Information Visualization: Using Vision to Think*. Morgan Kaufmann, Jan. 1999.

[4] H. Fuhrmann. *On the Pragmatics of Graphical Modeling*. Dissertation, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, Kiel, 2011.

[5] E. R. Gansner, E. Koutsofios, S. C. North, and K.-P. Vo. A technique for drawing directed graphs. *Software Engineering*, 19(3):214–230, 1993.

[6] M. R. Garey and D. S. Johnson. *Computers and Intractibility: A Guide to the Theory of NP-Completeness*. W. H. Freeman & Co, New York, 1979.

[7] R. Hebig, T. H. Quang, M. R. V. Chaudron, G. Robles, and M. A. Fernandez. The quest for open source projects that use UML: Mining GitHub. In *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS '16)*, pages 173–183, 2016.

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

[9] Object Management Group. OMG Unified Modeling Language Specification, Version 2.5.1, Dec. 2017. `https://www.omg.org/spec/UML/2.5.1/`.

[10] C. Papadimitriou. The NP-completeness of the bandwidth minimization problem. *Computing*, 16:263–270, 1976.

[11] M. Petre. UML in practice. In *Proceedings of the 2013 International Conference on Software Engineering (ICSE '13)*, pages 722–731, Piscataway, NJ, USA, 2013. IEEE Press.

[12] M. Petre. “No shit” or “Oh, shit!”: responses to observations on the use of UML in professional practice. *Software & Systems Modeling*, 13(4):1225–1235, Oct. 2014.

[13] T. Poranen, E. Mäkinen, and J. Nummenmaa. How to draw a sequence diagram. In *Proceedings of the Eighth Symposium on Programming Languages and Software Tools (SPLST'03)*, 2003.

[14] T. Reenskaug. Models – Views – Controllers, Dec. 1979. Xerox PARC technical note.

[15] C. D. Schulze, Y. Lasch, and R. von Hanxleden. Label management: Keeping complex diagrams usable. In *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '16)*, pages 3–11, 2016.

[16] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical system structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, Feb. 1981.

[17] K. Wong and D. 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.