On Order-preserving, Gap-avoiding Rectangle Packing

Sören Domrös[a], Daniel Lucas[b], Reinhard von Hanxleden[c] and Klaus Jansen[d]
Department of Computer Science, Kiel University, Westring, Kiel, Germany

Keywords: Automatic Layout, User Intentions, Rectangle Packing.

Abstract: We present 2D rectangle packing heuristics that preserve the initial ordering of the rectangles while maintaining a left-to-right reading direction. Furthermore, rectangles are placed such that inner whitespace (“gaps”) can be eliminated by enlarging and repositioning them without enlarging the drawing. This is achieved by initially approximating the required width and using a strip packing algorithm to pack the rectangles. The algorithms are suitable for interactive scenarios and can also be applied to strip packing problems to maintain the reading direction.

1 INTRODUCTION

Packing rectangles in a specific area, width, height, aspect ratio, or different bins has been studied on several occasions and remains in most cases a hard problem (Dowsland and Dowsland, 1992). This problem is relevant for numerous areas. For example, the transportation industry does not only need tightly packed packages, but also ordered packages and a stable packing (Da Silveira et al., 2014). The packages should be ordered such that they can be removed easily at a specified destination. This is expressed by adding stability and removal order constraints to the strip packing problem of determining the minimal height for an overlapping free packing of rectangles in a bounded width and infinite height.

Another application, which has actually motivated the work presented in this paper, is the placement of *regions* in SCCharts (von Hanxleden et al., 2013), a synchronous language for Model-Driven Engineering (MDE). SCCharts are a graphical language and graphical models are predominantly still created manually. In this paper, however, we are interested in creating SCChart diagrams automatically, from a purely textual model. Since the inception of SCCharts, this approach has been realized and stress-tested in the Kiel Integrated Environment for Layout Eclipse Rich Client (KIELER)[1] (Fuhrmann and von Hanxleden, 2010; von Hanxleden et al., 2011), which makes use of the open source Eclipse Layout Kernel (ELK)[2]. The application case of SCCharts has already inspired a number of works in the area of graph drawing (Chimani et al., 2011; Rüegg et al., 2014; Jabrayilov et al., 2016; Gutwenger et al., 2014; Rüegg et al., 2016; Rüegg et al., 2017; Schulze et al., 2014). However, these previous works have focused on the drawing of the node-link diagrams present in SCCharts; the rectangle packing problem posed by SCChart regions has not been covered so far.

As illustrated in Figure 1, SCChart regions can be *expanded* making their content visible, as it is the case for regions F, H, and K in the example, or *collapsed*. In both cases, each region requires a rectangular area. Collapsed regions usually have the same height but vary in width. Expanded regions can have any size and are typically much bigger than the collapsed regions.

An SCChart drawing usually has to fit into a certain drawing area that has a specific width and height, which also prescribes an aspect ratio. The drawing should not only fit this aspect ratio, but it should also use minimal area to draw regions with a higher zoom level, as expressed by the scale measure, formally defined later. A bigger scale measure means that regions can be drawn bigger, which means all components and labels can be read more easily.

Regions should also not have any “gaps”, i. e., no unnecessary whitespace between them. They should be placed such that their size can be increased to eliminate the remaining whitespace to obtain an aestheti-

[a] https://orcid.org/0000-0002-8011-8484
[b] https://orcid.org/0000-0001-6090-2637
[c] https://orcid.org/0000-0001-5691-1215
[d] https://orcid.org/0000-0001-8358-6796
[1] www.rtsys.informatik.uni-kiel.de/en/research/kieler

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

cally pleasant drawing. For example, Figure 1d contains undesired whitespace, which is eliminated in Figure 1e.

Contributions & Outline

The technical contributions are:

- The presentation and formalization of the region packing problem (Section 2), including placement constraints that lead to a reading direction for regions, as well as whitespace elimination constraints.
- A first, rather simplistic *box* algorithm to solve the region packing problem (Section 3, see Figure 1b).
- Second, an alternative approach, the *rectpacking* algorithm (Section 4, seen in Figure 1c). This includes a width approximation step and also introduces and provides specific treatment of the *one big region* case.
- Finally, the *LR-rectpacking* algorithm (Section 5, see Figure 1e). This includes an improved compaction step to maintain the reading direction as well as the elimination of special handling for the one big region case.

The algorithms are evaluated against an optimal solution (Section 6, see Figure 1f). We conclude and present future work in Section 7.

Related Work

Dowsland and Dowsland give an overview of numerous works on rectangle packing and strip packing (Dowsland and Dowsland, 1992). Each of them does not consider ordering and reading direction as it is proposed by this paper.

Da Silveira et al. present a strip packing algorithm that considers the packaging order by assigning packages to classes, which indicate the removal order (Da Silveira et al., 2013). They initially place packages in rows as in the box layout algorithm. However, to optimize the used space, they reverse the even rows and compact the drawing, which may destroy the reading direction and may produce a drawing in which whitespace cannot be eliminated.

Augustine et al. explore strip packing with precedence constraints and strip packing with release times for FPGA programming (Augustine et al., 2006). Rectangles are placed such that they are above/below other rectangles or above their release time, which represents dependencies between different jobs. This placement also allows to eliminate the whitespace between the different rectangles, but this is not considered in this context. In contrast to our work, their precedence constraints only restrict the vertical placement and do not consider a reading direction.

Kenyon and Rémila present an asymptotic fully polynomial approximation scheme for strip-packing (Kenyon and Rémila, 2000). They pack all wide rectangles sorted by width in a stack and group neighboring rectangles by their cumulative heights. Rounding up rectangle width in a group allows to solve this as a fractional strip packing problem. The narrow rectangles are placed using a next fit decreasing height algorithm. In contrast to this paper, the ordering of the rectangles is not considered. However, it might be useful to design an approximation algorithm for the proposed rectangle packing problem.

Bruls et al. suggest a method to visualize file system or company structures via treemaps (Bruls et al., 2000). They solve the issue of long and small rectangles in treemaps by forming rows and subrows similar to our approach. However, they order the rectangles by their size, since this produces the best drawings. Moreover, only the area of their rectangles is given. The rectangle bounds can be changed. We have to deal with a minimum width and height instead and consider a reading direction.

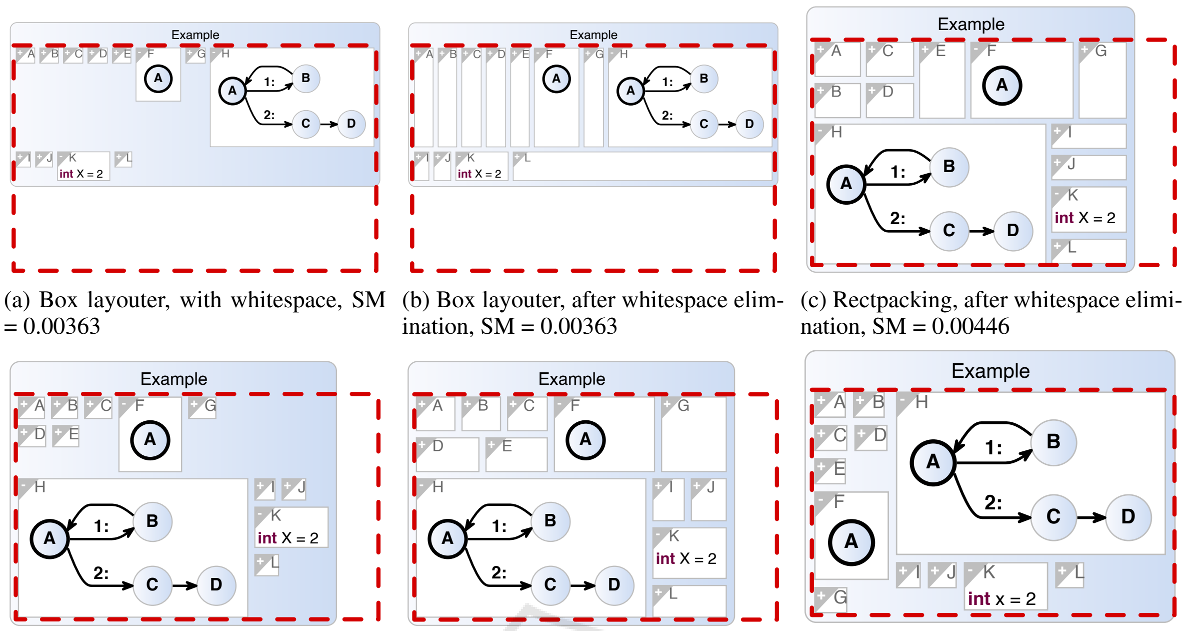
Wang et al. present EdWordle, which allows to move and edit words in wordclouds while preserving the neighborhood of words (Wang et al., 2017). In contrast to our approach this does not consider on ordering of the words. Moreover, a reading direction is not necessary in that scenario, gaps are allowed, it is not necessary to align words in rows of columns, and the dimensions of words seem rather restricted and do not seem to differ much in size.

The second author presented the *rectpacking* algorithm in his thesis (Lucas, 2018), including further details on how the width approximation works, how the algorithm was implemented, and how the one big region case works.

2 THE REGION PACKING PROBLEM

Given an ordered sequence of regions $R = (r_1, r_2, \ldots r_n)$ with $r_i = (w_i, h_i)$ for region r_i, with the minimal width w_i and minimal height h_i, we compute a drawing by assigning coordinates x_i and y_i and a new computed width w_i and height h_i to each region r_i. Henceforth, we call the regions r_{i-1} and r_{i+1} the *neighbors* of r_i.

Clearly, a requirement on the drawing is that regions must not overlap with other regions, and that computed widths/heights are at least the specified minimum widths/heights. Beyond these correctness



(a) Box layouter, with whitespace, SM = 0.00363
(b) Box layouter, after whitespace elimination, SM = 0.00363
(c) Rectpacking, after whitespace elimination, SM = 0.00446
(d) LR-rectpacking, with whitespace, SM = 0.00446
(e) LR-rectpacking, after whitespace elimination, SM = 0.00446
(f) CP optimizer solution, SM = 0.00508

Figure 1: An SCChart with region F, H, and K expanded, desired aspect ratio of 1.6 (red-dashed bounding box), with scale measure SM (larger is better).

requirements, we seek to produce drawings that make best possible use of the drawing area, and that consider the mental map of the user, as detailed in the following sections.

Since we talk about layout creation and the mental map is usually only used for layout adjustment (Purchase et al., 2006), we need to broaden this term. The user begins to create a mental map by writing the textual model. We should preserve the order of that textual model also in the corresponding diagram. Since our drawings are not one dimensional but two dimensional, the drawing should make clear when regions are ordered horizontally and when vertically. A trivial solution to achieve this is the box layout algorithm (see Section 3), which has a clear placement of regions in rows and a left to right reading direction. Moreover, we deem the dimension of a region as unimportant to recognize the region. In an SCCharts scenario, inner state machines define the look of a region, which is not influenced by the dimension of the region. To summarize, regions should be discoverable by their name, their contents, and their display order.

2.1 Rows, Blocks, and Subrows

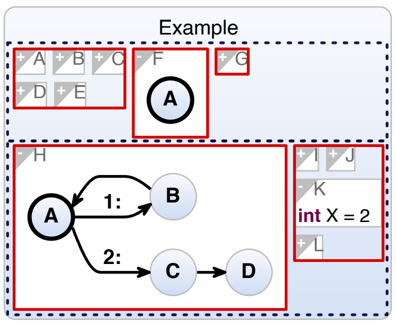
If we want to talk about reading direction and placement of regions, we have to introduce proper terminology to describe region placement and alignment.

A drawing consists of *rows*, which in turn consist of *blocks*, which in turn consist of of *subrows* (rows within rows). Figure 2 has two rows. The top row consist of three blocks. The first block consist of two subrows. The top subrow consist of three regions: region A, region B, and region C. Henceforth we call the upper bound of a row the *row level* of that row. In Figure 2 region A, B, C, F, G, H, I, and J are on the row level of their row. This means they align at their top with their row level.

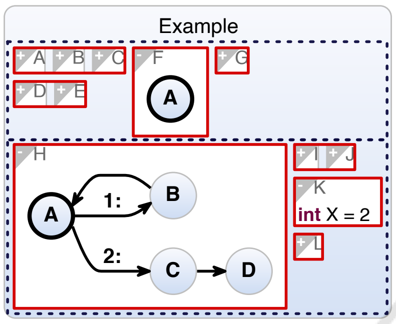
2.2 Aspect Ratio and Scale Measure

The size of the different elements is an important limiting factor regarding readability and understandability.

Let $A = (w_d, h_d)$ be the drawing area defined by the desired width w_d and the desired height h_d. This defines the *desired aspect ratio* $DAR = w_d/h_d$. Similar to the *DAR*, the *aspect ratio* is defined as $AR = \frac{w_a}{h_a}$ given by the actual width w_a and the actual height h_a. The *original scale measure* $\text{OSM} = \min(\frac{w_d}{w_a}, \frac{h_d}{h_a})$ expresses how well the drawing uses the given area (Rüegg and von Hanxleden, 2018). E.g., an OSM of 1 means that both the width and the height fit the drawing area, but that the drawing cannot be enlarged anymore without exceeding the drawing area. An OSM of 0.5 means that the drawing has to be shrunk by a



(a) Blocks (red)



(b) Subrows (red)

Figure 2: Rows, blocks, and subrows (dotted, black) in a rectangle packing.

factor of 2 to fit the drawing area. Clearly, a larger OSM allows a more readable diagram and is hence better. In our case we only have a given desired aspect ratio and not the desired width and height. This means that we can arbitrarily assume $h_d = 1$, which results in $w_d = DAR$. This yields the *scale measure* $\mathrm{SM} = \min(\frac{DAR}{w_a}, \frac{1}{h_a})$, which is based on the desired aspect ratio DAR. Again, just as for the OSM, larger SM is better. Practically, we allow ourselves to produce drawings of arbitrary height and width, and leave it up to the rendering to scale the drawing such that it fits the drawing area, modulo zooming/panning actions taken by the user.

2.3 Ordering and Whitespace

We assume that regions are correctly ordered if for each pair of regions r_i, r_j with $i < j$ the following holds:

$$x_i + w_i \leq x_j \vee y_i + h_i \leq y_j$$

This means that the region i is horizontally or vertically before the region j. This constraint is not enough to identify the order of the regions just by their placement. In bigger graphs this hinders the ability to discover regions in the drawing even if the previous ordering and the position of its neighbors is known. The reading direction (visible in Figure 1d) makes the region ordering better discernible. Here regions are placed such that they preferably align horizontally and form rows of regions.

A region is either placed

(1) directly right of its preceding neighbor

(2) right of its preceding neighbor on the current row level

(3) in the next subrow

(4) in the next row

This is not enough to clearly identify a reading direction, as seen in Figure 1d and 1f. A random alignment of subrows, as in Figure 1f, could irritate the user and suggest to continue right instead of down. Therefore, the height of each row should be defined by the highest element in it. This helps the user to follow a reading direction.

We also want to be able to eliminate the whitespace in a drawing, as seen in Figure 1d compared to Figure 1e. This allows to use the available space to its full extent and creates a visually pleasing drawing by increasing the width or height of the regions. Moreover, it is easier to follow the region order. If empty space is below and right of a region, it is unclear where to continue. If the whitespace is eliminated, the user continues below, if the right region does not align horizontally with the current one, and right if it does align. It is not always possible to remove all whitespace in arbitrary drawings. However, since we only consider the position (1) to (4) for a node, the whitespace can always be eliminated.

A lower-numbered constraint is preferred over a higher one to maintain a left-to-right reading direction. Constraint (1) is true if a region is directly right of its preceding neighbor (see region B in Figure 1d). A region fulfills (2) if it is on its row level right of its preceding neighbor. Region F in Figure 1d fulfills this constraint. Constraint (3) means a region is in the next subrow below and possibly left of its preceding neighbor (see i. e. regions H and K in Figure 1d). Region H in Figure 1d is an example for Constraint (4). The box layouter fulfills (1) or (4) but fulfills neither (2) nor (3) without fulfilling (1) or (4) at the same time. Since the (2) implies (1) and (3) implies (4) in case of the box layouter. Rectpacking considers all of them, but prefers (3) over (2). LR-rectpacking uses these constraints as they are intended to maintain the reading direction.

Additionally to the reading direction, the drawing has to be space efficient and near the desired aspect ratio to produce better drawings than the one in Figure 1b, which is expressed by the scale measure.

To summarize, we want an algorithm that produces drawings with a high scale measure, makes regions discoverable by maintaining a reading direction, and places the regions such that the inner whitespace can be eliminated.

3 BOX LAYOUTER

Algorithm 1 presents the box algorithm, a greedy algorithm for layouting regions.

The algorithm reduces the rectangle packing problem to a strip packing problem by estimating the width based on the area and the region sizes (Algorithm 2).

The layout algorithm places the regions next to each other in rows (line 3) without considering the height of such a row, as seen in Figure 1a.

Next, the regions are expanded to fill the row height (see regions A to G in Figure 1a compared to Figure 1b). The last region in a row is also horizontally enlarged (see region F). Note that it is also possible to divide the available width among all regions of a row, but the use of this algorithm allows to only enlarge the width of the last node compared to the following ones.

Algorithm 1: box.

```
Input: Regions $rs$, $DAR$
Output: Placed regions $rs$
1 // Width approximation, see Algorithm 2
2 $width = \text{approx}(rs, DAR)$
3 boxPlace($rs, width$) // Region placement
4 expand($rs$) // Whitespace elimination
```

Algorithm 2: boxWidthApproximation.

```
Input: Regions $rs$, $DAR$
Output: Approximated width
1 $totalArea = \sum area(rs)$
2 $area = totalArea + |rs| * \text{stddev}(totalArea)$
3 return $\max(\text{maxWidth}(rs), \sqrt{area * DAR})$
```

The algorithm produces rather good drawings for graphs with regions of similar height. Since the regions are aligned in rows, one can find a region without much effort if their initial ordering is known. In the SCCharts case, big regions, which contain state machines, are easy to identify and can be used as a reference point. Here, the shape of the region itself is irrelevant for the mental map, since the shape of their innards is preserved and it is discoverable by the position of its neighboring regions.

Algorithm 3: boxPlace.

```
Input: Regions $rs$, width $w$
Output: Placed regions $rs$
1 $lineX, lineY, lineHeight = 0$
2 foreach $r$ in $rs$ do
3     if $lineX + r.width \leq w$ then
4         $r.x = lineX$
5         $r.y = lineY$
6         $lineX \mathrel{+}= r.width$
7         $lineHeight =$
            $\max(lineHeight, r.height)$
8     else
9         $lineY \mathrel{+}= lineHeight$
10        $lineX = 0$
11        $lineHeight = r.height$
12        $r.x = 0$
13        $r.y = lineY$
```

However, big regions are also the weak point of this algorithm, as seen in Figure 1b. For graphs with different region sizes, the estimated width is too big, and by stacking the regions inside a row the whitespace could be drastically reduced. As a result, region names and inner behavior are very small and difficult to read. The user needs pan and zoom action to understand and read everything. This limits understandability and is quite time consuming.

One big expanded region and many small collapsed regions are a common use case in SCCharts development and inspired the one big region case in the rectpacking algorithm (see Section 4.4).

4 RECTPACKING HEURISTIC

The rectpacking algorithm aims to optimize the scale measure SM (see Section 2.2), and with it the readability of the diagram, while preserving a reading direction to not disturb the mental map of the user. A structural overview can be seen in Algorithm 4. The algorithm exists in two variations, LR-rectpacking and plain rectpacking, which have the same structure but vary in which subroutines are called, as indicated by the optional "lr" prefix.

Algorithm 4: [LR-]rectpacking.

```
Input: Regions $rs$, $DAR$
Output: Placed regions $rs$
1 $width = \text{rpApprox}(rs, DAR)$
2 [lr]rpPlace($rs, width$)
3 [lr]compact($rs$)
4 [lr]expand($rs, width, DAR$)
```

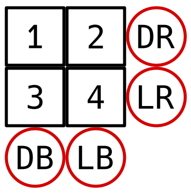


Figure 3: Candidate positions.

4.1 Width Approximation

If one wants to improve the box algorithm, a more compact drawing is expected. Since the static width approximation of the box algorithm does not take the order of the rectangles into account and does over-estimate the width in no relation to the actual drawing width, we propose to use a greedy placement algorithm that does not consider reading direction (see Figure 4a) but approximates the area better than the box algorithm, as seen in Figure 4c.

The greedy algorithm places one region after the other on one of four candidate positions, as seen in Figure 3.

- Directly right of the last placed region (LR)
- Right of the whole drawing, top aligned (DR)
- Directly below the last placed region (LB)
- Below the whole drawing, left aligned (DB)

These positions are weighted based on the resulting scale measure, area, and aspect ratio. If one wants to optimize scale measure, it is the primary criterion, area the secondary, and aspect ratio the tertiary. If one wants to optimize the aspect ratio, they are applied in the following order: aspect ratio, area, scale measure. Depending on whether the aspect ratio or the scale measure should be optimized, the option that currently yields the best values is chosen. Regions placed at LR might be shifted up, regions placed at LB might be shifted left to optimize the current drawing, since the width is often overestimated. After potentially shifting the previous region, the next one is placed. A complete placement can be seen in Figure 4a. The scale measure is the main optimization goal. In this example, region E is placed on the DR candidate position, since DR is preferred over LR and both produce the same scale measure, which is better than the one of the DB and LB position, since the used *DAR* prefers wider drawings over higher ones.

4.2 Region Placement and Compaction

The approximated width reduces the problem to a strip packing problem. In the first step, the regions are placed in rows the same way as in the box algorithm, as seen in Figure 4b. Using the maximum height of a row, the drawing is compacted by assigning regions to *stacks*, which are sequences of regions that are aligned vertically, inside a row, as seen in Figure 4c. Note that this imposes a top-down reading direction compared to the left-to-right reading direction of the rows. In Figure 4c region A and B, region C and D, region E, region F, region G, region H, and region I, J, K and L form a stack.

4.3 Whitespace Elimination

The whitespace is eliminated by iterating through all stacks in a row and enlarging the height of the regions in the stacks such that the stack fits the row height, as seen in Figure 5. The additional width of a row is divided among all stacks. Moreover, all stacks are enlarged to fit the row height. All regions in a stack are enlarged to fit the stack width and equally enlarged to fit the new stack height, which creates an aesthetically pleasing drawing (Figure 1b compared to Figure 1c).

4.4 Big Region Handling

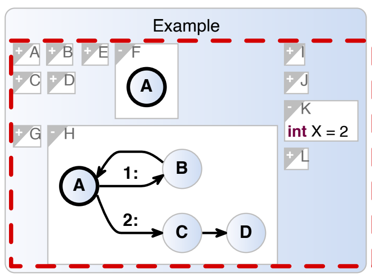
The rectpacking heuristic includes a special case handling for sets of regions that indicate SCCharts regions with one expanded region, i. e. a set of regions with one big region and other regions with the same height. We henceforth refer to this as the *one big region* case.

We consider a left-to-right reading direction for the regions, as in Figure 6b, to be more pleasing than the standard case top-down reading direction in Figure 6a. Moreover, a left-to-right reading direction is already present in the rows the rectpacking algorithm forms.

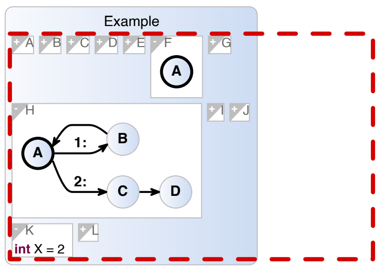
To achieve drawings as in Figure 6b, we assume that all small regions before and after the big region have the same width and height respectively. Since all regions have the same size they can easily be placed in rows, as seen in Figure 6b. For the normal case this is not trivial, since we only know the bounding height, but no bounding width. This is why we form rows of stacks, what is basically solving a vertical strip packing problem using the box layouter vertically.

5 LR-RECTPACKING

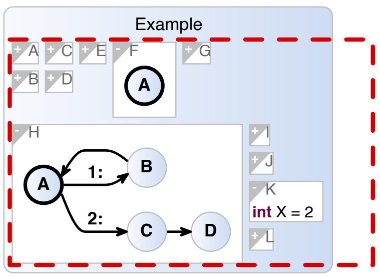
The rectpacking algorithm achieves a better scale measure than the box layouter. However, it lacks a consistent reading direction. Furthermore, the one big region case does not consider the *DAR* and makes the



(a) After width approximation, SM = 0.00446

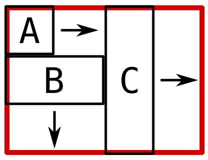


(b) After placement, SM = 0.00371

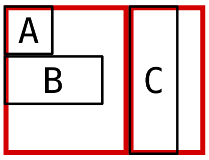


(c) After compaction, SM = 0.00446

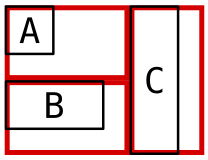
Figure 4: Layout with rectpacking heuristic, *DAR* = 1.6 (red-dashed).



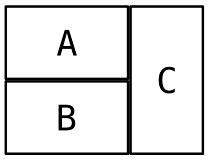
(a) After compaction



(b) Move stacks

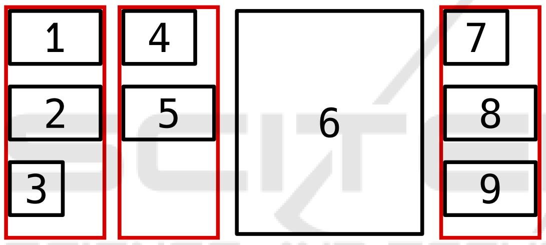


(c) Move regions

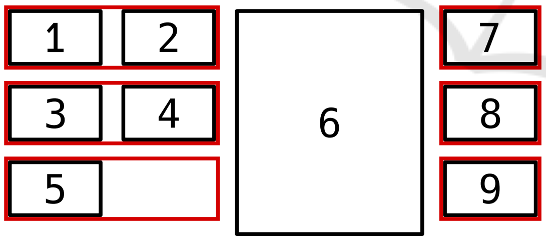


(d) Expand regions

Figure 5: Rectpacking whitespace elimination in a row, the arrows indicate the direction in which the regions are enlarged.



(a) Normal stacks



(b) One big region case layout

Figure 6: Normal rectpacking compared to the one big region case.

algorithm difficult to maintain in terms of software engineering.

We, therefore, propose the following algorithm that changes the placement, compaction, and expansion step. The width approximation step does not change. It places the regions to get a target width, as seen in Figure 4a and described in Section 4.1.

5.1 Region Placement

In the next step, the regions are placed in rows, as seen in Algorithm 5, which results in the same placement as the region placement of the rectpacking algorithm in Figure 4b. However, regions with “similar height” are grouped into *blocks*, since such regions do not lose too much space if they are aligned in subrows. A region fits in a block, if it can be put directly right of all regions in it without exceeding the target width and has a similar height than the regions in this block. The height, width, and position of rows and blocks are updated on creation. Regions are placed when they are added to their block. As seen in Figure 4b, blocks only have one subrow and the result looks like a placement the box layouter would create.

Later, during compaction, such blocks are grouped to form *stacks of blocks*. After region placement each block forms a stack of blocks containing only itself. During compaction other blocks can be added to it as described in the following.

5.2 Region Compaction

Given the blocks, which are drawn as flat as possible, and stacks of blocks that only contain one block after placement, we try to compact the drawing by a one pass algorithm which handles each block one after the other, as seen in Algorithm 6. After placement, the row heights are defined by the highest node in them. This height only decreases during compaction.

Algorithm 5: lrrpPlace.

```
Input: Regions rs, width w
Output: Placed regions rs
Row row = new Row(w)
Stack stack = new Stack(row)
Block block = new Block(row, stack)
foreach r in rs do
    similar = hasSimilarHeight(block, r)
    fit = fitRow(block, r)
    if similar ∧ fit then
        block.add(r)
    else if fit then
        stack = new Stack(row)
        block = new Block(row, stack, block)
        block.add(r)
    else
        row = new Row(w, row)
        stack = new Stack(row)
        block = new Block(row, stack, block)
        block.add(r)
```

Algorithm 6: lrcompact.

```
Input: Regions rs, width w
Output: Placed regions rs
rows = getRows(rs)
foreach row ∈ rows do
    foreach block ∈ row.blocks do
        Block next = block.nextBlock()
        block.addRegions(next)
        if fitTop(block, next) then
            block.stack.add(next)
        else if fitRight(block, next, w) then
            block.stack.drawInRowHeight()
            block.placeRight(next)
        else
            block.stack.drawInRowWidth(w)
```

The current block places the next block above itself if it does not exceed the row height (see line 6-7). If the block fits, it is added to the stack of the current block (see blocks I to J, K, and L in Figure 1d). If the next block is in the next row, its regions are added to the current block if their height is similar to the current block height (see line 5).

If placing the next block below itself is not possible but the next block does fit right (line 8), the current stack is drawn as narrow as possible (see stack A to E in Figure 1d and line 9-10). Else the current stack is drawn as flat as possible in the remaining row width to facilitate the left-to-right reading direction.

Remember that each block aligns their regions in subrows, which does not waste much space since their height is similar. Therefore, one wants to find the minimal width such that all blocks of the current stack do not exceed the row height, which is realized by a binary search algorithm. This leaves room for improvement. If the number of region widths is limited, we only need to check width permutations between the minimum stack width and the current stack width. The next block/stack is placed next to the compacted current block or at the beginning of the next row and we continue with the next block/stack of blocks until all blocks are placed.

5.3 Whitespace Elimination

The whitespace elimination step in LR-rectpacking is similar to the one of the rectpacking procedure presented in Section 4.3 and produces drawings such as the one in Figure 1e. Since the regions are now grouped in subrows, blocks, and stacks of blocks, the additional width and height is now equally divided under the stacks of blocks in each row, the blocks, and henceforth the subrows and the regions in them (see Figure 7) instead of dividing it among the stacks and their regions, as it is done in Section 4.3.

The algorithm has an option to enlarge the regions such that the drawing exactly fits the *DAR* during the whitespace elimination step.

5.4 Worst Case

Clearly this algorithm performs worst, if very high and very wide regions alternate, as seen in Figure 8.

In this example, the rows are highlighted in dotted black, stacks of blocks and blocks are highlighted in red if their bounds are unclear. For Figure 8a, the width approximation yields a much too small width, since it does not respect the reading direction. The optimal packing (see Figure 8b) gets more freedom since its row height is not limited by the highest element in it. However, the reading direction suffers. Without a proper numbering of the regions, one would rather think that n8 is before n7. Only the left aligned placement with n4 lets the user guess that n7 should belong to the same row as n8. Whitespace elimination helps to highlight this alignment, since n2 and n8 would fill their entire block and would enframe the row and highlight the blocks and stacks. The influence on readability has to be quantitatively determined in future work.

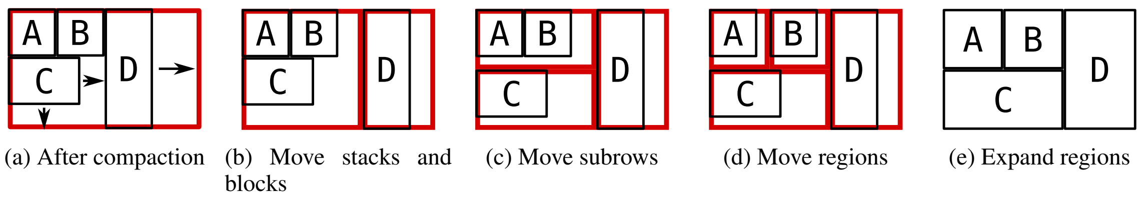


Figure 7: LR-rectpacking whitespace elimination in a row, the arrows indicate the direction in which the regions are enlarged.

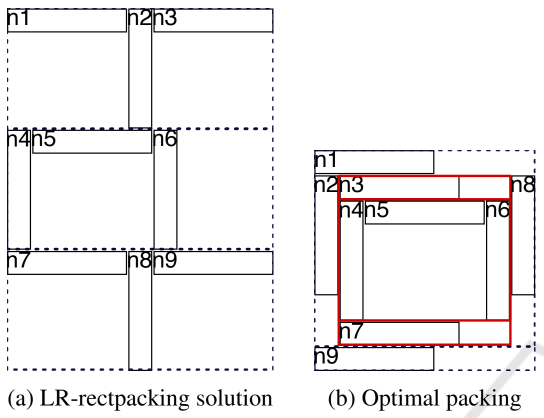


Figure 8: Worst case for LR-rectpacking.

6 EVALUATION

All layout algorithm were implemented in the ELK framework[3]. We evaluated the performance of the algorithms using the GrAna tool (Rieß, 2010) with 200 graphs for each graph class. The number of regions is between 20 and 30 to make the instances solvable by CP optimizer. Normal regions have a height of 20 and a width with the mean of 100 and a standard deviation of 20. The big nodes class (BN) has 2 to 5 big regions with a width and height between 300 and 1000. The one big node class (OB) has only one big region. The same height class (SH) has no big regions. The graphs are drawn and evaluated for six different algorithm configurations with a *DAR* of 1.3 and a spacing of 1 between regions:

- B: Using the box layouter with set priorities to enforce region order
- AA: Using only the width approximation step of the rectpacking heuristic optimized for aspect ratio (see Section 4.1)
- EA: Using LR-rectpacking optimized for aspect ratio
- AS: Using the width approximation step of the rectpacking heuristic optimized for scale measure (see Section 4.1)
- ES: Using LR-rectpacking optimized for scale measure
- OS: Using CP optimizer

OS maximizes the scale measure and minimizes the used area as a secondary criterion. The run time of OS is limited to at most one hour per graph. Five graphs in BN, and two in OB were removed, since the time limit was reached. The regions are placed by considering only the position (1) to (4) in Section 2.3, as seen in Figure 1f.

The run times of the box algorithm and the rectpacking algorithm are clearly in $O(n)$. LR-rectpacking solves the placement problem in $O(n\log(n))$ since the compaction step uses binary search to calculate the best width for a block. However, in practice the run time of B, AA, EA, AS, and ES seem linear in the problem size and are in our experiments in millisecond range.

The box layouter heuristic (B) handles same height regions (SH) quite well, but if big regions occur it tends to overestimate the needed width, as seen in Figure 9a – 9c. Therefore, the aspect ratio suffers, as seen in Figure 9h and Figure 9i. The height of the OB and BN graphs seems to depend mostly on the size of the big regions in these graphs, as seen in Figure 9e and Figure 9f. This can be observed in the scale measure for the box layouter in Figure 9m – 9o. In the SH graphs it achieves a relatively good scale measure, a better one than the LR-rectpacking approaches EA and ES. However, the big region graphs OB and BN tend to have smaller scale measures than the rectpacking approaches.

The LR-rectpacking approaches optimized for aspect ratio (EA) and scale measure (ES) do not perform well for the SH graphs. Since the greedy width approximation overestimates the width, the following strip packing cannot perform well in these cases, as seen in Figure 9a. As a result the height tends to be smaller than needed, as seen in Figure 9d. However, this cannot be generalized, and there are OB or BN cases that underestimate the width. For the big region cases OB and BN the algorithm achieves values near the optimum, as seen in the aspect ratio in Figure 9h and Figure 9i, the whitespace in Figure 9k and Figure 9l, and the resulting scale measure in Figure 9n

[3]https://www.eclipse.org/elk/reference/algorithms.html

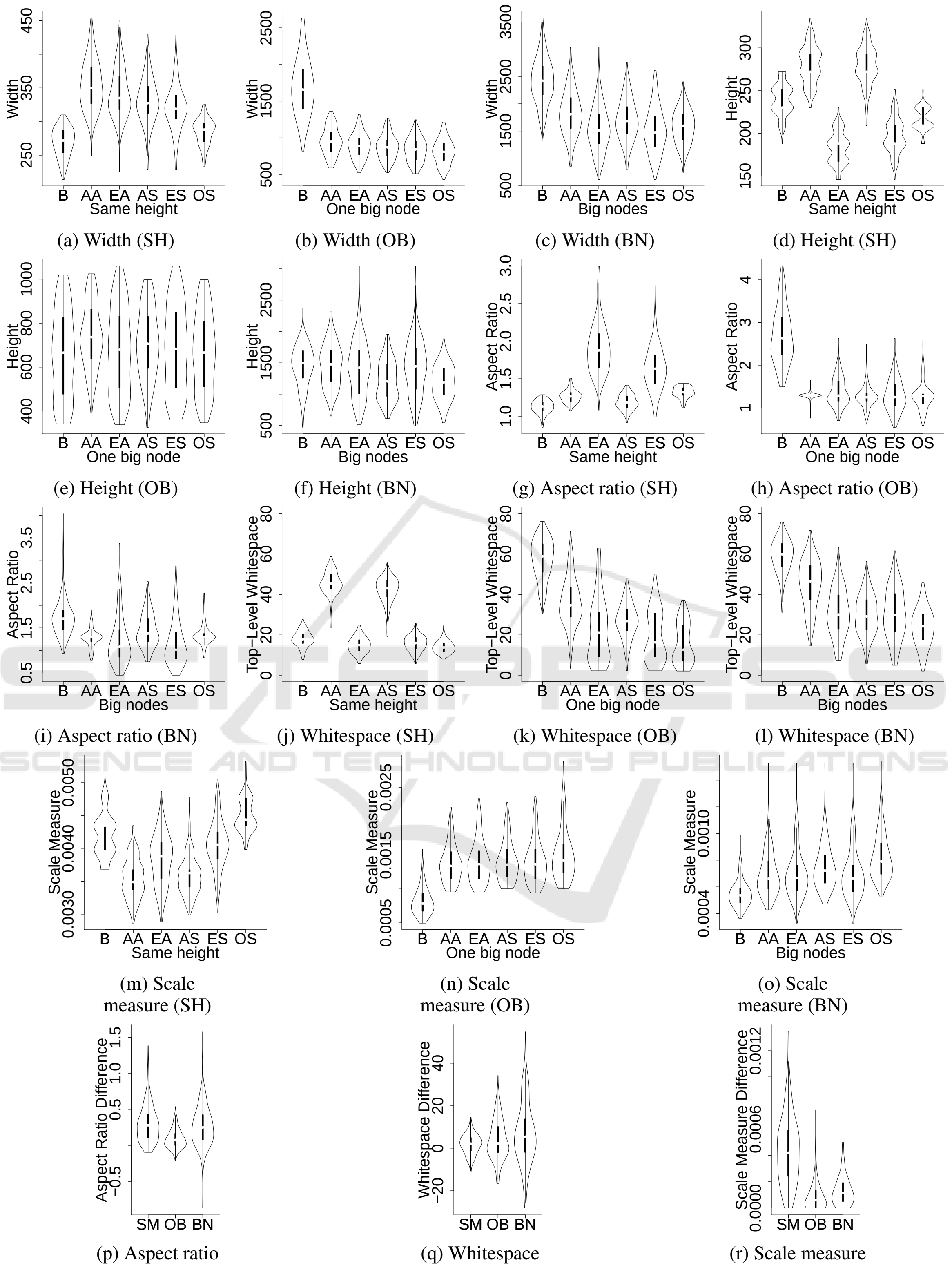


Figure 9: Width, height aspect ratio, whitespace, scale measure, and comparison.

and Figure 9o. It outperforms the box layouter in all non-trivial cases.

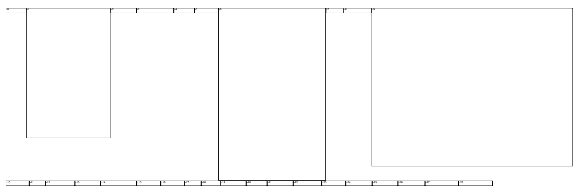
The heuristics focused on aspect ratio (AA, EA) and scale measure (AS, ES) perform nearly the same. They both optimize the aspect ratio to be as close as possible at the desired aspect ratio in the width approximation step. The AA approach in Figure 9g – 9i seems to be more strict to achieve the aspect ratio. In terms of scale measure, the scale measure approach, not surprisingly, outperforms the aspect ratio approach. In terms of aspect ratio the aspect ratio approach achieves better results. When whitespace is eliminated to fit the aspect ratio, AA is the better algorithm. Otherwise, ES should be used. Therefore, ES is the standard approximation strategy for the rectpacking algorithm in ELK.

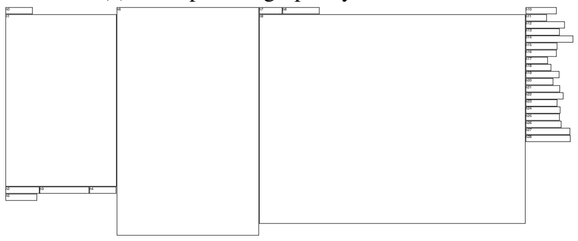
Figure 9p shows a comparison between ES and OS in terms of aspect ratio, calculated via $|DAR - ar(\text{ES})| - |DAR - ar(\text{OS})|$ with ar being the aspect ratio for the given set of regions. A negative value means that ES is closer to the desired aspect ratio than OS. OS is nearly always better than the heuristic. The negative cases occur because space was sacrificed to achieve a better aspect ratio.

Figure 9q presents the difference in whitespace usage for the specific graphs $ws(\text{OS}) - ws(\text{ES})$, with ws being the whitespace for the given set of regions. Again, a negative value means that ES is better than OS. For the SH graph set, OS is nearly always better. For OB and BN the median is near zero and some cases are even better. However, these cases have a worse aspect ratio.

Figure 9r shows the difference in scale measure $sm(\text{OS}) - sm(\text{ES})$ with sm being the scale measure. As expected, no negative values are present. The heuristic seems to be most effective for graphs with only one big region and bad for sets without big regions, as mentioned before, and is in most cases near the optimum.

Figure 10 shows an example BN graph layouted with B, ES and OS. Figure 10c achieves a better scale measure than Figure 10b. However, the reading direction suffers. In Figure 10c subrows of different blocks align, which visually creates rows that do not exist. Figure 10a shows that graphs layouted with the box layouter always align their regions in rows. The resulting scale measure is clearly worse than the ones of Figure 10 and Figure 10c. The solution is clearly far from optimal since the algorithm refrains from stacking the regions inside the rows.


(a) Example BN graph layouted with B


(b) Example BN graph layouted with ES

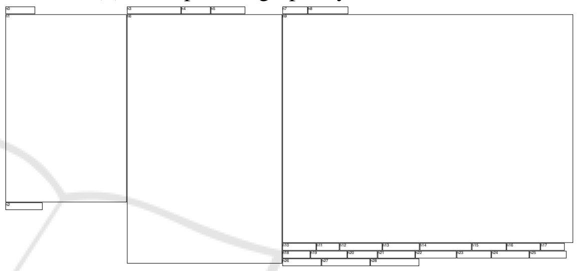

(c) Example BN graph layouted with OS

Figure 10: Example BN graph.

7 CONCLUSION AND FUTURE WORK

The proposed LR-rectpacking algorithm achieves better results than the box layouter and the initial rectpacking algorithm in all interesting cases. Same height regions usually only occur in SCCharts if all regions are collapsed. Then the zoom level is not that important since all regions are relatively small in this case. We propose to use the LR-rectpacking algorithm for SCCharts regions in the future. This algorithm is currently also used in the Object Explorer tool[4] (created by Xplain Data) in order to analyze complex mass data. In this application a user may open multiple windows. The “box” or “rectpacking” algorithm is used to later create a smooth layout.

The heuristic produces drawings with scale measures near the optimum for the graphs with at least one big region.

As seen in the results for SH graphs, the algorithm has room for improvement in the width approximation step, which will be optimized in future work.

[4]https://www.xplain-data.de/

We believe that an ordering of regions and a reading direction helps the user to discover regions and helps to maintain the mental map. The exact influence of this has to be determined in future work.

Part of future work is to analyze real SCCharts to compare the algorithms and to include user feedback for the different drawings.

REFERENCES

Augustine, J., Banerjee, S., and Irani, S. (2006). Strip packing with precedence constraints and strip packing with release times. In *Proceedings of the Eighteenth Annual Acm Symposium on Parallelism in Algorithms and Architectures (SPAA'06)*, pages 180–189, New York, NY, USA. ACM.

Bruls, M., Huizing, K., and Van Wijk, J. J. (2000). Squarified treemaps. In *Data visualization 2000*, pages 33–42. Springer.

Chimani, M., Gutwenger, C., Mutzel, P., Spönemann, M., and Wong, H.-M. (2011). Crossing minimization and layouts of directed hypergraphs with port constraints. In *Proceedings of the 18th International Symposium on Graph Drawing (GD '10)*, volume 6502 of *LNCS*, pages 141–152. Springer.

Da Silveira, J. L., Miyazawa, F. K., and Xavier, E. C. (2013). Heuristics for the strip packing problem with unloading constraints. *Computers & operations research*, 40(4):991–1003.

Da Silveira, J. L., Xavier, E. C., and Miyazawa, F. K. (2014). Two-dimensional strip packing with unloading constraints. *Discrete Applied Mathematics*, 164:512–521.

Dowsland, K. A. and Dowsland, W. B. (1992). Packing problems. *European Journal of Operational Research*, 56(1):2 – 14.

Fuhrmann, H. and von Hanxleden, R. (2010). On the pragmatics of model-based design. In *Proceedings of the 15th Monterey Workshop 2008 on the Foundations of Computer Software. Future Trends and Techniques for Development, Revised Selected Papers*, volume 6028 of *LNCS*, pages 116–140, Budapest, HR. Springer.

Gutwenger, C., von Hanxleden, R., Mutzel, P., Rüegg, U., and Spönemann, M. (2014). Examining the compactness of automatic layout algorithms for practical diagrams. In *Proceedings of the Workshop on Graph Visualization in Practice (GraphViP '14)*, pages 42–52.

Jabrayilov, A., Mallach, S., Mutzel, P., Rüegg, U., and von Hanxleden, R. (2016). Compact layered drawings of general directed graphs. In *Proceedings of the 24th International Symposium on Graph Drawing and Network Visualization (GD '16)*, pages 209–221.

Kenyon, C. and Rémila, E. (2000). A near-optimal solution to a two-dimensional cutting stock problem. *Mathematics of Operations Research*, 25(4):645–656.

Lucas, D. (2018). Order- and drawing area-aware packing of rectangles. Bachelor thesis, Christian-Albrechts-Universität zu Kiel, Faculty of Engineering. https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/dalu-bt.pdf.

Purchase, H. C., Hoggan, E. E., and Görg, C. (2006). How important is the "mental map"? – an empirical investigation of a dynamic graph layout algorithm. In *Proceedings of the 14th International Symposium on Graph Drawing (GD '06)*, volume 4372 of *LNCS*, pages 184–195. Springer.

Rieß, M. (2010). A graph editor for algorithm engineering. Bachelor thesis, Kiel University, Department of Computer Science.

Rüegg, U., Ehlers, T., Spönemann, M., and von Hanxleden, R. (2017). Generalized layerings for arbitrary and fixed drawing areas. *Journal of Graph Algorithms and Applications*, 21(5):823–856.

Rüegg, U., Kieffer, S., Dwyer, T., Marriott, K., and Wybrow, M. (2014). Stress-minimizing orthogonal layout of data flow diagrams with ports. In *Proceedings of the 22nd International Symposium on Graph Drawing (GD '14)*, pages 319–330.

Rüegg, U., Schulze, C. D., Grevismühl, D., and von Hanxleden, R. (2016). Using one-dimensional compaction for smaller graph drawings. In *Proceedings of the 9th International Conference on the Theory and Application of Diagrams (DIAGRAMS '16)*, volume 9781 of *LNCS*, pages 212–218. Springer.

Rüegg, U. and von Hanxleden, R. (2018). Wrapping layered graphs. In *Proceedings of the 10th International Conference on the Theory and Application of Diagrams (DIAGRAMS '18)*, pages 743–747. Springer.

Schulze, C. D., Spönemann, M., and von Hanxleden, R. (2014). Drawing layered graphs with port constraints. *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout*, 25(2):89–106.

von Hanxleden, R., Duderstadt, B., Motika, C., Smyth, S., Mendler, M., Aguado, J., Mercer, S., and O'Brien, O. (2013). SCCharts: Sequentially Constructive Statecharts for safety-critical applications. Technical Report 1311, Christian-Albrechts-Universität zu Kiel, Department of Computer Science. ISSN 2192-6247.

von Hanxleden, R., Fuhrmann, H., and Spönemann, M. (2011). KIELER—The KIEL Integrated Environment for Layout Eclipse Rich Client. In *Proceedings of the Design, Automation and Test in Europe University Booth (DATE '11)*, Grenoble, France.

Wang, Y., Chu, X., Bao, C., Zhu, L., Deussen, O., Chen, B., and Sedlmair, M. (2017). Edwordle: Consistency-preserving word cloud editing. *IEEE transactions on visualization and computer graphics*, 24(1):647–656.