# Incremental Diagram Layout for Automated Model Migration

Ulf Rüegg
Dept. of Computer Science
Kiel University
Kiel, Germany
uru@informatik.uni-kiel.de

Rajneesh Lakkundi
Ashwin Prasad
Anand Kodaganur
National Instruments
Bangalore, India
{rajneesh.lakkundi, ashwin.prasad, anand.kodaganur}@ni.com

Christoph Daniel Schulze
Reinhard von Hanxleden
Dept. of Computer Science
Kiel University
Kiel, Germany
{rvh,cds}@informatik.uni-kiel.de

## ABSTRACT

A range of successful modeling tools to develop complex systems use node-link-style diagrams as their underlying language. Over the years such languages can change, for instance as part of a tool update. When migrating existing models, changes in syntax directly affect the placement of elements in their diagrams. Increasing the size of certain nodes may for example result in node overlaps.

In this paper we propose two methods based on graph drawing techniques to adjust the layout of existing diagrams after migration. Although we designed these techniques for diagram migration, they are applicable to other scenarios as well, such as users interactively adding or resizing nodes. We evaluate the techniques based on real world diagrams from the LabVIEW suite and discuss the scenarios each technique seems best suited for.

## 1. INTRODUCTION

Model-Driven Engineering (MDE) is an established means to develop complex software systems. Well-known tools such as *LabVIEW* (National Instruments) or *Simulink* (Mathworks) represent models as visual diagrams, more specifically as dataflow diagrams where *actors* (or *nodes*) consume and produce data transmitted between actors through *links* (or *edges*) that connect them. Such graphical representations are intuitive and aim to help the developer cope with increasing complexity.

Just like software written in textual languages, software written in graphical languages evolves as its requirements change and as problems are fixed. However, there are also external factors that force software to be changed that developers have less control over: both the languages and the tools used to develop the software change over time, requiring the software to be adapted accordingly. This can affect both the *abstract syntax* as well as the *concrete syntax*. While changes to the abstract syntax may not affect the diagram at all, changes in the concrete syntax can have immediate consequences: if the size of a given node changes from one version of the language to the next, this may result in overlaps with other nodes that were not present before in existing diagrams, effectively reducing legibility; if a diagram with node overlaps is considered to be illegal by the language, this may even break existing diagrams. A more severe variant of this problem arises if an actor ceases to be offered by the language and must be replaced by a small network of other actors that replicate its functionality in the language's new version.

Fig. 1 shows a diagram migrated from one version of a visual language to the next, both before and after solving the aforementioned layout problems caused by the migration. Note how the layout problems are solved in a way that keeps the diagram's general topology fairly stable in order to not invalidate the user's *mental map* [19] of the diagram. Such layout stability, however, is just one possible goal and can conflict with other aesthetics criteria. For example, not trying to keep the layout stable will give the layout algorithm the freedom to reduce the diagram's size or to reduce the number of edge crossings by changing the diagram's topology. We will argue that it depends on the use case how much freedom should be given to the layout algorithm.

Migration scenarios like this of course not only affect one diagram, but all diagrams. Asking users to manually fix layout problems resulting from tool updates is not acceptable, giving rise to the need for an automated process. If such a process works well (and most importantly does not break existing diagrams), it gives language and tool developers the means to provide updates to users with a smooth upgrade path and a minimum amount of additional work. Tools to support such a process have been discussed before, but a lack of adequate layout support has been identified [22, 21]. Filling that gap is the aim of this paper.

Another interesting use case, though not explicitly discussed in this paper, is the presentation of the same diagram across different tools that may use slightly different concrete syntaxes.

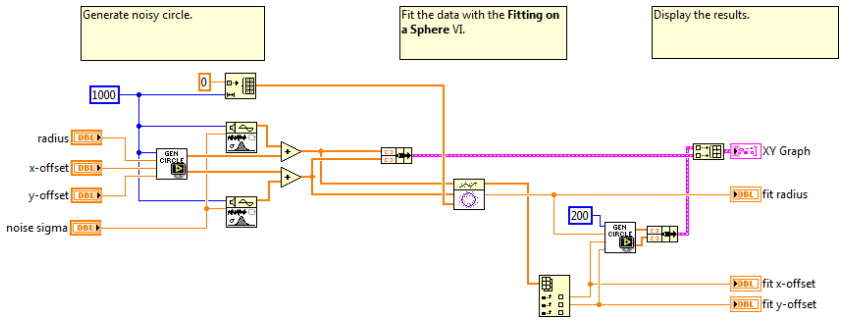
*Contributions.* Our main contributions are twofold. First, we present two layout algorithms that fix invalid layouts



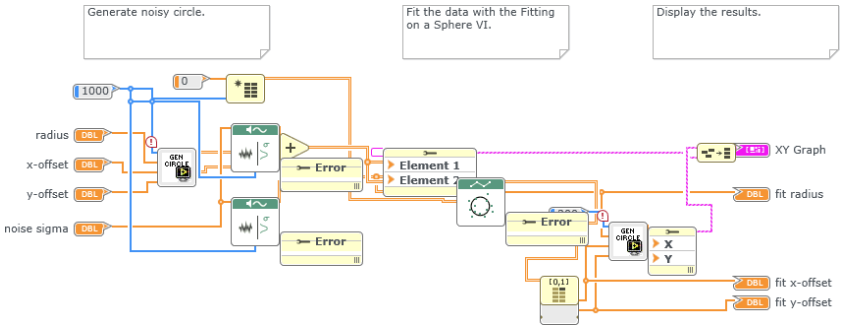
*MODELS '16 October 02-07, 2016, Saint-Malo, France*



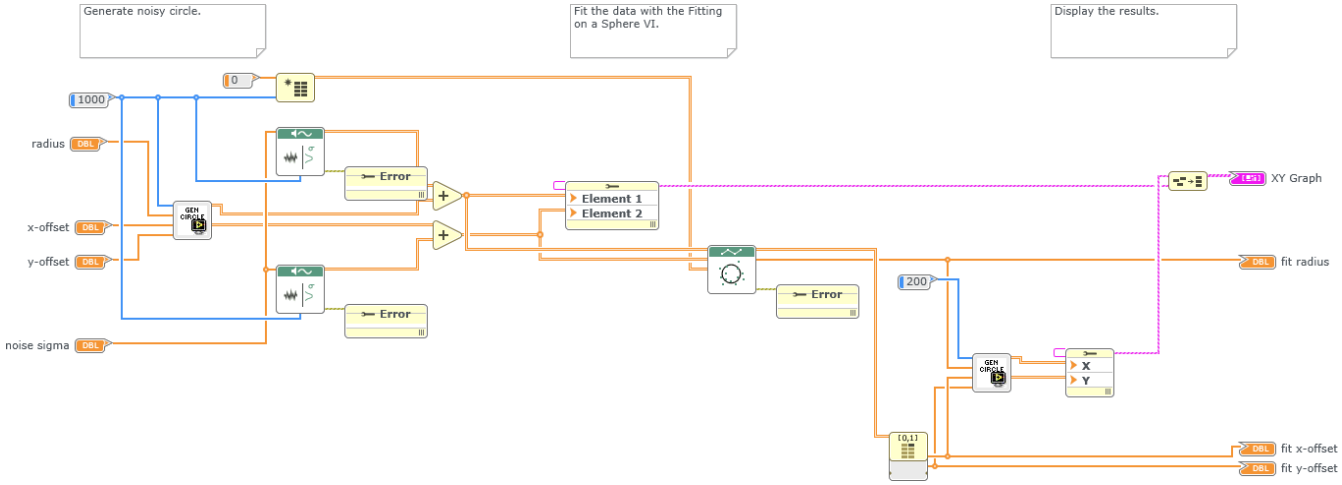
ACM ISBN 978-1-4503-4321-3/16/10.
DOI: http://dx.doi.org/10.1145/2976767.2976805




(a) Original, before migration




(b) Migrated, before applying methods from this paper




(c) Migrated, after applying methods from this paper

Figure 1: A diagram migrated from one version of the used language to the next. All images have the same scaling applied to them.

while preserving the user's mental map. One is a novel algorithm based on the network simplex algorithm, the second has been introduced for other scenarios before [26], but is adapted and applied to our scenario. Second, we present a case study to evaluate different layout approaches and their results. The case study is based on industrial-scale dataflow diagrams of the LabVIEW suite. Each diagram was written in one version of the language and then automatically migrated to a new version, which entails node size changes and changes in the actors available to developers.

*Outline.* The remainder of this paper is structured as follows. In Sec. 2 we formally introduce the problem we tackle and summarize related work in Sec. 3. Sec. 4 presents the two methods we propose, which we compare and evaluate in Sec. 5. We conclude in Sec. 6.

## 2. PROBLEM DESCRIPTION

During model migration one or more of the following events can happen, some of which are shown in Fig. 2:

- Structural
  - New nodes are introduced
  - Existing nodes are deleted
  - Existing nodes are split into multiple new nodes
  - New ports are introduced
  - New edges are introduced
  - Edges change connection points
- Appearance
  - Node dimensions change
  - Ports alter their relative position

The list is not exhaustive. Note that the structural changes affect the underlying graph as well as the diagram, while the changes to the appearance only affect the latter.

Let $D$ denote the drawing of an existing diagrams, i. e. the given positions for nodes, ports, and edges. After model migration, the problem is to find a drawing $D'$ in which no pair of nodes overlaps. Next, we discuss additional goals that may be of interest to obtain high-quality results.

### 2.1 Possible Strategies

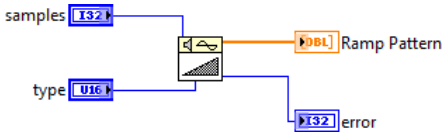
Possible strategies to find a drawing $D'$ given $D$ include the following:

S1 Use existing positions, new elements are positioned at the diagram's origin.

S2 Remove overlaps between elements but preserve the original character of the diagram, and find reasonable positions for new elements.

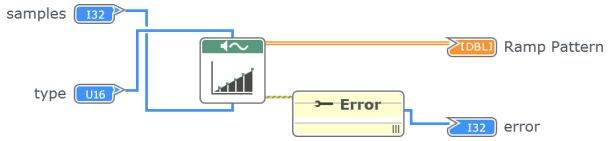
S3 Determine positions for all elements from scratch.

While S1 is the easiest solution from a technical point of view, it implies high manual effort for the user to clean up the migrated diagram. In general, this strategy is not acceptable as all models to be migrated are affected. For well-established tools this can mean thousands of diagrams. Therefore, we consider an overlap-free drawing to be a hard requirement, leaving us with strategies S2 and S3. At this




(a) The original diagram, written in the language's old version.


(b) The diagram as migrated into the language's new version. Most node sizes have increased, the position and order of connection points have changed, and the center node was split into two nodes with a new edge inserted to connected them.




(c) The new diagram after the first method from this paper has been applied. Nodes have been moved to reserve enough space for larger and additional nodes. The order of the samples and type nodes has been preserved even though this results in an edge crossing. New nodes were properly integrated into the layout.

Figure 2: A diagram as it is converted into a new version of the underlying language.

point (subjective) *aesthetics* of a drawing have to be considered: diagram elements must be placed and routed in such a way that the result is syntactically correct and understandable. Both remaining strategies aim to meet this requirement, but do so in different ways. S2 assumes that the user has invested time and thought into how to place the diagram elements in the original version. It thus tries to change the layout as little as possible, only moving nodes around as is necessary to remove overlaps and trying to stick with existing positions as much as possible. Ideally, this also helps the user recognize and understand the diagram quickly as the result broadly corresponds to the mental map they already have of the diagram. This approach to automatic layout is also known as *layout adjustment* [19]. S3 assumes that maintaining existing positions is not as important as computing a clean layout from scratch that meets aesthetic criteria that are deemed important, such as few edge crossings. This is also called *layout creation* and is what most automatic layout algorithms do unless they are further modified. Whatever strategy is employed, the involved algorithms must observe typical layout features of the type of diagram to be laid out. For dataflow diagrams, these mainly include the following:

- Making the flow of data obvious by having the majority of edges point in the same direction.
- Routing edges orthogonally, i. e. each edge segment runs either vertically or horizontally.
- Respect the containment of nodes that are children of (hierarchical) nodes.

Therefore, preserving the overall look of a dataflow diagram includes making sure that nodes are correctly positioned relative to each other (which effectively preserves the diagram's topology), keeping edges straight if they were straight in the original diagram, and, possibly less importantly, preserving node alignments. As an example consider Fig. 2. The two nodes samples and type are aligned w. r. t. to their left x-coordinate. The edge between the middle node and Ramp Pattern is straight.

## 3. RELATED WORK

Model migration in the context of MDE has been discussed before, and existing tools have been compared in the form of contests [22]. Paige et al. emphasize, however, that the migration of a visual model's concrete syntax, i. e. drawing, has not been given enough attention yet [21].

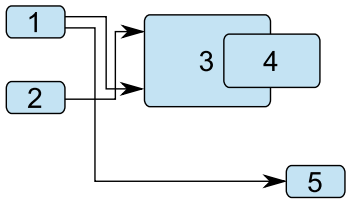
Rearranging elements of a diagram is studied in the area of *graph drawing*, and *creational* and *adjusting* layout techniques are distinguished [19]. While creational layout algorithms determine positions for diagram elements from scratch, adjusting layout algorithms rely on existing positions to derive new positions that are *similar* to the original positions while pursuing additional goals, for instance, to remove overlaps between diagram elements.

Similarity is often defined in terms of preserving the user's *mental map* [19], i. e. when a user looks at a newly layouted diagram that they were familiar with, they should immediately recognize the diagram. Mental map preservation is defined as preserving *topology*, *proximity*, and *orthogonal ordering*, and studies have been conducted to determine meaningful metrics to measure the similarity of two drawings [2, 4]. However, while the identified metrics give a general feeling of similarity, they may miss domain-specific knowledge and requirements.

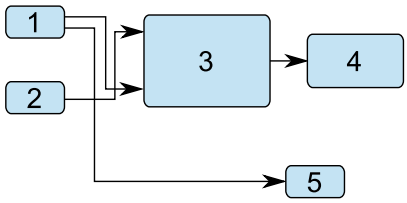
Several papers tackle the problem of removing node overlaps in a given drawing, using a broad variety of techniques [14, 18, 17, 6, 16, 10]. While most of them aim at a trade-off between preserving the mental map and minimizing area, usually no further features, such as node alignments, are preserved.

Topology-preserving methods for most of the well-known graph drawing approaches have been presented. For instance, Bridgeman et al. addressed *orthogonal* drawing methods [3], Dwyer et al. used a stress model and separation constraints for *force-based* drawing methods [7], and North and Woodhull extended the *layer-based* layout method [20]. Still, the produced drawings reflect the aesthetics selected by the specific method, while our first method is chosen to first and foremost preserve the mental map. In our use case this is much more important since the aesthetic aspect of the diagram was decided by the user. Another example of a topology-preserving algorithm is the *sketch-driven* approach by Brandes et al. [1]. It takes a sketch of a drawing as input and produces an orthogonal-style drawings similar to the input.

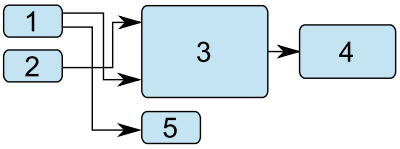
The most closely related work to our first approach is the method presented by Freivalds and Kikusts [9]. They preserve the topology by minimizing the distance between a node's new and old position subject to certain ordering constraints. They use a more intricate solution method based on quadratic programming and gradient projection, and do not yet consider preserving further diagram features, such as straight edges and alignments.



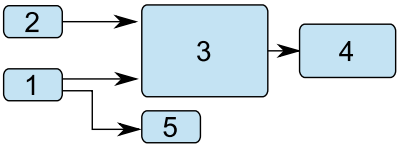

(a) The diagram after conversion, without any methods applied yet. Nodes 3 and 4 overlap.




(b) The diagram after applying network simplex-based incremental layout. The overlap has been removed with minimal effort, by moving node 4 rightwards.




(c) The diagram after applying interactive layer-based layout. The diagram's topology is still intact, but the layout has been "cleaned up" by moving node 5 closer to the other nodes, resulting in a smaller diagram.




(d) The diagram after applying fully automatic layer-based layout. Switching the order of nodes 1 and 2 changes the topology, but eliminates two edge crossings.

Figure 3: Results produced by the different layout methods described in this paper.

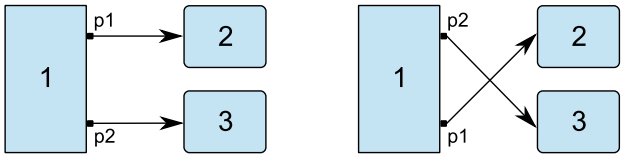
## 4. METHODS

In this section we present two methods for migrating the concrete syntax of existing diagrams, as summarized in Fig. 3. The goal is to update the layout of a diagram such that no pair of diagram elements overlaps and additionally certain features, such as node alignments, are preserved. While we optimized our methods for the specific requirements of dataflow diagrams, they can easily be adapted to other diagram types. In Sec. 2 we distinguished three strategies that can be pursued when migrating a diagram, with S2 being the one we address in this paper. We stated that preserving "the overall look of a dataflow diagram" is important. This can be concretized further by the following (not mutually exclusive) points:

S2.1 Move elements as little as possible.

S2.2 Preserve the mental map.

S2.3 Preserve additional "features" of the diagram.



(a) Original diagram (b) Swapped ports

Figure 4: Since the ports $p1$ and $p2$ swapped their relative order it is not possible to preserve the straightness of both edges without changing the order of nodes 2 and 3.

Before we continue, we briefly introduce some notations. A dataflow diagram, and similar diagrams, can be formalized as follows. Let $G = (V, P, E)$ be a graph with nodes, ports, and edges. Every node $v \in V$ has a position $(x_v, y_v) \in \mathbb{R}^2$ and size $(w_v, h_v) \in \mathbb{R}^{+2}$. Two nodes are said to *overlap* if their rectangular bounding boxes overlap. *Ports* $p \in P$ belong to nodes, where $\pi(p)$ denotes the *parent node*. A port has a position $(x_p, y_p)$ relative to the top-left position of $\pi(p)$. Edges $e = (p, q) \in E$ connect nodes via ports, where $\pi(p), \pi(q) \in V$, and are additionally described by a sequence of bendpoints $bp(e) = ((x_1, y_1), \ldots, (x_n, y_n))$ that describes how they are routed through the diagram. Edges are usually not allowed to overlap, unless they share a common source or target port.

*New Nodes.* We have to decide where to place new nodes that have not been part of the original diagram. Assume there is a new node $w^*$, which is related to an existing node $v$, e. g. by an edge that connects the two. We create a small auxiliary graph from $w^*$ and $v$ and layout that graph with existing creational layout techniques. Afterwards we think of the two combined nodes as a single node $v'$ in the original graph with a bounding box matching the area of the layouted auxiliary graph. Thus, for every original node $v \in G$ there is exactly one related node $v' \in G'$.

If an existing node $v$ gets split into multiple new nodes $w_1^*, \cdots, w_n^*$, we can proceed in a similar fashion. Only instead of creating an auxiliary graph for two nodes, we create one for $n$ nodes.

*Hierarchy.* As mentioned earlier dataflow diagrams can contain hierarchical nodes. In particular, the diagram itself can be regarded as a hierarchical node and the content of inner hierarchical nodes can be seen as sub-diagrams. In our experience, the number of nodes per sub-diagram is usually quite manageable. In every method discussed here, we treat each sub-diagram separately in a bottom-up fashion, starting with the inner-most diagram working our way up to the root of the hierarchy.

## 4.1 Network Simplex-Based Incremental Layout

The first method we discuss can intuitively be described by a linear program (LP). We will show later how it can be implemented without the need for a dedicated solver. Given a graph $G = (V, P, E)$, let $G' = (V', P', E')$ denote the graph after migration. Note that starting now, $v' \in V'$ always denotes the corresponding node of $v \in V$. The method aims to:

**Algorithm 1:** Ordering Constraints

**Input:** $G = (V, E)$: graph with positioned nodes
**Output:** $C$: set of constraints

```
// Step #1: Collect left and right boundaries of nodes
boundaries ← empty list
for v ∈ V do
    boundaries add (v, LEFT, x_v)
    boundaries add (v, RIGHT, x_v + w_v)
sort boundaries by ascending x-coordinate
// Step #2: Execute scanline
active_nodes ← empty set
for b = (v, s, x) ∈ boundaries do
    if s == LEFT then
        add constraints between active_nodes and v
    else
        add v to active_nodes
        for constraint c incoming to v do
            remove source node of c from active_nodes
```

1. keep $x_{v'}$ as close to $x_v$ as possible.
2. preserve the orthogonal ordering of $G$ as much as possible; that is, for two nodes $v, w \in V$ it must hold that $x_v < x_w \Leftrightarrow x_{v'} < x_{w'}$ and $y_v < y_w \Leftrightarrow y_{v'} < y_{w'}$.
3. preserve as many domain-specific artifacts as possible.

Note that by the way we treat nodes, we can assume that there is an injective mapping of the nodes, such that for every $v' \in V'$ there is a unique $v \in V$, with $|V| \geq |V'|$.

To simplify matters, we treat the x and y-dimensions separately and limit our explanations to the former; the y-dimension works equivalently. Furthermore, we allow node positions to increase only: $x_v \leq x_{v'} \ \forall v \in V$. This matches our case study, in which nodes usually grow in size, requiring other nodes to shift rightwards or downwards to accommodate them. If nodes can shrink as well, this method can still be applied with an additional postprocessing step to compact the drawing [23].

These assumptions allow us to state the basic migration procedure as a simple optimization problem:

$$\min \quad \sum_{v' \in V'} x_{v'} \tag{1}$$

$$\text{subject to} \quad x_v \leq x_{v'} \ \forall v \in V \tag{2}$$

Intuitively this simply minimizes the distance between the original and new positions. In the remainder of this section, we will add constraints to ensure the layout is valid and the characteristics we want to preserve are actually preserved.

*Orthogonal Ordering.* To keep nodes from overlapping and to preserve the orthogonal ordering, we introduce the following additional constraints for each pair of nodes subject to a minimum spacing $s_x$ to be kept between them:

$$x_a + w_a < x_b \Rightarrow x_{a'} + w_{a'} + s_x < x_{b'} \ \forall a \neq b \in V \tag{3}$$

Naively every pair of nodes has to be compared to find all required constraints and several of the constraints in Eq. 3 make constraints of Eq. 2 superfluous. However, most constraints are transitive: if $u, v, w \in V$ such that $u$ is left of $v$ and $v$ is left of $w$, the complete set of constraints would

include the constraint for $u$ to be left of $w$, even though that is already implied by the other two constraints.

With this in mind, we apply the scanline-based algorithm outlined in Alg. 1. The scanline proceeds from left to right and reacts to two kinds of events: "a node starts" and "a node ends". In step 1 of the algorithm, the left and right boundaries of each node are collected as they appear in the diagram, sorted from left to right by their x-coordinate. Step 2 contains the actual constraint building logic. The idea is that at any given time, we have a set of active nodes: nodes whose right boundary was encountered by the scanline and no constraint was created to another node in the active list. Whenever we encounter the start of a new node $v$, we add constraints from the active nodes to $v$ since the active nodes have ended before $v$ starts. However, all constraints incoming to $v$ do not have to be added to nodes we encounter afterwards. Nodes that have constraints incoming to $v$ are no longer active when $v$ becomes active, thereby preventing transitive constraints from being added.

Note that adding these constraints yields a valid, overlap-free layout. However, it can be argued that some of the generated constraints are still unnecessary. Take for example two nodes $v, w \in V$ such that $v$ is left of $w$. If $v$ and $w$ overlap in the y direction, a constraint between them is necessary to keep them from overlapping in the final diagram. If, however, we assume that $v$ is at the top of the diagram and $w$ is at the bottom, the user will arguably not recognize any order between them, thus rendering any constraints between them unnecessary. We leave such considerations for future work.

*Straight Edges.* Let $S \subseteq E$ be the set of edges that were straight in the original drawing. A straight edge in the x-dimension essentially is an alignment of the involved nodes in the x-dimension with a certain offset induced by the ports. Since ports can change positions or swap their relative order it may not be feasible to preserve all straight edges (cf. Fig. 4). Therefore, instead of realizing the straight edge-preservation as equality constraints over the nodes' positions, we make it part of the objective, by minimizing:

$$\sum_{(p,q)\in S} \omega^s_{(p,q)} \cdot \mid (x_{\pi(p)'} + x_{p'}) - (x_{\pi(q)'} + x_{q'}) \mid, \qquad (4)$$

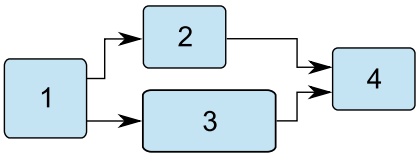
where $\omega^s_{(p,q)} \in \mathbb{R}^+$ denotes a weight indicating the importance of keeping an edge straight. The set $S$ can be calculated in time linear to the number of edges by checking each edge for straightness in the original drawing.

*Alignments.* Let $A \subseteq (V \times V)$ contain pairs of nodes that are aligned in the original diagram and still present in the migrated diagram. Handling the x-dimension, this means that the x-coordinates of a pair of nodes are equal. As nodes may split into multiple nodes and may change in size, alignments may not be feasible in the migrated model. Thus we add following term to the objective:

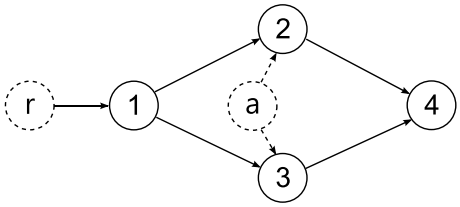
$$\sum_{\{v,w\}\in A} \omega^a_{\{v,w\}} \cdot \mid x_{v'} - x_{w'} \mid, \qquad (5)$$

where $\omega^a_{\{v,w\}}$ denotes a weight indicating the importance of preserving the alignment between $v$ and $w$.

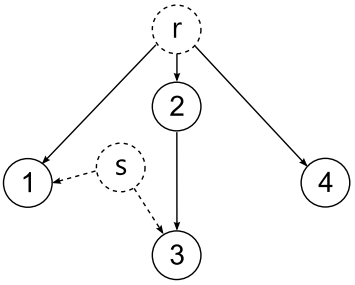
The alignments $A$ can efficiently be calculated using a scanline method similiar to the one outlined in Alg. 1.




(a) Input graph




(b) Constraint graph for x-dimension




(c) Constraint graph for y-dimension

Figure 5: Illustration of the two constraint graphs created for the input graph (a). Both constraint graphs contain a root node $r$. In (b) an additional dummy node $a$ is added to try preserve the x alignment of nodes 2 and 3. In (c) a dummy node $s$ is added to try to keep the edge (1, 3) straight.

As is the case with preserving the orthogonal ordering, it may be worthwhile to refrain from generating alignment constraints between nodes that are far away, or to consider nodes that are aligned w. r. t. their right border as well. Also, since most users won't put too much effort into aligning nodes perfectly, it will be a good idea to allow the alignment to be slightly off for two nodes to still be considered aligned. We leave this for future work though.

*Network Simplex.* To solve the described optimization problem, we use the *network simplex algorithm*, which is the graph-theoretic equivalent to the well-known simplex algorithm for solving linear optimization problems. It always finds an optimum and as opposed to general LP solvers runs fast in practice.

In the context of graph drawing, Gansner et al. described how it can be used to solve the layering step of the layer-based layout approach [11] (cf. Sec. 4.2), and gave a detailed implementation scheme.

Given a graph $G^* = (V^*, E^* \subseteq (V^* \times V^*))$ with weighted edges ($\omega_e \in \mathbb{R}^+$) and a minimum length for each edge ($d_e \in \mathbb{N}_0^+$), find a mapping $l : V \mapsto \mathbb{N}$, such that

$$\sum_{e=(v,w)\in E^*} \omega_e \cdot \ (l(w) - l(v)) \tag{6}$$

is minimized subject to

$$l(w) - l(v) \geq d_{(v,w)} \qquad \forall (v, w) \in E^*. \tag{7}$$

Interpreting the mapping $l$ as new x-coordinates for the nodes, the network simplex algorithm thus seeks for minimal edge lengths of the edges of $E^*$ with respect to the specified weights and minimum edge lengths. The algorithm is guaranteed to find a global optimum.

We now explain how the above optimization problem can be translated into a network simplex instance. See Fig. 5 for an illustration. First, an imaginary root node $r$ is added to $G^*$. Second, for every $v' \in V'$ we add a node $v^*$ to $G^*$. Third, for every constraint between nodes $v'_1$ and $v'_2$ computed by Alg. 1, we add an edge $e = (v_1^*, v_2^*)$ to $G^*$ and set the minimum separation to $d_e = \max(x_{v_2} - x_{v_1}, w'_{v_1} + s_x)$ and $\omega_e = 1$. Fourth, for every node $v^*$ in $G^*$ that has no incoming edge, we add an edge $e$ from $r$ with $d_e = x_v$ and $\omega_e = 1$.

To transform Eq. 5, which tries to preserve node alignments, we have to get rid of the absolute function. Gansner et al. do so by using a dummy node and two edges. For each alignment $a$ between two nodes $(v_1, v_2)$ we add a dummy node $v_a$ to $G^*$ and two edges $e_1 = (v_a, v_1^*)$ and $e_2 = (v_a, v_2^*)$. We set $d_{e_1} = d_{e_2} = 0$ and for both edges can directly transfer the weight $\omega^a$. Since network simplex tries to minimize the edge lengths, it tries to reduce the edges $e_1$ and $e_2$ to a length of zero, hence (left-)aligning $v_1$ and $v_2$. The same construction can be used to model the terms to preserve straight edges described in Eq. 4.

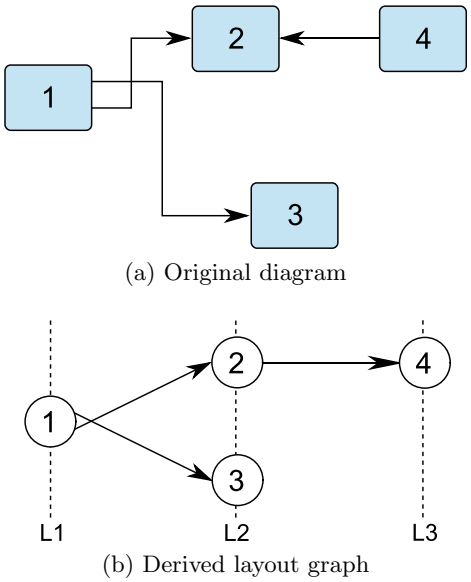
Note that in its current form the method computes new node coordinates only. Edges can either be routed with with dedicate edge routing algorithms [15, 28] or, in case of orthogonal edge routes, the vertical (horizontal) edge segments can be converted into dummy nodes and thus placed by the algorithm as well [23].

## 4.2 Interactive Layer-Based Layout

We now turn to a method that is given slightly more freedom to improve the layout quality of a given diagram. Schulze et al. present methods that are specifically tailored to draw data flow diagrams [25]. Their algorithm is a fully-fledged layout creation algorithm with port support based on the *layer-based* layout method indroduced in 1981 by Sugyiama et al. [27]. Spönemann discussed how *interactivity* can be added to the approach [26]. The main new contribution in this section is how we apply interactive layout to disconnected components, which was not covered in this context before.

Note that interactivity here refers to the algorithm respecting, to a certain extent, existing positions of elements as opposed to user interactivity. See Fig. 6 for a simple example of how the method explained here tries to preserve the diagram's general topology.

The layer-based layout method is a pipeline of five steps, each of which we will now explain, including details on how to respect existing diagram coordinates. In the following, assume that we have a main layout direction from left to right. While literature on the layer-based layout method usually assumes a top-down direction, a left to right direction is customary for dataflow diagrams.



(a) Original diagram

(b) Derived layout graph

Figure 6: Example of how the layered layout algorithm derives a valid layering and node order from an original diagram. Even though the edge crossing could easily be avoided by switching the order of nodes 2 and 3, the layout algorithm chooses not to in order to preserve the diagram's original topology. Also, the edge that connects nodes 2 and 4 was reversed by the cycle breaking phase to yield a proper layering; the original edge direction is restored before layout results are applied.

*Cycle Breaking.* While the original proposal [27] assumed the input graph to be acyclic to be able to have all edges point in the same direction, dataflow diagrams often include feedback edges that form cycles in the graph. The aim of this phase is thus to find a minimum set of edges that, when reversed, removes all cycles from the graph. This is equivalent to solving the NP-complete Minimum Feedback Arc Set (MFAS) Problem [12]. Of course, the original directions of all edges are restored at the end of the layout algorithm.

Given positions of an existing drawing, an acyclic graph can be derived by letting edges point from nodes with lower x-coordinate to nodes with higher x-coordinates. Since in our use case nodes can have (different) sizes, it is not necessarily clear which is the relevant x-coordinate to use. As pointed out by Spönemann one can use the center point, one of the corners, or the point where the edge actually connects to the node via a port. Which strategy to use depends on the application. If several nodes share the same x-coordinate and form a cycle, an MFAS heuristic can be used.

*Layering.* Nodes are assigned to indexed layers such that all edges point from layers with lower index to layers with higher index. This is possible because the previous phase ensured that we have an acyclic graph. Some edges will now connect nodes in adjacent layers, but some edges will span multiple layers. These edges, so-called *long edges*, are split

by introducing *dummy nodes* such that every edge connects nodes in adjacent layers. Different strategies for layer assignment are possible, e. g. minimizing the total edge length or the number of layers.

Given initial positions, layers can again be inferred based on node positions. A problem arises if two nodes are connected, but overlap in the original diagram. Depending on how exactly a layer assignment is inferred from node positions, additional care has to be taken to ensure that connected nodes do not end up in the same layer.

*Crossing Minimization.* The goal of this phase is to minimize the total number of edge crossings. This is done by finding node permutations for each layer such that the number of edge crossings is minimized. Minimizing the number of crossings between a pair of layers is NP-complete [13]. The usual method employed here is a layer-sweep algorithm that sweeps back and forth through all layers, trying to minimize crossings between each pair of adjacent layers at a time.

With given node positions the order of nodes in a layer can be derived in a similar fashion as described before for layer assignment, this time using y-coordinates instead of x-coordinates. One questions that remains is where to place dummy nodes; after all, they are not present in the input graph. Spönemann proposes to use the edge's route in the existing diagram. When preserving the edge routes is of lesser interest it is also possible to derive an order based on y-coordinates solely for the regular nodes, and apply standard crossing minimizing techniques for the dummy nodes, keeping the order of regular nodes constant by using ordering constraints [8]. We call the latter strategy *semi-interactive*.

The three steps discussed so far were only concerned with the topology of the diagram. Making these steps aware of existing node positions as described above yields results where the general topology is approximately preserved. The remaining two phases are concerned with computing actual node coordinates. While it is possible to a certain extent to include existing node coordinates into the remaining phases of the algorithm, we chose not to. We hypothesize that computing new coordinates from scratch (that preserve the general topology) will result in a "cleaned up" layout. Of course, this will not result in a layout where nodes are moved as little as possible. Comparing the results to the network simplex approach will be the subject of Sec. 5.

*Node Coordinate Assignment.* This phase computes explicit y-coordinates for both regular nodes and dummy nodes. Node positions have a direct effect on which edges can be drawn straight and which cannot. The dummy nodes' positions directly affect the edge routes. Possible strategies are, for instance, to achieve a balanced node placement with short edges or trying to maximize the number of straight edges. Which strategy is preferred for dataflow diagrams mostly depends on personal preference: while increasing the number of straight edges is generally desirable, the downside is that doing so will often increase the height of a diagram.

*Edge Routing.* This phase determines final routes of edges depending on the desired edge routing style, and computes x-coordinates of all nodes. Once this phase has finished, edges split by dummy nodes during layer assignment are restored and their bend points are applied.

*Disconnected Components.* One challenge the layered approach to layout faces is that of disconnected components: groups of graph elements where edges always only connect elements that are part of the same group. Each disconnected component will usually model a functionally separate part of the system under development. There are two ways to handle disconnected components in the layer-based approach:

1. Each disconnected component can be regarded as a separate diagram and thus be laid out separately. The finished disconnected components can then be placed to arrive at a final layout.

2. The fact that there are disconnected components is ignored in that all diagram elements are laid out at once.

Both methods yield correct results, but the latter method can lead to unfortunate layouts since all diagram elements are forced into the same set of layers, whether they are related in some way or not. It is thus usually best to apply the former method. In the context of interactive layout, however, this poses the additional challenge that the placement of disconnected components needs to take existing coordinates into account.

Currently we partly follow the first strategy. To add interactivity, we calculate a barycenter for each component from the original positions of the component's nodes and place the center of the newly layouted component at this barycenter. This can result in components overlapping, which we simply solve by moving them apart from each other until no overlaps exist. Overlap-removal methods as discussed in Sec. 3 can be used to improve the procedure.

*Comments.* Just like textual languages, visual languages usually provide support for comments: nodes that contain text to explain how a model works. LabVIEW diagrams, such as the diagram in Fig. 1, are no exception. Which node a comment refers to can be made clear either explicitly through a connection between the two nodes or implicitly through their placement only. The latter case is most common, and can be problematic when using automatic layout algorithms: without knowledge regarding which node a comment refers to, the two may end up in vastly different places.

Since both methods presented in this paper try to preserve a diagram's general topology, they are not as prone to the problem as layout algorithms that compute a new layout from scratch. The latter require further preprocessing to gather information about which comment refers to which node to keep them from being placed too far apart. This process is called *comment attachment* and has already been studied [24].

## 5. CASE STUDY

As mentioned earlier and as seen in Fig. 1, the LabVIEW suite recently introduced a new version of their graphical modeling language. The tool ships with several demo models, 70 of which we use for the following case study.

Recall that we differentiated three strategies S1–3 in Sec. 2 regarding how to migrate the concrete syntax of a model. With S1 being insufficient we are left with S2 (adjusting layout) and S3 (creational layout). The questions we seek to answer are:

Table 1: Results for 70 diagrams with an average of 28.5 [14.9] nodes and 36.3 [22.1] edges. Figures in brackets denote rounded standard deviations. $\bar{\delta}$ denotes the average distance, $\bar{o}_c/\bar{o}_l$ are the constant-/linear-weighted orthogonal ordering metrics. $\bar{a}$ is is the average area occupied by a drawing in pixels, $\overline{ec}$ and $\overline{el}$ are the average number of edge crossings per edge and the average edge length.

| | Similarity Metrics | | | Aesthetic Metrics | | |
|---|---|---|---|---|---|---|
| | $\bar{\delta}$ | $\bar{o}_c$ | $\bar{o}_l$ | $\bar{a}$ | $\overline{ec}$ | $\overline{el}$ |
| NSI | 158.2 [156.0] | 0.03 [0.02] | 0.01 [0.00] | 759,501 [685,204] | 0.9 [0.7] | 197.4 [100.7] |
| LBI | 177.6 [149.4] | 0.09 [0.03] | 0.02 [0.01] | 702,885 [715,863] | 1.1 [0.8] | 186.1 [94.8] |
| CLD | 182.8 [107.1] | 0.21 [0.08] | 0.05 [0.02] | 634,063 [619,592] | 0.5 [0.4] | 144.0 [71.0] |

1. How well do the adjusting layout methods preserve the character of the diagram?

2. How do the adjusting layout methods differ from creational layout methods in terms of "classical" aesthetic criteria?

The two methods presented in Sec. 4.1 and Sec. 4.2, the network simplex-based incremental algorithm (NSI) and the layer-based interactive algorithm (LBI), implement S2. As representative for S3 we use a creational layout method specifically tailored for dataflow diagrams (CLD) presented by Schulze et al. [25].

The quality of drawings produced by creational layout algorithms are evaluated using metrics measuring certain aesthetics criteria, among which the number of edge crossings, the overall drawing area, and the average edge length rank among the more important ones [5]. Layout adjustment techniques can be assessed by measuring the similarity between the original and the adjusted drawing. We use two metrics a user study found to be significant [4]: *average distance* and *orthogonal ordering*. Since the latter two metrics require some further explanations, we provide brief definitions here before turning to the results.

The average distance between two drawings $D$ and $D'$ is the normalized distance of the sum of any pair of node positions in the original and new drawing:

$$\bar{\delta}(D, D') = \frac{1}{|V'|} \sum_{v' \in V'} d(v, v'),$$

where $d$ is some distance function. We used the Euclidean distance between the nodes' center points. Remember that by definition (cf. Sec. 4) there is always a unique $v \in V$ for every $v' \in V'$.

As mentioned by Bridgeman and Tamassia [4] the metric is sensitive to the specific positions of a drawing, i. e. shifting all positions in one drawing by a certain amount of pixels may yield a different result. For this reason the drawings are *aligned* beforehand. Point matching algorithms can be used to translate, scale, and rotate the drawings such that some objective is fulfilled. Here we do not allow rotation and scaling and seek for an alignment such that the average distance is minimized. Furthermore, given a hierarchical diagram, we translate all points into a common coordinate system and align them all at once. Another option would be to calculate the metric for every hierarchy level separately.

Orthogonal ordering is one of the three key features to mental map preservation [19]. The metric given by Bridgeman and Tamassia measures the change in relative positioning between two points in the plane. Let $P$ and $P'$ denote the set of center coordinates of all nodes of $V$ and $V'$. Assume both sets have the same size. The metric takes values in $[0, 1]$, where a 0 indicates no change at all:

$$\bar{o}(P, P') = \frac{1}{W} \sum_{p,q \in P} \min \left\{ \int_{\theta_{pq}}^{\theta_{p'q'}} \omega(\theta) d\theta, \int_{\theta_{p'q'}}^{\theta_{pq}} \omega(\theta) d\theta \right\},$$

where $W$ is a normalization factor, $\omega$ a weighting function and $\theta_{pq}$ the angle between the positive x axis and the vector $q - p$.

Two versions of the metric are differentiated: a) the *constant-weighted* version $\bar{o}_c$ with $W = \pi \cdot |P|$ and $\omega(\theta) = 1$, and b) the *linear-weighted* version $\bar{o}_l$ with $W = \frac{\pi}{2} \cdot |P|$ and

$$\omega(\theta) = \begin{cases} \frac{\theta \bmod \frac{\pi}{2}}{\frac{\pi}{4}} & (\theta \bmod \frac{\pi}{2}) < \frac{\pi}{4} \\ \frac{\frac{\pi}{2} - (\theta \bmod \frac{\pi}{2})}{\frac{\pi}{4}} & \text{otherwise.} \end{cases}$$

The intuition behind the linearly weighted version is to penalize changes of the relative positions between nodes stronger as they are more noticeable to viewers and thus have a larger negative impact on their mental map. For example, if node $a$ was originally to the right of node $b$, it would have a greater effect if it was above $b$ in a new diagram than if it was still to the right of $b$, but had its position only slightly changed upwards.

***Test Diagrams.*** The evaluations are based on 69 diagrams shipping with the LabVIEW suite as examples. The diagrams regularly serve as starting points for users' applications.

Of about 2200 nodes, 33 were split into two or more nodes, and 132 nodes were newly added to the diagrams. The most significant change is that node sizes are about three times larger after migration, with significant variations across the different types of nodes.

***Results.*** For the results of the different methods to be comparable, we configured every method to use a minimum separation of 10 pixels between pairs of nodes. Also, since the methods themselves use different strategies for edge routing, we used the edge routing algorithm by Wybrow et al. [28] after node coordinates have been determined. This allows statements regarding the quality of node placement from the edge routing's perspective.

Tab. 1 summarizes the results for all three methods. As expected, it can be seen that with increasing freedom to position nodes the average distance ($\bar{\delta}$) and orthogonal ordering ($\bar{o}_{c/l}$) metrics increase as well. For NSI the orthogonal ordering metric is almost 0, indicating a strong preservation of a diagram's overall look. Interestingly, the linear-weighted

version of the orthogonal ordering metric ($\bar{o}_l$) of CLD is very small (0.05), compared to 0.21 for the constant-weighted version. This is very likely due to the nature of dataflow diagrams having a clear flow from left to right and thus already prescribing most of the horizontal relations of the nodes. It also indicates that even though no previous positions are considered, the overall look of the diagram is preserved. On the contrary, the aesthetic metrics improve with increasing freedom to place nodes. The CLD method produces smaller drawings with less edge crossings and shorter edges, on average.

Executed on an up-to-date laptop all three methods finish in well under 100ms for every of the tested diagrams.

We conclude that every method has its strengths and it depends on the specific application and objective which one to use.

# 6. DISCUSSION

As mentioned, automatic layout algorithms can be designed to fulfill different goals: preserving a diagram's topology yields different results than computing a new layout from scratch. The methods presented in this paper are each optimized towards a subset of possible goals, as the case study shows: while network simplex-based incremental layout tries to keep node movement to a minimum and only resolves obviously invalid layouts, interactive layer-based layout tends to "clean up" diagrams by pulling diagram elements closer together. Both preserve the diagram's topology. Fully automatic layout, on the other hand, computes everything from scratch and thereby for example minimizes the number of edge crossings by choosing not to preserve the existing topology. With this in mind, the question to answer is which method is superior. That question can only be answered for each concrete use case independently.

We believe that the main factor to influence this decision should be the question of how much the user can be expected to accept diagram changes in a given use case. The main guideline we see here is whether the layout run was explicitly requested by the user, or was triggered implicitly by the tool. The network simplex-based incremental layout works best for migration scenarios, such as the one we focused on in this paper. Since diagram migration can be considered a forced necessity resulting from language or tool migration, it seems sensible to change existing diagrams as little as possible. Another use case this behavior is required for is interactive editing, such as the user adding or resizing nodes. The tool can help moving adjacent nodes to accommodate the new or enlarged node. Larger changes should only be applied if the user explicitly requests them. This is where interactive layer-based and fully automatic layout can be offered as two degrees of layout: the former only cleans up the diagram while the latter produces a new, optimized layout regardless of the diagram's current state.

Of course, it is not only the scenario that impacts the decision of which method to use. Technical restrictions may also impact which methods can be applied in the first place. For example, if a tool forces nodes to be aligned along a regular grid, a layout algorithm needs to support that requirement. From the methods presented here, only network simplex-based incremental layout does so. Note that this actually enables another use case: that of making non-grid-aligned graphs grid-aligned with as little changes as possible.

The network simplex-based method can also be adapted to other types of diagrams in that it, without any modification, will preserve the relative positions of nodes of any graph it is given. Diagrams such as UML class and activity diagrams are good examples for diagrams that can be handled directly. Addressing diagram-specific requirements is then a question of leaving out, or adding, constraints.

# 7. REFERENCES


[1] U. Brandes, M. Eiglsperger, M. Kaufmann, and D. Wagner. Sketch-driven orthogonal graph drawing. In *Proceedings of the 10th International Symposium on Graph Drawing (GD'02)*, volume 2528 of *LNCS*, pages 1–11. Springer, 2002.

[2] S. Bridgeman and R. Tamassia. Difference metrics for interactive orthogonal graph drawing algorithms. In *Graph Drawing*, pages 57–71. Springer, 1998.

[3] S. S. Bridgeman, J. Fanto, A. Garg, R. Tamassia, and L. Vismara. InteractiveGiotto: An algorithm for interactive orthogonal graph drawing. In *Graph Drawing (Proc. GD'97)*, volume 1353, pages 303–308. Springer, 1997.

[4] S. S. Bridgeman and R. Tamassia. A user study in similarity measures for graph drawing. *Journal of Graph Algorithms and Applications*, 6(3):225–254, 2002.

[5] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999.

[6] T. Dwyer, K. Marriott, and P. J. Stuckey. Fast node overlap removal. In P. Healy and N. S. Nikolov, editors, *Proceedings of the 13th International Symposium on Graph Drawing (GD'05)*, volume 3843 of *LNCS*, pages 153–164. Springer, 2006.

[7] T. Dwyer, K. Marriott, and M. Wybrow. Topology preserving constrained graph layout. In *Revised Papers of the 16th International Symposium on Graph Drawing (GD'08)*, volume 5417 of *LNCS*, pages 230–241. Springer, 2009.

[8] M. Forster. Applying crossing reduction strategies to layered compound graphs. In *Proceedings of the 10th International Symposium on Graph Drawing (GD'02)*, volume 2528 of *LNCS*, pages 115–132. Springer, 2002.

[9] K. Freivalds and P. Kikusts. Optimum layout adjustment supporting ordering constraints in graph-like diagram drawing. volume 55 of *Proceedings of the Latvian Academy of Sciences*, pages 43–51. 2001.

[10] E. R. Gansner and Y. Hu. Efficient node overlap removal using a proximity stress model. In I. G. Tollis and M. Patrignani, editors, *Graph Drawing*, pages 206–217, Berlin, Heidelberg, 2009. Springer-Verlag.

[11] 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.

[12] 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.

[13] M. R. Garey and D. S. Johnson. Crossing number is NP-complete. *SIAM Journal on Algebraic and Discrete Methods*, 4(3):312–316, 1983.

[14] K. Hayashi, M. Inoue, T. Masuzawa, and H. Fujiwara. A layout adjustment problem for disjoint rectangles

preserving orthogonal order. In *Graph Drawing*, pages 183–197. Springer, 1998.

[15] D. W. Hightower. A solution to line-routing problems on the continuous plane. In *Proceedings of the 6th Annual Design Automation Conference*, DAC '69, pages 1–24, New York, NY, USA, 1969. ACM.

[16] X. Huang, W. Lai, A. Sajeev, and J. Gao. A new algorithm for removing node overlapping in graph visualization. *Information Sciences*, 177(14):2821–2844, 2007.

[17] W. Li, P. Eades, and N. Nikolov. Using spring algorithms to remove node overlapping. In *Proceedings of the 2005 Asia-Pacific symposium on Information visualisation-Volume 45*, pages 131–140. Australian Computer Society, Inc., 2005.

[18] K. Marriott, P. Stuckey, V. Tam, and W. He. Removing node overlapping in graph layout using constrained optimization. *Constraints*, 8(2):143–171, Apr. 2003.

[19] K. Misue, P. Eades, W. Lai, and K. Sugiyama. Layout adjustment and the mental map. *Journal of Visual Languages & Computing*, 6(2):183–210, June 1995.

[20] S. C. North and G. Woodhull. Online hierarchical graph drawing. In *Revised Papers of the 9th International Symposium on Graph Drawing*, volume 2265 of *LNCS*, pages 232–246. Springer, 2002.

[21] R. F. Paige, N. Matragkas, and L. M. Rose. Evolving models in Model-Driven Engineering: State-of-the-art and future challenges. *Journal of Systems and Software*, 111:272 – 280, 2016.

[22] L. M. Rose, M. Herrmannsdoerfer, S. Mazanek, P. Van Gorp, S. Buchwald, T. Horn, E. Kalnina, A. Koch, K. Lano, B. Schätz, and M. Wimmer. Graph and model transformation tools for model migration. *Software & Systems Modeling*, 13(1):323–359, 2012.

[23] U. Rüegg, C. D. Schulze, D. Grevismühl, and R. von Hanxleden. Using one-dimensional compaction for smaller graph drawings. In *Proceedings of the 9th International Conference on the Theory and Application of Diagrams (DIAGRAMS'16)*, 2016.

[24] C. D. Schulze, C. Plöger, and R. von Hanxleden. On comments in visual languages. In *Proceedings of the 9th International Conference on the Theory and Application of Diagrams (DIAGRAMS'16)*, 2016.

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

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

[27] 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.

[28] M. Wybrow, K. Marriott, and P. J. Stuckey. Orthogonal connector routing. In *Proceedings of the 17th International Symposium on Graph Drawing (GD'09)*, volume 5849 of *LNCS*, pages 219–231. Springer, 2010.