

# Model Order

Reconciling Automatic Layout and User Intentions

Msc. Sören Domrös  
geb. in Lübeck

Dissertation  
zur Erlangung des akademischen Grades  
Doktor der Ingenieurwissenschaften  
(Dr.-Ing.)  
der Technischen Fakultät  
der Christian-Albrechts-Universität zu Kiel  
eingereicht im Jahr 2025

Kiel Computer Science Series (KCSS) 2025/3 dated July 16, 2025

ISSN 2193-6781 (print version)

ISSN 2194-6639 (electronic version)

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

The author can be contacted via [soed.privat@gmail.com](mailto:soed.privat@gmail.com)

Published by the Department of Computer Science, Kiel University

Real-time and Embedded Systems Group

Please cite as:

- ▷ Sören Domrös. *Model Order – Reconciling Automatic Layout and User Intentions* Number 2025/3 in Kiel Computer Science Series. Department of Computer Science, 2025. Dissertation, Faculty of Engineering, Kiel University.

```
@book{Domroes25,  
  author   = {Sören Domrös},  
  title    = {Model Order -- Reconciling Automatic Layout and User Intentions},  
  publisher = {Department of Computer Science, Kiel University},  
  year     = {2025},  
  number   = {2025/3},  
  doi      = {10.21941/kcss/2025/3},  
  series    = {Kiel Computer Science Series},  
  note     = {Dissertation, Faculty of Engineering,  
              Kiel University.}  
}
```

© 2025 by Sören Domrös

# About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit <http://www.informatik.uni-kiel.de/kcss> for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter: Prof. Dr. Reinhard von Hanxleden  
Christian-Albrechts-Universität zu Kiel  
Kiel, Germany
2. Gutachter: Prof. Dr. Partha Roop  
University of Auckland  
Auckland, New Zealand

Datum der mündlichen Prüfung: 04.07.2025

# Zusammenfassung

Automatische Layoutalgorithmen ermöglichen es aus textuellen Modellen automatisch visuelle Repräsentationen, sogenannte Diagramme, zu generieren. Da aber Layoutalgorithmen üblicherweise nur die Ästhetik von Diagrammen optimieren, befinden sich Diagrammelemente nicht zwangsläufig dort wo Nutzer:innen sie erwarten, sondern dort wo sie ebendiese Ästhetikmetriken optimieren, zum Beispiel um möglichst wenige Kantenkreuzungen zu erzeugen. Das Problem daran ist, dass Layoutalgorithmen dabei nicht die *Stabilität* und die *Intention* hinter einem Layout berücksichtigen und Nutzer:innen keine *Kontrolle* über das Layout erlauben.

Als Lösung schlage ich vor, Layoutalgorithmen durch die Nutzung der Reihenfolge im textuellen Modell zu erweitern, der *Model Order*. Diese Model Order erlaubt durch das textuelle Modell Kontrolle über das Layout auszuüben, überführt die Intention im textuellen Modell in das Diagramm und verbessert die Stabilität, indem deterministische und kontrollierbare Layoutentscheidungen getroffen werden.

In dieser Arbeit präsentiere ich erstens einen *Rechteckpackalgorithmus*, welcher die Model Order der gegebenen Rechtecke als Lesereihenfolge nutzt. Zweitens zeige ich, wie Model Order in den verschiedenen Phasen des Sugiyama bzw. *Layered-Algorithmus* als Bedingung oder als Entscheidungshilfe genutzt werden kann. Drittens zeige ich, wie die Model Order sich auf das *Packen separater Komponenten* mit Verbindungen nach außen auswirkt. Da Layouts nicht komplett durch die textuelle Ordnung kontrolliert werden können, zeige ich zusätzlich, wie Model Order und Layoutbedingungen miteinander interagieren. Diese Algorithmen evaluiere ich anhand der synchronen Sprache *SCCharts* und der polyglotten Koordinationssprache *Lingua Franca* und wende die gewonnenen Erkenntnisse über Model Order-Algorithmen auf andere Modellierungssprachen an.

Die präsentierten Algorithmen sind Teil der Open-Source Layoutbibliothek ELK und werden in der Praxis genutzt.



# Abstract

Automatic layout algorithms are a key enabler to use graphical representation of modeling languages, so-called diagrams, effectively. However, layout algorithms typically only focus on readability of the diagrams, using metrics such as the number of edge crossings, the number of backward edges, or required area. Therefore, algorithms do not place elements where a user expects them to be but rather where they result in a better value for a readability metric. Moreover, algorithms often disregard user requirements such as *stability*, *secondary notation*, and *control* over the layout to optimize readability metrics.

I propose to solve this by augmenting layout algorithms with the order of the underlying model, the *model order*. This model order gives developers control over the layout and transfers textual secondary notation into the diagram while at the same time improving stability by making controllable and deterministic layout decisions.

First, I present a *rectangle packing* algorithm that preserves the reading direction given by the rectangle model order while optimizing readability. Second, I illustrate how model order can be utilized during the different phases of the Sugiyama or *layered algorithm* to preserve user intentions. Third, I investigate *packing of separate connected components* with connections to the outside while preserving model order. Fourth, I present *diagram interaction techniques* and their interaction with model order. Finally, I evaluate the presented model order algorithms using the synchronous language *SCCharts* and the polyglot coordination language *Lingua Franca* to discuss model order strategies resulting in model order layout configurations to be used by these and other modeling languages.

The presented algorithms and strategies are part of the open-source graph drawing library *ELK* and are used in industry.





# Contents

<b>1</b>	<b>Introduction</b>	<b>1</b>
1.1	Contributions and Outline . . . . .	3
<b>2</b>	<b>Preliminaries</b>	<b>7</b>
2.1	The Required Editing Paradigm . . . . .	7
2.2	SCCharts . . . . .	10
2.3	Lingua Franca . . . . .	13
<b>I</b>	<b>Model Order</b>	<b>19</b>
<b>3</b>	<b>User Intentions and the Concept of Model Order</b>	<b>21</b>
3.1	User Intentions and Model Order . . . . .	22
3.2	Readability and Model Order . . . . .	23
3.3	Stability and Model Order . . . . .	24
3.4	Secondary Notation and Model Order . . . . .	26
3.5	Control, Model Order, and Automatic Layout . . . . .	27
3.6	Formalizing Model Order . . . . .	29
<b>4</b>	<b>Model Order in Rectangle Packing</b>	<b>33</b>
4.1	The Region Packing Problem . . . . .	34
4.1.1	Scale Measure . . . . .	35
4.1.2	Whitespace Elimination . . . . .	35
4.1.3	Row Structure and Reading Direction . . . . .	37
4.2	Related Work . . . . .	41
4.3	A Simple Heuristic . . . . .	44
4.4	Rectpacking Heuristic . . . . .	47
4.4.1	Rectpacking Width Approximation Step . . . . .	48
4.4.2	Rectpacking Placement and Grouping Step . . . . .	52

## Contents

4.4.3	Rectpacking Compaction Step . . . . .	55
4.4.4	Worst Case Rectpacking . . . . .	61
4.4.5	Improving the Compaction . . . . .	63
4.4.6	Rectpacking Whitespace Elimination Step . . . . .	69
4.5	Solving Region Packing Optimally . . . . .	73
4.6	Evaluating the Region Packing Problem . . . . .	78
4.6.1	Region Packing in Hierarchical Graphs . . . . .	84
4.7	Remaining Obviously Non-Optimal (ONO) Region Packings . . . . .	87
4.7.1	Greedy Placement Hinders Compaction . . . . .	87
4.7.2	Greedy Block Creation hinders Compaction . . . . .	87
4.7.3	Wrong Stack Alignment . . . . .	89
4.8	Summary . . . . .	90
<b>5</b>	<b>Model Order in Layered Layout</b>	<b>93</b>
5.1	Layered Layout . . . . .	95
5.1.1	Layered Algorithm in the Eclipse Layout Kernel . . . . .	96
5.1.2	The Cycle Breaking Problem . . . . .	98
5.1.3	The Layer Assignment Problem . . . . .	101
5.1.4	The Crossing Minimization Problem . . . . .	101
5.1.5	The Node Placement Problem . . . . .	103
5.1.6	The Edge Routing Problem . . . . .	103
5.1.7	Random Decisions in Automatic Layout . . . . .	104
5.2	Related Work . . . . .	105
5.2.1	Layered Layout Constraints and Model Order . . . . .	106
5.2.2	Order in Layout Algorithms for Node-Link Diagrams . . . . .	108
5.3	Model Order and Cycle Breaking . . . . .	111
5.3.1	Strict Model Order Cycle Breaking . . . . .	111
5.3.2	Greedy Model Order Cycle Breaking . . . . .	113
5.3.3	DF and BF Model Order Cycle Breaking . . . . .	117
5.3.4	Summarizing Model Order Cycle Breaking . . . . .	119
5.4	Model Order in Layer Assignment . . . . .	119
5.4.1	Tie-Breaking Model Order during Layer Assignment . . . . .	119
5.4.2	Breadth-First Model Order Layer Assignment . . . . .	120
5.4.3	BF Model Order Layer Assignment by Post-Processing . . . . .	124
5.4.4	Depth-First Model Order Layer Assignment . . . . .	126

5.4.5	Summarizing Model Order Layer Assignment . . . . .	129
5.5	Model Order in Crossing Minimization . . . . .	130
5.5.1	Leveraging Model Order for Crossing Minimization .	131
5.5.2	Model Order Crossing Minimization by Pre-Sorting .	135
5.5.3	Model Order Barycenter Heuristic . . . . .	145
5.5.4	Crossing Minimization Weighted by Model Order . . .	147
5.5.5	Full Control Model Order Crossing Minimization . . .	150
5.5.6	Summarizing Model Order Crossing Minimization . .	151
5.6	Model Order in Node Placement . . . . .	152
5.7	Group Model Order for Layered Layouts . . . . .	153
5.7.1	Group Model Order in Cycle Breaking . . . . .	154
5.7.2	Group Model Order in other Layout Phases . . . . .	159
5.8	Solving Layered Topology Efficiently using Model Order . . .	159
5.9	Evaluating the Layered Algorithm . . . . .	164
5.9.1	Study 1: Qualitative Evaluation of SCCharts Cycle Breaking . . . . .	164
5.9.2	Study 2: Quantitative Evaluation of SCCharts Cycle Breaking . . . . .	169
5.9.3	Study 3: Quantitative Evaluation of Lingua Franca Cycle Breaking . . . . .	170
5.9.4	Study 3 continued: Quantitative Evaluation of Lingua Franca Crossing Minimization . . . . .	174
5.9.5	Study 4: Model Order Metric and its Effect on Edge Crossings for SCCharts . . . . .	176
5.9.6	Study 5: Model Order for Lingua Franca by Experts .	180
5.9.7	Study 6: Model Order for Lingua Franca by Students .	184
5.9.8	Study 7: Model Order for SCCharts in Practice . . . .	201
<b>6</b>	<b>Model Order in Component Packing</b>	<b>203</b>
6.1	The Component Packing Problem . . . . .	204
6.2	Component Packing by Schulze . . . . .	205
6.3	Model Order Component Packing . . . . .	207
6.4	Summarizing Component Packing . . . . .	210

<b>II</b>	<b>Interactivity</b>	<b>213</b>
<b>7</b>	<b>An Interactive Constraints Framework</b>	<b>215</b>
7.1	Interactive Layered Layout . . . . .	219
7.2	Interactive Rectpacking . . . . .	222
7.3	Generalizing Interactivity and Model Order . . . . .	222
<b>8</b>	<b>Structure-Based Editing</b>	<b>225</b>
8.1	Model Order in Structure-Based Editing . . . . .	226
<b>III</b>	<b>Application</b>	<b>229</b>
<b>9</b>	<b>Model Order in Practice</b>	<b>231</b>
9.1	Model Order for SCCharts . . . . .	231
9.1.1	The SCCharts Layered Problem . . . . .	232
9.1.2	The SCCharts Region Packing Problem . . . . .	238
9.2	Model Order for Lingua Franca . . . . .	239
9.2.1	Lingua Franca Cycle Breaking . . . . .	240
9.2.2	Lingua Franca Crossing Minimization . . . . .	243
9.2.3	Lingua Franca Component Packing . . . . .	246
9.2.4	Lingua Franca and Hierarchy-Aware Layout . . . . .	246
9.3	Model Order for Other Modeling Languages . . . . .	249
9.3.1	Rectangle Packing for Modeling Languages . . . . .	249
9.3.2	Cycle Breaking for Modeling Languages . . . . .	250
9.3.3	Layer Assignment for Modeling Languages . . . . .	252
9.3.4	Crossing Minimization for Modeling Languages . . . . .	253
9.3.5	Component Packing for Modeling Languages . . . . .	255
9.4	Lessons Learned . . . . .	257
9.4.1	How To Get Good Models? . . . . .	257
9.4.2	How To Find Developers? . . . . .	259
9.4.3	How To Get Good Feedback? . . . . .	260
<b>10</b>	<b>Conclusion and Future Work</b>	<b>263</b>
10.1	Summarizing Model Order in Rectangle Packing . . . . .	263
10.2	Summarizing Model Order in Layered Layout . . . . .	265

10.3	Summarizing Model Order in Component Packing . . . . .	267
10.4	Open Problems and Future Work . . . . .	267
10.4.1	Fuzzy Target Width for the Rectpacking Heuristic . . . . .	268
10.4.2	Constraints for the Rectpacking Heuristic . . . . .	268
10.4.3	Deterministic Crossing Minimization . . . . .	269
10.4.4	Dangling Nodes, Hyperedges, and Feedback Edges . . . . .	269
10.4.5	Width Approximation for Component Packing . . . . .	270
10.4.6	Compaction for Component Packing . . . . .	271
10.4.7	Improving Structure-Based Editing . . . . .	272
10.4.8	Automatic Model Obfuscation . . . . .	272
10.4.9	Intention Detection via Text Processing . . . . .	273
10.4.10	Configuring Layout via Machine Learning . . . . .	274
10.4.11	Evaluating more Modeling Languages . . . . .	276
<b>A</b>	<b>Publications</b>	<b>279</b>
A.1	Relevant Publications . . . . .	279
A.2	Relevant Advised Theses . . . . .	281
A.3	Additional Related Publications . . . . .	283
<b>B</b>	<b>Abbreviated Models</b>	<b>285</b>
B.1	load_balancer . . . . .	285
B.2	SleepingBarber . . . . .	286
B.3	reactionOrder . . . . .	286
B.4	Chrono . . . . .	287
	<b>Bibliography</b>	<b>303</b>



# Introduction

Automatic layout rises in popularity<sup>1</sup> and is often integrated in modern diagram tools such as yEd<sup>2</sup> or mermaid<sup>3</sup>. However, What You See Is What You Get (WYSIWYG) editors or graphical editors that place the burden of layout on the user [Pet95] and rarely employ automatic layout are still very common. One advantage of manual layout is that users can very directly *control the secondary notation* [Pet95], i.e., the alignment, grouping and ordering that emphasizes the meaning, of the diagram. Moreover, *stability*, i.e., elements moving unexpectedly on change, is typically only an issue when using automatic layout. Controlling and creating the layout of models exactly as one wants it to be might often be more desirable than using an algorithm that optimizes *readability* criteria such as edge crossings or the number of backward edges. As Taylor reported for WYSIWYG type-setting: “People like having feedback and control” [Tay96].

Automatic layout based on a textual modeling language typically lacks secondary notation, stability, and control [Pet95]. One approach to give users control over the layout are layout constraints [BP90], which are typically specified via annotations or a separate user interface. However, using layout constraints requires knowledge on how the underlying layout algorithm works, which users typically do not have, and again adds the burden of manually changing the layout by expressing these constraints.

I propose adding control over secondary notation to automatic layout algorithms using information that is already given to us by the user: A layout algorithm that layouts the model just as it is written does not require

---

<sup>1</sup>See <https://npmrends.com/elkjs> for the usage of the automatic layout library elkjs.

<sup>2</sup><https://www.yworks.com/products/yed>

<sup>3</sup><https://mermaid.js.org/>

## 1. Introduction

additional manual effort and is able to control the layout using intention expressed by the textual order of the modeling language, the *model order*.

Without constraints, layout algorithms typically optimize readability criteria instead of focusing on secondary notation, stability, and control, or considering model order. This may be the case since unorderedness is rooted in the description of layout problems, which typically describe a graph, a layout may consist of, as an *unordered* set of nodes and an *unordered* set of edges. However, order matters if a graph describes the visual representation of a modeling language, as seen in Figure 1.1, and will also matter for an algorithm that iterates over some set.

Figure 1.1 depicts the Source-Through Lingua Franca model, which is a simple data-flow model that causes a causality loop in Lingua Franca. Lingua Franca model developers write their programs textually, as seen in the abbreviated textual model in Figure 1.1a. Figure 1.1b and Figure 1.1c show two automatically synthesized visual representations of the textual model. Both drawings have the same number of edge crossings and backward edges, which are typically optimized and used to measure the readability of a diagram. Hence, if one does not consider order, a layout algorithm may arbitrarily put the Source box at the end of the model, as seen in Figure 1.1b. The developer of the given model, however, cares about the order and expressed the desire for the ordering depicted in Figure 1.1c in the textual model in Figure 1.1a. Here, the Source *s* is defined before the Through *t*.

Hence, I propose to combine the benefits of having control to express secondary notation and stable layouts with the benefits of using automatic layout. I argue that a key to this is to consider the model order for the layout of modeling languages with visual representations. This leads to the following research questions:

*IMPL* How can model order be integrated into automatic layout for visual presentations of modeling languages?

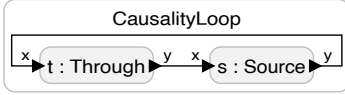
*EVAL* How does model order interact with the layout goals readability, stability, secondary notation, and control for visual presentations of modeling languages?



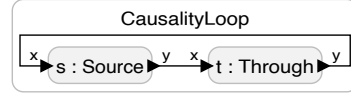
## 1.1. Contributions and Outline

```
main reactor {  
  s = new Source();  
  t = new Through();  
  s.y -> t.x;  
  t.y -> s.x;  
}
```

(a) Abbreviated textual model



(b) ONO layout



(c) Obvious Yet Easily Superior (OYES) layout

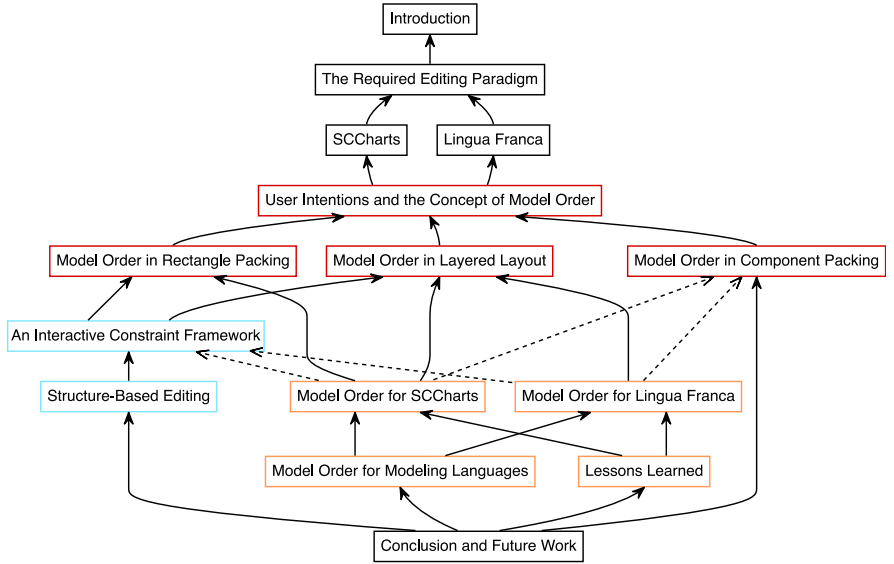
**Figure 1.1.** The Source-Through Lingua Franca model created by Edward A. Lee with an abbreviated textual model in Figure 1.1a and two different graphical models in Figure 1.1b and Figure 1.1c.

## 1.1 Contributions and Outline

The reading dependency structure of this work can be seen in Figure 1.2. Here, dependency edges connect chapters and important sections if one should read a particular part of this work to understand a succeeding part. Additionally, the three parts of the dissertation are highlighted in red, light-blue, and orange. Chapters that only weakly depend on another chapter are connected by dotted edges. This figure shows multiple paths through the following work and highlights potential chapters to skip.

Chapter 2 introduces the editing paradigm that enables utilizing model order and introduces two example languages, SCCharts and Lingua Franca. Part I answers research question *IMPL*—how to integrate model order into layout algorithms—by proposing model order strategies for different algorithms. These are evaluated to answer *EVAL*—how does model order influence layout goals. Since layout constraints are an established way to control a model, Part II covers diagram interaction to express intention and its interaction with model order. Part III generalizes the findings regarding model order on the example of SCCharts and Lingua Franca and suggests concrete algorithms and strategies for general modeling languages. Finally,

## 1. Introduction



**Figure 1.2.** The dependency structure of the dissertation chapters. Part I is highlighted in red, Part II in light-blue, Part III in orange, and dashed edges visualize weak dependencies.

Chapter 10 concludes the paper and suggests future work regarding model order and preserving user intentions in automatic layout.

More concretely, I contribute the following insights about model order and the layout for modeling languages with a visual presentation.

## Controlling the Layout via Model Order

Chapter 3 presents how model order relates to layout goals on a meta level. This chapter discusses how model order relates to readability, secondary notation, stability, and control over the layout [DH24c] serving as an introduction to model order in preparation for the detailed discussion based on concrete algorithms in the following chapters.

## Model Order for Rectangle Packing

Chapter 4 presents the first application of model order as a constraint on the example of the *region packing problem* for the layout of SCCharts regions. A region packing for SCCharts should preserve the reading direction given by the model order to increase stability while packing rectangles to optimize the scale measure—the scaled size of the packing given the aspect ratio of the screen [DLH+21; DLH+23].

In this chapter, I illustrate how a *simple heuristic* creates a feasible packing that preserves the reading direction but fails to create a good scale measure. This motivates the *rectpacking heuristic* while highlighting the need for stability and control. I present the different phases of the rectpacking heuristic, analyze their complexity, and illustrate possible improvements, worst-case packing effectiveness, and ONO-cases, i. e., packings that are still obviously not optimal using the rectpacking heuristic. To evaluate the heuristic, I contribute a maximization problem that describes the optimal solution to the region packing solution problem. The evaluation covers a comparison of all present heuristics, their improvements, and the optimal solution on the basis of generated packing problems, which emulate SCCharts regions, and hierarchical SCCharts models.

## Model Order for Layered Layout

Chapter 5 presents how node-link diagrams that use the Sugiyama [STT81] or layered layout algorithm—named due to its division of the graph into layers—may utilize model order. I contribute how the cycle breaking [Rie22; DRv23], layer assignment [DRv23], crossing minimization [DH22; DRv23; Rie24], and node placement steps can employ model order as a tie-breaker or a constraint. Motivated by the example of Lingua Franca, I contribute how *group model order* can be considered for models with multiple partial orders.

User studies regarding readability, secondary notation, and model order as well as static analysis regarding model order illustrate how users express secondary notation in text and diagram and how model order affects readability and stability.

## 1. Introduction

### **Model Order for Separate Component Packing**

Chapter 6 extends the packing of separate connected components with edges to the “outside” proposed by Schulze [Sch19b] to use model order, which contributes the missing pieces for the layout of SCCharts and Lingua Franca. I show how an existing layout algorithm for packing components with edges to the outside can adapt model order as a constraint to preserve the reading direction and discuss the effect on readability.

### **Layout Constraints, Diagram Interaction, and Model Order**

Part II focuses diagram interaction and how it expresses intention. Chapter 7 contributes how the rectpacking heuristic and layered layout [Pet19; Sch19a; PDS+23] can be extended by constraints. Additionally, I present how layout constraints interact with model order as a mechanism to exert control.

### **Structure-Based Editing and Model Order**

Chapter 8 presents how structure-based graphical editing allows expressing intent via diagram interaction while still relying on automatic layout [Jöh22]. Additionally, I contribute how such diagram interaction may be translated to model order.

### **Model Order for SCCharts and Lingua Franca**

Part III contributes how to apply model order to SCCharts, Lingua Franca, and other languages. Chapter 9 discusses how and why SCCharts and Lingua use model order and its various strategies including lessons learned. The discussion shows how other languages might use model order and how configurations for model order layout algorithms can be extracted.

# Preliminaries

Not all modeling languages can effectively use model order to improve their graphical representation. To use model order effectively, the ordering in the textual model must express intention. Hence, a suitable modeling language needs a—preferably textual—model that carries intention.

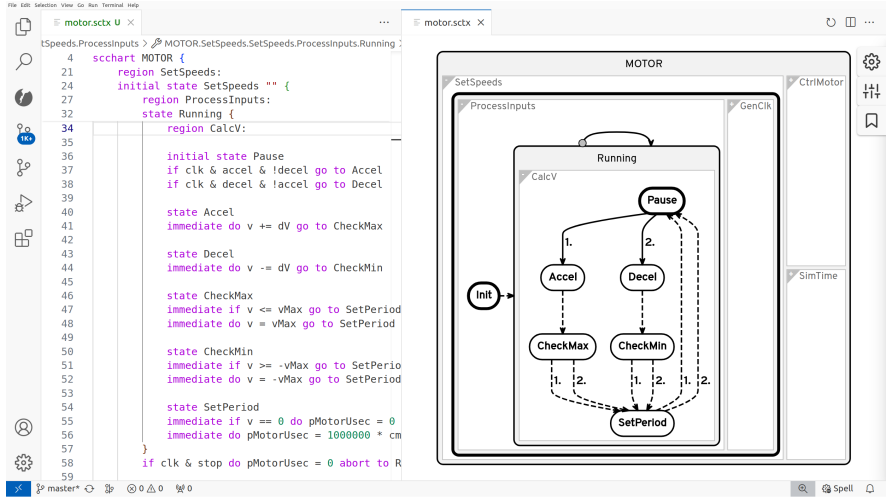
In this chapter, I introduce the required editing paradigm and tooling features that result in an intentional model order. As a running example, I introduce the SCCharts and Lingua Franca modeling languages and their graphical notations and illustrate how these modeling languages qualify for the effective use of model order. In later chapters, SCCharts and Lingua Franca, additionally, serve as a running example and motivation for layout algorithms that employ model order.

## 2.1 The Required Editing Paradigm

To successfully answer the research questions on how model order can be integrated in layout algorithms (*IMPL*) and how model order interacts with the layout goals readability, stability, secondary notation, and control (*EVAL*), I focus on modeling languages that have a graphical representation and a textual model that carries intention. Languages that employ *modeling pragmatics* [HLF+22] and utilize a textual editor and a diagram side-by-side, as seen in Figure 2.1, qualify for this.

Using modeling pragmatics allows developers to have state-of-the-art textual editing support in form of content assist, finding references, standard version management, and syntax highlighting, as it is the case for the

## 2. Preliminaries



**Figure 2.1.** The motor SCChart in the KIELER Visual Studio Code (VS Code) extension with the textual model on the left and the diagram with hidden sidebar (cog symbol) on the right.

SCCharts<sup>1</sup> in Figure 2.1 depicted in VS Code. An additional diagram view provides an overview of the model and can be configured to the current use-case. In particular, the view can be filtered to show only relevant information without hindering the editing capabilities of the textual editor, as it is often the case for diagram-first tools [KRD+24].

Three aspects of modeling pragmatics allow languages that use modeling pragmatics to employ model order since they make sure that the user directly interacts with the model to pour intention into it.

First, the textual model is the ground truth of the model and is primarily edited. Additionally, the textual model should be in a human-readable format, e.g., not XML or XML in a project archive-file but rather a Domain Specific Language (DSL). This is essential since such an unreadable format will most likely never be interacted with and can hence not be changed

<sup>1</sup>For better readability of the figures, I will shorten all SCCharts transition guards to only show their priority throughout this work.

## 2.1. The Required Editing Paradigm

directly. A human-readable primarily edited textual model instead allows us to extract developer-intention poured into the textual model. If algorithms or diagram interaction generate the textual model, the order of elements in it typically does not express intention but rather shows us the creation order of elements.<sup>2</sup> Similarly, if users manually set layout constraints or positions of elements as part of the model development, the ordering information in the textual model may hold less value since it does not influence the layout. Additionally, if the textual model has a form that is not human-readable, humans would rarely edit it and more importantly not adjust the model to express intention.

Second, diagram and text are tightly coupled. The diagram updates immediately using automatic layout once the textual model changes. Hence, the underlying layout algorithm needs to be fast enough if automatic layout should be continuously employed to update the drawing. Moreover, textual elements map to the corresponding diagram elements and the other way around to further allow users to use the textual or the graphical model representation depending on their use-case.

Third, free placement of diagram elements should not be allowed. The only options to control the layout should be by using and configuring a layout algorithm to keep text and diagram consistent. E. g., the layout only changes by changing layout diagram options via the hidden sidebar in Figure 2.1 (see cog symbol on the right), by interacting with the diagram to configure the layout using the textual model as the ground truth (see Chapter 7), or by changing the textual model.

Most tools employed today do not fulfill these requirements. A typical example is Ptolemy II [Pto14], which is an actor-oriented framework supporting heterogeneous, concurrent modeling and design. First, the editing paradigm is diagram-first and involves dragging elements from a palette into a diagram view. Second, the underlying textual model is an XML-file, which is not human-readable. Additionally, the XML-file stores placement data, which further hinders comprehension and manual editing with ordering intention in mind. Third, the order in the textual file only corresponds to the creation order of elements. Fourth, automatic layout can only be invoked

---

<sup>2</sup>Chapter 8 further dives into how one might change the model order while using visual programming techniques.

## 2. Preliminaries

on demand, severing the link between textual and graphical model. To summarize, the textual model of Ptolemy models does not have meaningful ordering information.

This also holds for the commercial Simulink tool. Here, a user again uses palettes or commands to create new elements. Moreover, the user typically only uses an abstracted graphical view on the model but newer interacts with the archive file that holds the model as an XML-file, which only contains elements ordered by their creation order. Furthermore, the automatic layout feature is here typically only used on demand similar to Ptolemy II. Finally, the automatic layout can typically not leverage its full potential since the Simulink-actors have predefined anchor points for edges, so-called ports, such that edges cannot be freely reordered, which constraints most of the layout graph.

However, the editing paradigm of SCCharts (see Section 2.2) and Lingua Franca (see Section 2.3) presented below qualify for the usage of model order since they employ the concept of modeling pragmatics [HLF+22] using the best of both textual and graphical modeling.

### 2.2 SCCharts









SCCharts [HDM+14] is a statechart [Har87] dialect to model safety-critical reactive systems. SCCharts are modeled textually, as seen in Figure 2.1. The textual model corresponds to the ground truth and the diagram continuously applies automatic layout based on changes to the textual model or the diagram configuration via the sidebar (see cog symbol in Figure 2.1). Per default, selecting a diagram element selects the corresponding text and users typically use both representations at the same time. Hence, the editing paradigm of SCCharts qualifies for the use of model order.

Table 2.1 depicts the different semantic elements that will become relevant throughout this thesis, which I exemplarily introduce using the motor SCChart model depicted in Figure 2.2.

The motor SCChart with an abbreviated textual model in Figure 2.2a, which was already depicted in Figure 2.1 inside VS Code. The SCChart shown in Figure 2.2b has six different hierarchy levels: three rectangle packing

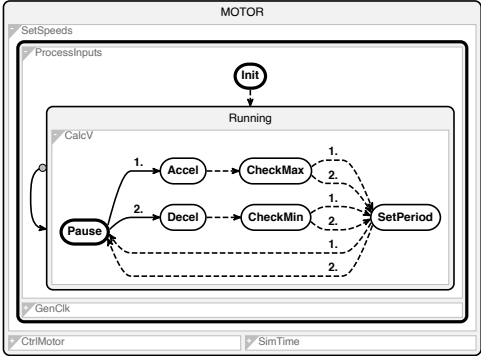


Table 2.1. Different elements of an SCChart

initial state		final state	
state		connector state	
collapsed region		expanded region	
transition with priority, guard, and effect		immediate transition	

**initial state** Pause  
... **go to** Accel  
... **go to** Decel  
**state** Accel  
...  
**state** Decel  
...  
**state** CheckMax  
...  
**state** CheckMin  
...  
**state** SetPeriod  
...

(a)



(b)

Figure 2.2. The motor SCCharts with an abbreviated textual model of the CalcV region in Figure 2.2a is automatically synthesized into a diagram depicted in Figure 2.2b.

## 2. Preliminaries









problems (see Chapter 4), which conform to three hierarchies of white boxes that need to be placed, and three layered layouts (see Chapter 5) with the first two hierarchies being trivial problems consisting of only one or two nodes. Each SCChart or state may become a superstate by specifying concurrent regions, which may either be collapsed or expanded hiding or showing their inner behavior. E. g., the white rectangles called *SetSpeeds*, *CtrlMotor*, *SimTime* are regions inside the motor SCChart. Algorithmically, the placement of regions corresponds to a rectangle packing problem of placing different sized rectangles. Each of these concurrent regions specifies a state machine, which may be any directed graph with one initial state. E. g., in Figure 2.2, *SetSpeeds* is expanded to show its inner state machine consisting of only one initial state, while *CtrlMotor* and *SimTime* are collapsed hiding their inner content, which allows filtering an SCChart depending on the context.

Rounded rectangles visualize states. A thick border highlights initial states (see *Pause* in Figure 2.2b) and two border-lines mark final states, which is shown in Table 2.1. Moreover, states that must be left immediately are called connector states, which are visualized in Table 2.1. Under each state users may specify outgoing transitions, as seen in Figure 2.2a, e. g., in line 2 to 3. The textual order of the specified transitions determines their priority. Figure 2.2b visualizes this using the priority as transition labels. Additionally to normal transitions, transitions might be an immediate transition shown by a dashed line. An immediate transition may fire directly after their corresponding state was reached and may hence “immediately” result in a transition to a new state, which is required for connector states.

In this work, I will only partly introduce the semantics of SCCharts since it is not the main focus here. In particular, I will not introduce abort, suspend, history, or final termination transitions since the nature of a transition is not relevant in the layout. However, anytime the semantics gives clues what SCCharts developers are interested in and how the layout should look like, I will briefly explain. If any open questions regarding the semantics of SCCharts remain, I refer readers to the dissertations of Motika [Mot17] and Schulz-Rosengarten [Sch24].

In terms of desired secondary notation, SCCharts developers show interest in the *flow* of their model. Meaning, SCCharts developers care

**Table 2.2.** Different elements of a Lingua Franca model

reactor		reaction	
startup trigger		shutdown trigger	
logical action		physical action	
labeled reactor ports		timer	

which edges go with or against the main layout direction. Since SCCharts models control-flow, the flow of the diagram should correspond to the actual control-flow of the model. Hence, *backward edges*—edges that go against the main flow of the diagram, e. g., if an edge goes from right-to-left while the main layout direction is left-to-right—should only be introduced if the control-flow somehow goes backward. A backward edge should visualize a repetition or a restart of something. E. g., in Figure 2.2, edges from SetPeriod to Pause correspond to a restart of the state machine and therefore go backward. Additionally, the initial state should be at the start of the model in text and diagram, transition priorities should be visualized by the edge order, and the vertical order of Accel and Decel or CheckMax and CheckMin should be preserved from the textual model if possible.

To fulfil these layout preferences, Section 9.1 presents the final discussion of model order for SCCharts based on the algorithm configurations presented in Chapter 4 and Chapter 5 and their evaluation.

## 2.3 Lingua Franca

Lingua Franca [LMB+21] is a polyglot coordination language for C, C++, Python, Rust, and Typescript that eliminates race-conditions and ensures determinism by construction. Lohstroh et al. conceptualized Lingua Franca as a textual language [LL19] but quickly added a graphical notation [LMS+20]. Moreover, Lingua Franca employs the concept of modeling pragmatics [HLF+22] similar to SCCharts and uses the same editing-paradigm, which

## 2. Preliminaries

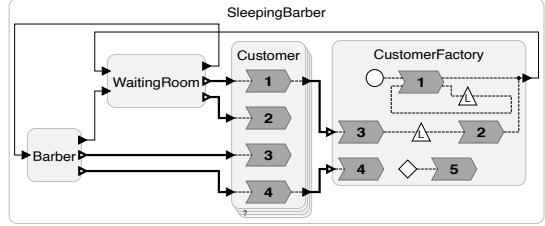
```

factory = new
  CustomerFactory(...)
room = new WaitingRoom(...)
barber = new Barber(...)
customers =
  new[num_customers]
  Customer()

factory.send_customer ->
  room.customer_enters
room.full -> customers.room_full
room.wait -> customers.wait

```

(a)



(b)

**Figure 2.3.** The Sleeping Barber Lingua Franca model [MCL+24]. The abbreviated textual model in Figure 2.3a is automatically synthesized into the diagram in Figure 2.3b.

```

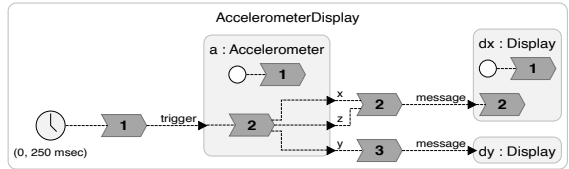
a = new Accelerometer();
dx = new Display(row = 0);
dy = new Display(row = 1);

timer t(0, 250 msec);
state count:int(0);

reaction (t) -> a.trigger
reaction(a.x, a.z) -> dx.message
reaction(a.y) -> dy.message

```

(a)



(b)

**Figure 2.4.** The AccelerometerDisplay Lingua Franca model [Lee24a]. The abbreviated textual model in Figure 2.4a is automatically synthesized into the diagram in Figure 2.4b.

qualifies it for the use of model order.

An overview over the different elements of a Lingua Franca model can be seen in Table 2.2, which I explain based on the two following examples in Figure 2.3 and Figure 2.4.

### 2.3. Lingua Franca

Figure 2.3b depicts the Sleeping Barber example Lingua Franca model with the textual model in Figure 2.3a. The sleeping barber is a classical problem posed by Dijkstra [Dij68] serving as an example for race-conditions between threads, which are partly solved by the implementation in Lingua Franca. Figure 2.4b depicts an accelerometer display with the textual model in Figure 2.4a. Lingua Franca visualizes data-flow between actors, which are here called reactors depicted as rounded rectangles. E. g., Figure 2.3b has four reactors, the two collapsed reactors Barber and WaitingRoom that hide their inner content and the two expanded reactors Customer and CustomerFactory.

A reactor has a private state that is only accessible to its inner content. Hence, a reactor can only communicate with other reactors through events that are part of its interface, which are visualized as ports with optional labels. E. g., Figure 2.3b hides its port labels while Figure 2.4 shows the inputs and outputs of the Accelerometer and Display reactors *a*, *dx*, and *dy*. The reactions (numbered arrow shapes), which reactors mainly consist of, may utilize the reactors state. These reactions hold host code, e. g., C code, and are numbered by their execution order inside their parent reactor. Additionally to reactions, a reactor may hold other reactors, connections between reactors, timers (clock symbol in Figure 2.4), startup (white circle in Figure 2.3b and Figure 2.4) or shutdown (white diamond in Figure 2.3b) triggers, and physical or logical actions (white triangles with letter L for logical actions and P for physical action in Figure 2.3b and Table 2.2). In terms of edges, only the connections between reactors are here modeled explicitly, as seen in Figure 2.3a. Connections between other elements are modeled implicitly. E. g., Figure 2.4a has no explicit connections defined between the reactors and the reactions they connect to. Instead, the definition of reaction 1 specifies that its output is *a.trigger* and the Accelerometer reactor *a* specifies that it has an input named *trigger*.

A Lingua Franca *view model*—the graphical representation of the textual Lingua Franca model—is therefore a compound graph, i. e., a graph consisting of nodes that might again contain graphs, as it is the case for reactors, which can contain reactors. Hierarchical ports connect the different hierarchy levels of a compound graph, as seen in Figure 2.4b. Here, the outputs of reaction 2 inside AccelerometerDisplay traverse the reaction

## 2. Preliminaries

boundary and are available outside the reactor as  $x$ ,  $y$ , and  $z$ . In SCCharts the view model does not contain connected hierarchies. To layout the hierarchical graph efficiently, the hierarchical graph is divided in subgraphs for bottom-up layout based on their hierarchy-level and containment [SSH14]. E. g., the SleepingBarber model in Figure 2.3b consists of three subgraphs: the inner behavior of Costumer, the inner behavior of CustomerFactory, and the top-level graph of the main SleepingBarber reactor, which can only be layouted after the inner graphs. Hence, even though reaction 1 in Customer and reaction 3 in CustomerFactory are connected by an edge in the drawing, they never occur in the same layout graph. Instead, hierarchical dummy ports split the hierarchical edges [SSH14].

In addition to the data-flow models shown above, Lingua Franca supports the use of modal models [SHL+23], which model control-flow. Since the underlying state machines are, however, only a simple version of the SCCharts state machines, I will omit the modal models in my analysis regarding model order.

Since Lingua Franca has a lot of different elements that a model may consist of, the model order of a Lingua Franca model is not uniform but is divided into *model order groups*, as further elaborated in Section 5.7. As a consequence, the intention in the textual model is harder to grasp and differs between distinct developers groups, as further elaborated in Section 9.2.







**Part I**

# **Model Order**



# User Intentions and the Concept of Model Order

*“In many cases, such as the drawing of flowcharts, the input data can be expected to determine the choice of such reversals. In the absence of such input data, we would like to reverse as few edges as possible.”*

*Jünger and Mutzel [JM04], Graph Drawing Software, p. 31,  
Making any Graph Acyclic and Directed*

Typically, there are four *layout goals* to consider for automatic layout: readability, stability, secondary notation, and control [DH24c]. *Readability* refers to the aesthetic criteria that make a layout pleasant to look at and good to work with and therefore helpful to a user that might utilize the layout. *Stability* means that small changes to the model only result in small changes in the layout. *Secondary* notation refers to placement, alignment, ordering, and grouping that highlight how a model works without being explicitly part of the model. *Control* means that the layout can be configured. E. g., users may want to control the layout to visualize secondary notation that may otherwise not be captured by a layout algorithm. *Model order*—the order of elements in the textual model—can be used to control the layout to express secondary notation while at the same time creating stability and balancing layout constraints with readability. Moreover, ordering itself might be an aesthetic criterion since users find unordered numbers irritating, as reported in Section 5.9.6.

### 3. User Intentions and the Concept of Model Order

## 3.1 User Intentions and Model Order

*Model order* is present in all languages that follow the concept of modeling pragmatics and the editing paradigm described in Chapter 2. Since the editing paradigm dictates that the input model is human-made and carries intention if the model itself has a meaning, the model order can harness this user intention for automatic layout making intention measurable by model order.

Utilizing order clashes with the commonly used definition of a graph that most layout algorithms rely on. Graphs are, in the common literature, considered to be *unordered* sets of nodes and edges [STT81; JM04; Kyn09; Tam13]. This is also the case for more recent papers published as part of the GD'22 conference proceedings [AH23]. All 32 papers assume that a general graph  $G = (V, E)$  is not ordered, and only eleven papers add ordering constraints in form of order preserving placement, edge order around a node, st-order, queue order, order inside a path, and similar orderings that do not consider the input order. However, languages that employ modeling pragmatics by design have an ordered input to rely on: the textual model. Hence, I consider *ordered* graphs consisting of *ordered* sets of nodes and edges and consider the model order of nodes, edges, and *ports*—the anchor points of edges—during automatic layout.

Figure 1.1 on page 3 visualizes how automatic layout may struggle to produce the desired results if one only focuses on readability. If we do not consider model order but only readability—here measured by the number of backward edges, edge length, edge bends, and edge crossings—, the drawing in Figure 1.1b and Figure 1.1c on page 3 are equally good. The naming and the model order of the reactors *s* and *t*, however, suggest that Figure 1.1c is preferred and how a human would draw it, which the developer of this model confirmed.

The quote by Jünger and Mutzel [JM04] applies: “In many cases, such as the drawing of flowcharts, the input data can be expected to determine the choice of such reversals. In the absence of such input data, we would like to reverse as few edges as possible.” However, evaluating whether the model order matches the intention and whether the one-dimensional textual order (including line breaks) matches with the two-dimensional layout is still a

non-trivial question. Moreover, to evaluate model order, one needs realistic models and preferably feedback of the original developers to identify the correct layout, i. e., the layout the users actually want to have.

## 3.2 Readability and Model Order

*Readability* focuses on optimizing structural aesthetic criteria such as area, aspect ratio, edge-crossings, edge direction, edge bends, edge length, and symmetry [Pur97] for node-link diagrams, or area and *scale measure* [Rüe18] (see Section 4.1.1) for packing problems. By optimizing these criteria, the drawing becomes less cluttered, more structured, and generally more readable. E. g., it is easier to follow edges if they do not bend, are short, do not overlap with other edges, and go in the primary reading direction. Moreover, reading a model becomes easier if its drawing is compact enough to fit on the screen without scaling it down to an unreadable level.

Existing layout algorithms typically measure and optimize these structural aesthetic criteria [Pur97; Sug02; HHE07; HEH+13]. However, these layout algorithms cannot capture the intention of the model since intention and with it secondary notation is not a measurable criterion. Model order that captures the intention of the developer, as further explored in the following chapters, is, however, measurable and should hence affect the layout. Since readability is nevertheless important and might conflict with model order, I propose two different ways of interaction between model order and readability.

Considering model order as a tie-breaker among drawings of equal quality w.r.t. aesthetic criteria does not compromise readability [DH21]. A tie-breaker solution finds the most ordered solution, i. e., a solution that follows the model order as much as possible, that optimizes the readability criterion that is currently optimized. This is possible by either using model order as a secondary criterion or by using model order as the default order an algorithm operates on. However, using model order as a tie-breaker does only allow partial control over the layout to express secondary notation.

Considering model order as a constraint creates stable layouts and allows to completely control the secondary notation using the textual model.

### 3. User Intentions and the Concept of Model Order

However, model order as a constraint potentially affects the structural aesthetic criteria and may therefore compromise readability, which would also be the case if layout constraints would constrain the drawing, as further discussed in Section 5.2.

In the following chapters, I discuss the concrete effects of model order on readability on the example of SCCharts and Lingua Franca when investigating how model order integrates into automatic layout. In particular, Chapter 4 and Chapter 6 focus on model order and scale measure in packing problems, and Chapter 5 focuses on model order its effect on backward edges, required area, and edge crossings.

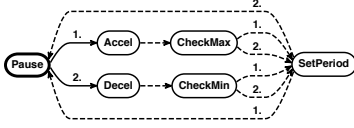
## 3.3 Stability and Model Order

*Stability* means that small changes to the model only result in small changes in the diagram. In contrast to readability, stability is not a structural criterion. Stability cannot be evaluated on static graph layouts [PPP12]. Missing layout stability only becomes apparent if the underlying model changes. Stability helps users to maintain their *mental map*—their inner representation of the model. The mental map helps developers to orient themselves in a view model [PHG06]. Moving many nodes and edges at the same time may compromise the mental map. Constraining the movement of nodes, ports, or edges, and preserving their topology keeps the view model stable and preserves the mental map.

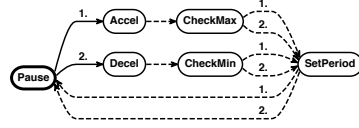
I argue that the mental map does not only exist for the view model but also for the textual model [DH24c]. Developers know where textually they declared elements and would be confused if changing the textual model would reorder unrelated text blocks. Additionally, I argue that the mental map of the textual model and the view model should be consistent with each other to further facilitate the mental map of the developer, as it is the case for Figure 1.1a and Figure 1.1c on page 3.

A more elaborate example for stability can be seen in Figure 3.1. In terms of readability Figure 3.1a and Figure 3.1b are more or less identical not considering the routing of the backward edges from SetPeriod to Pause. A small change to the model—introducing an edge from Accel to Check-

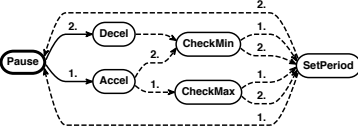
### 3.3. Stability and Model Order



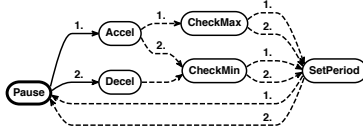
(a) Initial solution without model order.



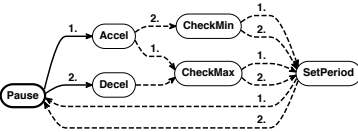
(b) Initial solution with model order.



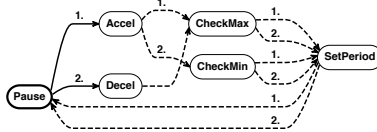
(c) Adding an edge without model order.



(d) Adding an edge with model order.



(e) Moving an edge with model order as a tie-breaker.



(f) Moving an edge with model order as a constraint.

**Figure 3.1.** The motor SCChart visualizes stability. A layout that relies on randomization as a tie-breaker can be seen in Figure 3.1a and a layout that uses the model order as a tie-breaker can be seen in Figure 3.1b. Figure 3.1c and 3.1d show how these layouts change if one adds an edge from Decel to CheckMin. Figure 3.1e and 3.1f show how a potential edge crossing might affect the layout.

Min—changes seemingly unrelated parts of the diagram. Even though one can clearly see that nothing obstructs the path from Accel to CheckMin, an algorithm might change the order of Accel and Decel and create Figure 3.1c. Such an effect may occur if algorithms unnecessarily employ randomization to avoid potential local minima in terms of edge crossings. Especially for larger graphs such a change might easily break the mental map if subgraphs change their node and edge order. If model order is considered, the drawing remains stable, as anticipated by users and depicted in Figure 3.1b and 3.1d compared to Figure 3.1a and 3.1c. Moreover, depending on the *model order configuration*, e. g., whether it is used as a constraint or as a tie-breaker,

### 3. User Intentions and the Concept of Model Order

whether the node, port, or edge model order is used, and the used layout strategy, the drawing can also be configured to be more or less stable and favor readability instead.

Figure 3.1 also shows that increasing the stability may compromise readability since even partially constraining the order of nodes, edges, or ports, may prevent finding the optimal solution regarding edge crossing, backward edges, or scale measure. E. g., moving the edge from Decel to CheckMin in Figure 3.1d to point to CheckMax would create an edge crossing, as seen in Figure 3.1f. This can only be resolved by reordering Accel and Decel or CheckMax and CheckMin, which violates the model order in favor of readability, as seen in Figure 3.1e compared to Figure 3.1f.

## 3.4 Secondary Notation and Model Order

Another layout goal that cannot be measured easily is *secondary notation*. Petre [Pet95] calls the use of alignment, grouping, order, or topology in a graphical representation that indicate how the model works but is not formally part of the model *secondary notation*. Since the secondary notation visualizes intention, it is one of the most important aspects of a diagram layout. While readability measures quantifiable and measurable criteria and aims to find an *optimal* solution by, e. g., minimize the number of backward edges, secondary notation aims to find the *correct* backward edges that highlight how the model works.

Secondary notation can typically not be captured by automatic layout since it is not formally part of the underlying textual model. E. g., repeating the introductory example, both Figure 1.1b and Figure 1.1c on page 3 are optimal w.r.t. their number of backward edges. However, the textual model in Figure 1.1a and the view model in Figure 1.1c employ secondary notation that correctly indicates that Source is the first element that executes. Figure 1.1b employs misleading secondary notation. The ordering of Source and Through instead suggests that Through is handled before Source.

Since secondary notation is not only present in the graphical but also in the textual model similarly to the mental map, model order can bring secondary notation into the layout. E. g., developers of state-machines usually



### 3.5. Control, Model Order, and Automatic Layout

put an initial state at the start of a model file and a final state at the end. Moreover, the order of states tends to respect the control-flow, as detailed in Section 5.9.1.

Again, Jünger and Mutzel [JM04] point out that an underlying ordering is the first choice to determine the *flow*. Hence, that ordering implies the **correct** backward edges of a graph layout, i. e., the layout users want. Purchase [Pur14] additionally suggests that all non-internal graph representations somehow suggest an ordering and people interpret ordering into them when manually laying out node-link diagrams. Considering the model order for layout is therefore a way to enable automatic layout algorithms to capture secondary notation by applying it from the textual model into the diagram.

Additionally, I propose to use model order also as a metric for desired secondary notation by measuring the model order violations in the layout. This model order violation metric could determine the quality of the secondary notation in a model without questioning a domain expert since the model order represents textual secondary notation and controls the layouts.

## 3.5 Control, Model Order, and Automatic Layout

*Control* is again a goal that is hard to quantify and hard to measure but very desirable for automatic layout: “People like having feedback and control” [Tay96]. Consequently, there are many solutions to control layout algorithms via constraints [BP90; Wad01; MSW19] or to control the layout manually in WYSIWYG editors such as the Ptolemy II editor [Pto14], the online editor draw.io<sup>1</sup>, or any GSLP [DLB22] editor<sup>2</sup>. Control solves the issue of mental map preservation by controlling the layout to be stable and the issue of expressing secondary notation by controlling what and how elements are displayed. Moreover, control most notably solves the issue of automatic layout not producing the desired output.

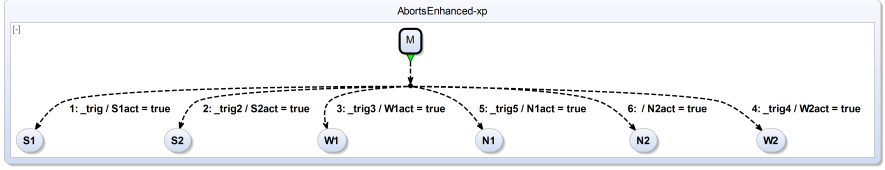
However, exerting control via manual layout or by introducing layout constraint for all elements is tedious [Pet95; PDS+23]. Instead of tedious

---

<sup>1</sup><https://draw.io>

<sup>2</sup><https://eclipse.dev/glspl/>

### 3. User Intentions and the Concept of Model Order



**Figure 3.2.** Figure 5.2.26b in the dissertation of Motika [Mot17] does not respect the model order, as indicated by the non-consecutive transition priorities.

manual editing [Pet95] or introducing layout constraints [PDS+23]<sup>3</sup>, which may be redundant, the intentionally designed textual model holds all the ordering information to exert the desired control. Hence, model order also enables control, and a suitable model order configuration balances between readability, stability, and control over the secondary notation.

Lack of control in automatic layout may produce issues such as the one shown in Figure 3.2. The depicted SCChart by Motika [Mot17] illustrates all available SCCharts transition types and their priority. However, because of the used layout algorithm<sup>4</sup>, the order of transitions in the diagram does not correspond to the priority and the order of states does not correspond to the desired ordering, which can be derived from their naming. This is the case even though ordering states and transitions as desired by Motika would not compromise any layout goal.

While users typically lack control over the layout, it is a highly requested algorithm feature. Many of the models I present in this work are models sent to me by users of Eclipse Layout Kernel (ELK). In the year 2023 alone, 34 of the 85 created tickets for the layout library ELK and its JavaScript cousin elkjs were about questions and issues regarding control over the topology or placement in the layout, which further highlights the importance of control.

Hence, I illustrate in the following how the one-dimensional model order may control the secondary notation of a model, increases stability, influences readability, and brings intention into the two-dimensional diagram while making intention measurable.

<sup>3</sup>See Section 5.2.1 for a discussion of constraints and model order.

<sup>4</sup>Here GraphViz dot [GKN+93] was used since at that time it produced better results than the ELK layered algorithm that is used as an example in this work.

### 3.6 Formalizing Model Order

Even though I will use model order mostly informally throughout this work, I want to formalize the concepts behind model order for interested readers.

One can think of model order as follows. A modeling language may define something I like to call the elements  $A$  of a modeling language. These elements will follow a total order relation  $\prec_A$  in the textual model. E. g., in Figure 3.3a, task1 in line 1 is before task 2 in line 4. Therefore, task 1  $\prec_A$  task 2 holds.

Based on these model elements, a *diagram synthesis*  $s : A \Rightarrow G$  translates the model into something a layout algorithm can work with, e. g., an ordered graph model  $G = \langle V, E \rangle$ , which I will use as a running example here. During this process, a subset of elements  $B \subseteq A$  is translated into a graph representation, e. g., nodes or edges. Moreover,  $s$  might create new graph nodes or edges  $C = \langle V_C, E_C \rangle$  that have no corresponding element in the model. The ordered graph model is therefore given by  $s(B) = \langle V_A \cup_o V_C, E_A \cup_o E_C \rangle$  with  $B \subseteq A$ . Here,  $V_A$  and  $E_A$  are the nodes and edges that have a corresponding element in the model, and  $\cup_o$  is the ordered unison of two sets meaning that  $s$  may insert nodes in  $V_C$  and edges in  $E_C$  not necessarily sequentially after the existing nodes and edges but anywhere in  $V_A$  and  $E_A$ .

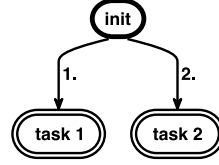
An example using a fictional modeling language can be seen in Figure 3.3. Here, only task 1 and task 2 are translated into the graph model while end and the two go to end transitions are only indirectly used to make task 1 and task 2 final states. Ignoring model elements or only using them indirectly might happen if the element is essential for the grammar of the given modeling language but not to its user. Here, I assume that a user might be interested in all sources of the underlying graph such that the diagram synthesis  $s$  adds a root node that connects to all sources in the model, i. e., task 1 and task 2. Here, the graph model is an abstraction of the textual model, as suggested by modeling pragmatics [HLF+22] to visualize user-relevant information.

Based on this, I define the model order of nodes as the partial order  $P_{o_v} = (V, \prec_{o_v})$  and the model order of edges as the partial order  $P_{o_e} =$

### 3. User Intentions and the Concept of Model Order

task 1  
go to end

task 2  
go to end  
end



(a) Textual model  $A_1$  with a total order of elements.  
(b) Graphical model of graph model  $G_1$  that follows the model order.

**Figure 3.3.** An example for a modeling language that showcases how model order relates to the order of model elements.

$(E, \prec_{o_e})$  with  $V = V_A \cup_o V_C$  and  $E = E_A \cup_o E_C$ . This also implies that the diagram synthesis  $s$  may also influence the model order, e. g., by putting the node representation of `init` in Figure 3.3b before all other nodes, such that  $G_1 = \langle \langle \text{init}, \text{task 1}, \text{task 2} \rangle, \langle e1, e2 \rangle \rangle$  instead of  $\langle \langle \text{task 1}, \text{task 2}, \text{end} \rangle, \langle e1', e2' \rangle \rangle$ , as suggested by the textual model Figure 3.3a.

Also note, when extending the model order by group model order in Section 5.7, this model order definition changes such that the might be multiple partial model orders for nodes. E. g., instead of  $P_{o_v} = (V, \prec_{o_v})$ , a graph model may use  $P_{o_{v_1}} = (V_1, \prec_{o_{v_1}})$  to  $P_{o_{v_p}} = (V_p, \prec_{o_{v_p}})$  for the different model order groups  $V_1$  to  $V_p$  with  $V = V_1 \cup \dots \cup V_p$  while assuming that  $\bigcap_{i \neq j} V_i \cap V_j = \emptyset$ .

**Formalizing Readability** Readability is typically already formalized as part of a layout algorithm. Typically, a layout algorithm will aim to optimize some readability metric or aesthetic criterion that is statically measurable by some metric  $m : D \Rightarrow \mathbb{R}$  which takes a drawing  $d \in D$  and outputs some kind of value associated with a drawing. This value does not only express how good a drawing looks, but also how good it will be to work with, which is typically evaluated for general aesthetic criteria such as edge crossings. Comparing the value of that metrics may allow algorithms to optimize regarding some metric that might be most important or even all at the same time [Pur02; ADD+20].

### 3.6. Formalizing Model Order

**Formalizing Stability** Stability on the other hand is not that easy to measure since it is not only about geometrical distance between elements in the drawing  $d$  over time, but also considers mental map preservation. Therefore, stability rather means that all “important”  $I = (V_I, E_I)$  with  $V_I \subseteq V$  and  $E_I \subseteq E$  preserve their ordering in drawing  $d'$  compared to  $d$ .

Meidiana et al. suggest solving this issue by considering *change faithfulness* [MHE20]. The key difference to my work, however, is that Meidiana know how their graph changed over time. They are using layout adjustment [MEL+95] techniques, while I focus on layout creation. I.e., I always create the layout from scratch. Moreover, they are considering force-directed algorithms for dynamic graph drawing that compare the change over time in their graph with the geometrical change in the drawing while also considering the *cluster change faithfulness*.

I instead assume that I know nothing of the graph except its corresponding model and use this to infer how the graph should evolve and keep it stable. The stability can here be measured by counting the model order violations in the drawing, as suggested in Section 5.5.4 for layered layout, by assuming at all important nodes and edges are marked by model order. Therefore, stability and mental map preservation can be seen as just another metric. However, I want to add here that especially the mental map is not that easy to capture formally since it is formed by thinking about the model, modeling the model textually, and looking at the textual model and the drawing, the graphical model. Therefore, the mental map itself is in constant change and adapts to the current circumstances.

**Formalizing Secondary Notation** I interpret secondary notation as user intention that is an unknown, partial relation between nodes and edges depending on the direction to capture the ordering aspect of secondary notation.

Secondary notation can therefore be seen as a partial order that corresponds to the desired ordering envisioned by a user. However, this ordering cannot be one-dimensional since the resulting drawings are two-dimension. Therefore, I think of secondary notation for graphs as partial ordering relations for  $x$  and  $y$  direction for nodes  $sc_{n_x} \subseteq V_{o_1} \times V_{o_1}$ ,  $sc_{n_y} \subseteq V_{o_2} \times V_{o_2}$  and edges  $sc_e \subseteq E_o \times E_o$  with  $V_{o_1}, V_{o_2} \subseteq V$  and  $E_o \subseteq E$  being sets of nodes

### 3. User Intentions and the Concept of Model Order

and edges for which the ordering matters. Since the secondary notation depends on what users of a modeling language actually want [Pet95], I must treat this as an unknown relation. However, I assume that this secondary notation is partially also poured into the textual model while writing it, which allows me to leverage the model order to capture intention.

Note, that this formalization does not capture the positioning, alignment, and grouping aspects of secondary notation. I. e., while concrete positions cannot be captured using model order, grouping and alignment can only be partially captured since the ordering relations  $P_{ov}$  and  $P_{oe}$  only allow to constrain which nodes and edges could be grouped together transitively. Real grouping and alignment requires a second dimension in model order, as detailed in Section 10.4.9.

**Formalizing Control** Control always has to be considering together with secondary notation or intention. Therefore, I see control as the ability to replicate the secondary notation relations using model order.

Given a drawing  $d$  based on a graph model  $G$  derived from an ordering of textual elements  $(A, \prec_A)$ , control means that a user may create any *sensible* ordering that fulfills the partial ordering relations  $sc_{nx}$ ,  $sc_{ny}$ , and  $sc_e$  given by the secondary notation by changing the textual model and the ordering of its elements from  $(A, \prec_A)$  to  $(A', \prec_{A'})$ . I. e., we have full control over the drawing if reordering the textual model allows us to create all desired secondary notations.

Henceforth, I will only briefly mention this formalization for algorithms and concepts in Part I since I think that it overcomplicated the basic concepts behind model order. E. g., in SCCharts the ordering  $(A, \prec_A)$  corresponds directly to the ordering of nodes  $P_{ov}$  and edges  $P_{oe}$  in the underlying graph model with both being total orders. At the same time the diagram synthesis  $s$  does not add new nodes or edges for SCCharts such that  $V_C$  and  $E_C$  are empty while the model order can be considered for all elements such that  $B = A$ . Finally, no model order group exist for SCCharts making it the simple example language compared to Lingua Franca.

# Model Order in Rectangle Packing

Automatic layout for modeling languages requires to consider the nature and intent of the language. The intent of a language defines how to improve readability, preserve the mental map, express secondary notation, and how users may control the layout. This is also the case for rectangle packing problems, e. g., the one seen in Figure 4.1, posed by SCCharts that requires packing rectangles in a given aspect ratio.

Typically, heuristics that pack rectangles into a given width or aspect ratio require sorting rectangles by size for area efficiency [DD92]. However, since the rectangle packing problem imposed by SCCharts regions involves regions changing their size when collapsed or expanded, sorting regions by size greatly compromises stability and disturbs the mental map. I. e., users loose track of the rectangles and therefore loose orientation since their mental image does no longer correspond to the depicted packing. Moreover, SCCharts developers express secondary notation and with it intention with their region ordering. A stable ordering is therefore important for the layout of SCCharts regions. Hence, the region placement should always represent the underlying region order. At the same time, a region placement needs to optimize the scale measure (see Section 4.1.1) to create drawings that fit into a given aspect ratio, as depicted in Figure 4.1b.

Since SCCharts regions have readability and stability constraints as well as a desired secondary notation, I will use the rectangle packing problem posed by SCCharts regions as an example for model order in rectangle packing. Moreover, SCCharts regions are a suitable candidate since I myself have experience in SCCharts development, I have excess to a variety of SCCharts models, and expert SCCharts developers are willing to be interviewed, which covers the criteria to investigate a given modeling

## 4. Model Order in Rectangle Packing

language for the use of model order, as detailed in Section 9.4.

The previously used layout algorithm for SCCharts regions—here presented as the *simple heuristic* in Section 4.3—constrained the rectangle order by model order by introducing ordering constraints as part of the diagram synthesis. This shows that stability is desired by SCCharts developers. However, this simple heuristic failed to create a space efficient layout, which sometimes severely decreased readability, as seen in Figure 4.1a. Here, more than half of the area given by the desired aspect ratio (dashed-black) is empty. The proposed *rectpacking algorithm* that solves the *region packing problem* (see Section 4.1)—motivated by SCCharts regions—instead produces a much more compact drawing, as seen in Figure 4.1b, while utilizing the model order as a constraint.

This chapter explains how model order can be used in rectangle packing (IMPL) in Section 4.4 and evaluates the proposed rectpacking algorithm in Section 4.6 comparing to the optimal solution while discussing how model order relates to readability, stability, secondary notation, and control (EVAL). To do this, I first introduce the region packing problem posed by SCCharts regions.

### 4.1 The Region Packing Problem

Given an ordered set of  $n$  rectangles  $r_i$ —here called regions—with a minimum width  $w_i$  and height  $h_i$ , one tries to find an overlap free placement by finding appropriate values for the region coordinates  $x_i$  and  $y_i$  while maximizing the scale measure (see Section 4.1.1). I.e., one aims to fit in a given aspect ratio without wasting space. This scale measure is the main readability criterion for the region packing problem.

Additionally, the placement should maintain the ordering of rectangles by considering the *reading direction*<sup>1</sup>(see Section 4.1.3) and the placement

---

<sup>1</sup>The dominant reading direction may differ from country to country and culture to culture. It is mainly influenced by the writing-direction in a country's main language. Since this work is developed in Germany, I assume that the dominant reading and writing direction is left-to-right. By transformation of the input problem this can be transformed into any other consistent reading direction.



## 4.1. The Region Packing Problem

should allow *whitespace elimination* (see Section 4.1.2), such that the width and height of regions can be increased to fill gaps between them<sup>2</sup>.

### 4.1.1 Scale Measure

The scale measure [Rüe18] expresses how big diagram elements can be drawn on a screen. If the scale measure is higher, as seen in Figure 4.1b compared to Figure 4.1a, each region and their inner content can be scaled such that the inner content becomes more readable. Hence, a large scale measure (SM) implies a better readability. We define the SM based only on the desired aspect ratio (DAR) as  $\min(\frac{DAR}{w_a}, \frac{1}{h_a})$  where  $w_a$  and  $h_a$  are the *actual width* and *actual height* of the drawing.

E.g., in Figure 4.1a the actual width  $w_a$  is the aggregated width of regions A to H and the actual height  $h_a$  is the aggregated height of region H and K<sup>3</sup>. The minimum of  $\frac{DAR}{w_a}$  and  $\frac{1}{h_a}$  describes how much the drawing must be scaled down to fit into a “1 : DAR” frame visualized by the black-dashed box in Figure 4.1a. In Figure 4.1a the actual width is equal to the width of the visualized DAR. Hence, the width is the limiting factor for the region scaling in Figure 4.1a. In Figure 4.1b the actual height is the limiting factor for the region scaling. Compacting the regions, as seen in Figure 4.1b, allows scaling the regions to be more readable while using the same DAR. Hence, Figure 4.1a has a lower scale measure since placing regions next to each other instead of stacking them is space inefficient.

To summarize, readability is increased by drawing elements bigger in a desired aspect ratio, allowing for whitespace elimination to remove gaps, and by forming a row structure creating a consistent reading direction, which I further explore in Section 4.1.2 and Section 4.1.3.

### 4.1.2 Whitespace Elimination

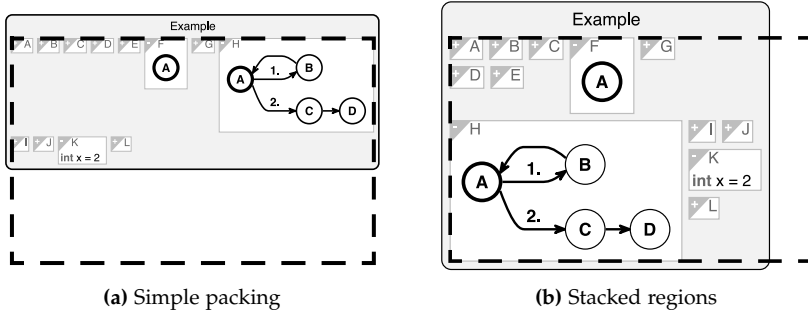
SCCharts users do not like if the region packing has gaps between them, as detailed by SCCharts experts in informal interviews. This has three reasons.

---

<sup>2</sup>This is why I defined rectangles by their minimum width and minimum height.

<sup>3</sup>Note that I omitted the region spacing here for brevity and simplicity.

#### 4. Model Order in Rectangle Packing

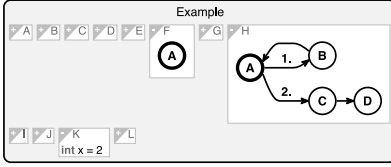


**Figure 4.1.** Stacking regions increases the scale measure. The scale measure of Figure 4.1a is significantly lower (i. e., worse) than the scale measure of Figure 4.1b since regions are not stacked. The desired aspect ratio is marked in dashed-black.

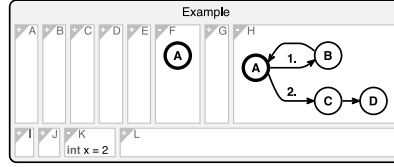
First, the gaps cannot be interacted with to expand or collapse the drawing. Therefore, the whitespace gaps reduce the interactivity of the model. Second, the whitespace gaps make it harder to visually navigate from region to region since there may be a lot of empty space between them, as seen in Figure 4.6b and Figure 4.6c on page 40. E. g., in Figure 4.6c, the whitespace elimination emphasizes the row structure of the packing. Third, whitespace elimination masks ONO region packings. A region packing is ONO if a user, e. g., an SCCharts developer, thinks that a trivial or obvious change to the packing could improve its readability or secondary notation. E. g., Figure 4.2a makes it fairly obvious that the drawing could be compacted by stacking regions below each other. If the whitespace is, however, eliminated, this problem becomes less obvious, as seen in Figure 4.2b. Hence, the packings structure should allow increasing region width and height to completely fill out the existing gaps without increasing the scale measure.

Figure 4.3 illustrates why arbitrary packings might not allow to completely eliminate whitespace. Here, the regions F, H, J, and K all border the same whitespace rectangle such that none of them can fill the gap while maintaining their rectangular form and maintaining the size of the drawing. If we, however, only place regions in rows consisting of stacks of subrows, we can assure that whitespace elimination is always possible.

## 4.1. The Region Packing Problem

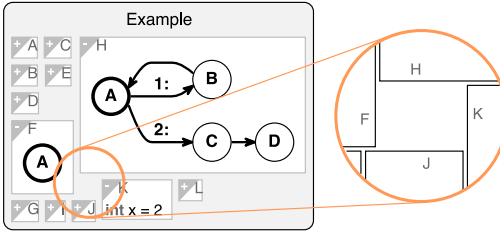


(a) Without whitespace elimination



(b) With whitespace elimination

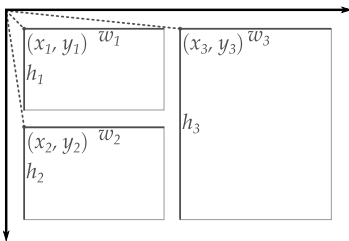
**Figure 4.2.** Whitespace elimination for a simple model. The whitespace gaps in Figure 4.2a are eliminated by increasing the width or height of the placed regions, as seen in Figure 4.2b.



**Figure 4.3.** An unconstrained region packing might hinder whitespace elimination. The regions F, H, J, and K all border the same whitespace such that none of them can fill the gap while maintaining their rectangular form.

### 4.1.3 Row Structure and Reading Direction

To describe the row structure a region packing has, I first introduce the coordinate system employed in this thesis. The coordinate system has its origins in the top-left corner and the origin of each element is consequently its top-left corner as well, as seen in Figure 4.4. Hence, a region with a high y-coordinate is placed at the bottom of a drawing. Moreover, users



**Figure 4.4.** The coordinate system has its origins in the top-left corner. Each region is therefore placed based on their top-left coordinate  $(x_i, y_i)$  with their width  $w_i$  and height  $h_i$  determining their size.

#### 4. Model Order in Rectangle Packing

read the packing from left-to-right starting at its top-left corner and use top-to-bottom as their secondary reading direction, which is called the Z-path of reading [Coh13] named after the shape of the letter Z.

Maintaining a consistent reading direction increases stability and makes regions discoverable by finding their neighbors. Moreover, the reading direction is one way to visualize the intended ordering of regions and with it the secondary notation of an SCChart. Hence, model order should control the packing to adhere to the reading direction creating a stable packing stable with improved secondary notation (*EVAL*).

A row structure is required to make the Z-path reading direction consistent. E. g., the row structure shown in Figure 4.5 preserves the reading direction and allows whitespace elimination. Here, the row structure divides rows (dotted-black) into stacks (red) of subrows (orange)<sup>4</sup>. In this row structure, the *dominant element*—the tallest region in the row—determines the row height and visually emphasizes the bounds of a row.

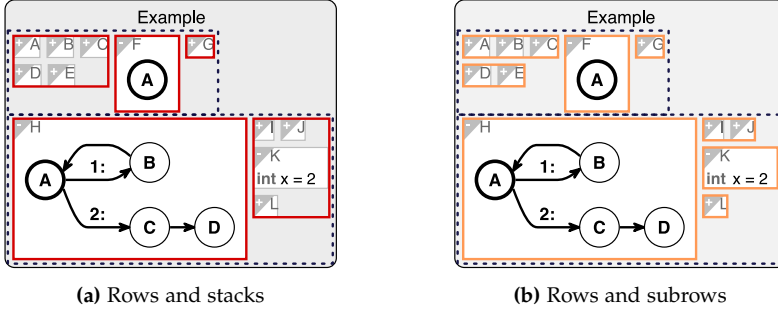
Comic panel packing uses a similar effect to intuitively let users read the next panel. When reading comic panels, such a dominant element that other panels may form subrows against is called a blockage [Coh13]. Cohn showed that experienced comic readers do diverge from the Z-path when facing a blockage and instead of continuing to read left-to-right they try to find the next panel below. Using the visual blockage, users assemble the subrows before a dominant element or the end of the row into one visual unit [Coh13], which I call a stack. The remaining alignment problem of subrows in neighboring stacks, as depicted in Figure 4.37b and Figure 4.37c on page 89, is solved for the region packing problem by grouping elements of similar size, as presented in Section 4.4.3, by primarily focusing on the left-to-right reading direction.

For region packing, Figure 4.6 illustrates how dominant elements visually create rows and subrows. The packing in Figure 4.6a consists of one row and does not use dominant elements. Hence, it is harder to grasp whether region H or G should be after region F without considering their lexicographic order given by their model order. This is solved by row-dominant elements

---

<sup>4</sup>Although we could further divide subrows into substacks and subsubrows, we limit ourselves for the sake of this work since SCCharts regions—the running example for the proposed rectangle packing algorithm—do not benefit from this.

## 4.1. The Region Packing Problem



**Figure 4.5.** The row structure of a packing. Figure 4.5a displays how rows (dotted-black) are divided into vertical stacks (red). Each stack may consist of several subrows (orange), as depicted in Figure 4.5b.

at the cost of scale measure, as seen in Figure 4.6b. Hence, Figure 4.6b is less space efficient than Figure 4.6a because of the dominant element constraint but visualizes the row structure, which is required for a consistent reading direction. Eliminating whitespace additionally emphasizes the rows and subrows, as seen in Figure 4.6c, making rows and subrows even easier to recognize. Hence, the region packing problem requires the resulting packing to use a row structure with dominant elements.

Formally, the row structure constrains the region packing as follows. All regions should adhere to the *basic ordering requirement*. For all regions  $r_i$  and  $r_j$  with  $i < j$ ,  $r_i$  should be horizontally or vertically before region  $r_j$ , which requires that Equation 4.1.1 holds.

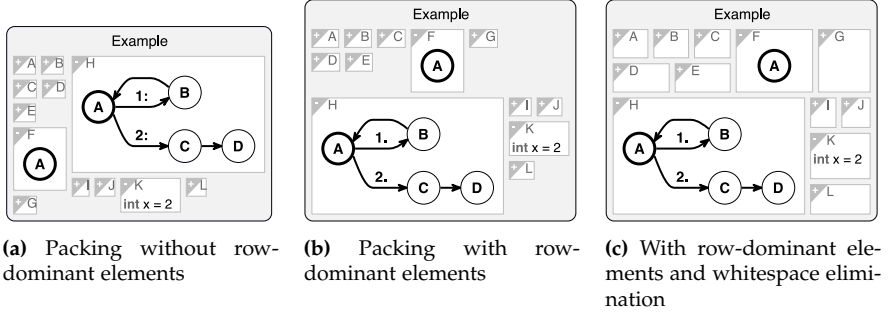
$$x_i + w_i \leq x_j \vee y_i + h_i \leq y_j \quad (4.1.1)$$

Furthermore, the row structure restricts the possible positions of region  $r_{i+1}$  based on the preceding region  $r_i$  by the following rules:

Rule 1:  $r_{i+1}$  is directly right of its preceding neighbor  $r_i$  and top-aligned (4.1.2)

Rule 2:  $r_{i+1}$  is right of its preceding neighbor and top-aligned at the

#### 4. Model Order in Rectangle Packing



**Figure 4.6.** While reading through the regions, Figure 4.6a makes it harder to discern whether H or G is the region after F since the packing creates a column structure that works against the reading direction instead of using a row structure. In Figure 4.6b, this is discernible since each row has a dominant element. The visual recognition of rows can be further improved by whitespace elimination, as seen in Figure 4.6c.

current row level, which corresponds to the placement

in a new stack (4.1.3)

Rule 3:  $r_{i+1}$  is in the next subrow in the same stack of  $r_i$  (4.1.4)

Rule 4:  $r_{i+1}$  is in the next row. (4.1.5)

Rule 1 (Equation 4.1.2) applies to region D and E or F and G in Figure 4.6b. Regions E and G are placed directly right of D and F in the same row or subrow. Rule 2 (Equation 4.1.3) applies to E and F and Rule 3 (Equation 4.1.4) applies to C and D or K and L. Region F is placed in a new stack, right of the last stack consisting of regions A to E. Regions D and L are placed in a new subrow directly below the subrows consisting of regions A to C and region K. Here, Rule 4 (Equation 4.1.5) only applies to regions G and H since the drawing has only two rows. Region H is placed in a new row beginning directly below all already placed regions.

Note that placement constraints do neither suffice to get good solutions nor do they capture the dominant element constraint. The region packing in Figure 4.1a only follows Rule 1 (Equation 4.1.2) and Rule 4 (Equation 4.1.5) and has a clear reading direction but a bad scale measure. Figure 4.11 on

page 47 does potentially follow all rules but does not use Rule 1 often enough.

The region packing problem does therefore remain inherently complex. Hence, one has to employ a heuristic to solve the region packing problem such that a resulting algorithm is usable in an interactive tool, as required in Chapter 2.

## 4.2 Related Work

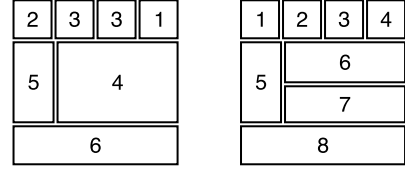
The comic panel placement [Coh13], as well as written text such as the text this work consists of, resembles the region packing problem. Written text, however, only consists of rows without subrows between them, making the placement of words based on their order trivial if the target width is known (see Section 4.3). Comic panel placement instead forms rows and subrows as required for the region packing problem. Moreover, comic panels typically require a gap-free placement. However, comic panel placement does not have panels with variable sizes, panels with tremendous size differences as it may be the case for SCCharts regions (e. g., region H had nearly five times the width and height of region A), and artists may take arbitrary long. Artists additionally have the freedom to change the size and content of panels to fit them on a page in a desired way as well as create artistic overlaps of panels [Coh13] which would be undesired for SCCharts regions. Finally, both have different goals. Comic panels serve a narrative, while a region packing “only” serves the layout goals and the intention of the developer.

In addition to the placement of comic panels, the region packing problem relates to several rectangle packing problems such as the ones listed by Dowsland and Dowsland [DD92]. However, these rectangle or *strip packing problems*—rectangle packing with a given target width—typically do not have an ordering, scale measure, reading direction, or whitespace elimination requirement. However, most are inherently complex. Hence, I present a selection of algorithms with similar goals in the following to illustrate their differences.

Rectangle packing problems with ordering constraints usually come

#### 4. Model Order in Rectangle Packing

**Figure 4.7.** Figure 4.7a illustrates a packing with unloading constraints by da Silva et al. [DMX13]. Figure 4.7b illustrates how region packing would create a valid packing with unloading constraints.



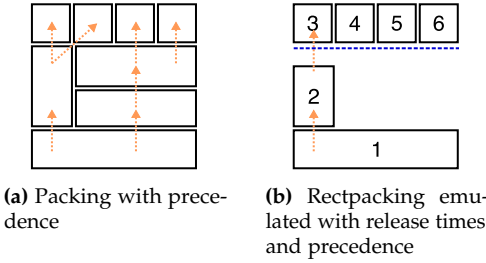
(a) Valid packing with unloading constraints (b) Valid region packing emulating unloading constraints

from the transportation industry. Da Silva et al. [DMX13; DXM14] packs packages such that they are stable and can be removed on the corresponding stop, as seen in Figure 4.7. Here, Figure 4.7a shows how such a packing may look like. Compared to the proposed region packing problem, they consider groups of packages instead of the order of each individual package and do not require that whitespace must be eliminated. Instead of having a consistent reading direction, they require that a package should not have a package with a higher number blocking the vertical space above it. Hence, one should be able to take out all packages from the top beginning with the lowest number. The region packing problem, as seen in Figure 4.7b, illustrates how a valid region packing is always a valid packing with unloading constraints. The region packing in Figure 4.7b allows taking out the regions with the highest number one after another. Finally, the layout goals stability and secondary notation required for diagrams used with the proposed editing paradigm are negligible for packing with unloading constraints. The packing itself has no secondary notation and therefore no intention and stability requirements.

Augustine et al. [ABI06] describe another interesting packing problem that uses order. Augustine et al. restrict the vertical placement of regions during strip packing with precedence constraints, as seen in Figure 4.8. Here, precedence constraints are marked with dotted-orange edges. Additionally, release times (dotted-blue) for rectangles constrain the strip packing problem. Augustine et al. use this packing to create schedules for dynamically reconfigurable Field Programmable Gate Arrays (FPGAs). Hence, the resulting packing does neither follow a clear reading direction nor does it always create a row structure with row-dominant elements that allows



## 4.2. Related Work



**Figure 4.8.** Strip packing with precedence constraints introduces vertical placement constraints for all rectangles, as seen in Figure 4.8a. Figure 4.8b visualizes how the region packing problem would work using precedence constraints. The release time of region 3 is marked in dotted blue and the precedence constraints are marked in orange.

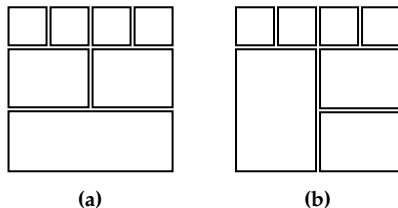
whitespace elimination. The reading direction required by the region packing problem instead requires vertical or horizontal placement constraints to form a clear reading direction, as detailed in Figure 4.8a. Similarly, release times cannot be trivially integrated into the region packing problem since it cannot directly constrain the vertical position of a region to be above a threshold. Doing so would be possible by placing a region that is not yet released in a new row starting at the release time, as visualized in the region packing in Figure 4.8b.

Another rectangle packing algorithm is Treemaps by Bruls et al. [BHV00]. Bruls et al. introduce a treemap visualization algorithm for file systems or other hierarchical structures. Treemaps also form rows and subrows to place the rectangles in. However, they only consider order for packing effectiveness and do not consider maintaining the reading direction and a row structure, as seen in Figure 4.9. Rectangles should be ordered by size such that placing the biggest one first increases the effectiveness of the packing algorithm. Moreover, their rectangles do not have a minimum width and height but rather a size goal that may be adjusted to fill all gaps of the packing, as it is also required to do whitespace elimination. Additionally, treemap visualization aims to avoid abnormally wide or high rectangles. Hence, a treemap visualization algorithm would aim for Figure 4.9b rather than Figure 4.9a.

Somehow related to rectangle packing are wordclouds. Wang et al. [WCB+17] present an algorithm to layout wordclouds that preserves the neighborhood of words. The region packing problem, however, may sepa-

## 4. Model Order in Rectangle Packing

**Figure 4.9.** Treemap visualizations allow rectangles with a more fluid shape. Instead of requiring a minimum width or height, the area should be preserved.



rate neighboring regions if they are in neighboring rows, stacks or subrows. Similarly, a neighborhood preserving wordcloud algorithm does not preserve a reading direction, does not create rows and subrows of regions to read them easily based on their order, does not have a clear order of regions, and typically does not allow eliminating whitespace between them.

Therefore, the region packing problem has different requirements than other packing problems, and I propose a first solution in the following.

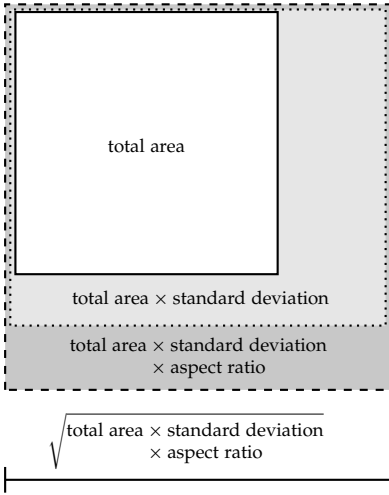
### 4.3 A Simple Heuristic

The easiest way to create a packing that adheres to a left-to-right reading direction, allows whitespace elimination, and is quick enough to be used in an interactive context is to pack all regions next to each other forming one long row of regions. The second-easiest way is the *simple heuristic* that forms multiple rows of regions, as seen in Figure 4.1a on page 36. The simple heuristic achieves this by dividing the region packing problem into a width approximation step, a trivial placement step, and an optional whitespace elimination step.

During width approximation, the target width of the packing is statically approximated. Specifically, if the regions have very similar sizes, the width can be approximated by the maximum of the widest region and the square root of the total rectangle area times the aspect ratio, as seen in Figure 4.10.

Here, the total area multiplied by the standard deviation results in the area that is most likely used by the packing as a result of the different region sizes. This area has no specific shape. Therefore, we cannot derive a target width from it. Therefore, we make sure that we can consider the area

### 4.3. A Simple Heuristic



**Figure 4.10.** The static width approximation of the box algorithm visualized. The total area is shown in white, the standard deviation between the different elements makes up for potential gaps if the size of the elements differs (light-gray with dotted border). The aspect ratio scales the size to account for the part the width has in the total area (gray with dashed border). A square root of the size of size in form of a square returns the approximated target width.

to be a square by multiplying it with the desired aspect ratio. Hence, taking the root of the square results in the desired target width. E. g., consider a box with area  $120 \times 100$  that visualizes the total area of all rectangles (see white total area box). With a standard deviation of 1.25, the light-gray area with a dashed border that visualizes the total area times the standard deviation is  $150 \times 125$ . This adds the area that would most likely be empty to the calculation. Since the desired aspect ratio is 1.2, the calculation  $150 \times 125 \times 1.2 = 22500$  yields the gray area with the dotted border. This calculates the percentage of the width on the total area. The square root of the gray area, which is  $22500 = 150 \times 150$ , yields the approximated target width of 150. Hence, the maximum of the approximated target width and the widest region determines the target width for the following strip packing problem.

The strip packing problem is solved by trivially placing regions in rows bounded by the target width, as seen in Figure 4.2a on page 37. A region is directly right of its preceding region if it fits in the target width. If the region does not fit, it is placed below all already placed regions in a new row. The optional whitespace elimination step may then eliminate gaps, as seen in Figure 4.2b on page 37, by increasing the region height to the row height

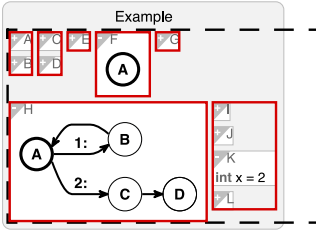
#### 4. Model Order in Rectangle Packing

and increasing the width of the last region in each row to fill the row width. Since the width approximation, placement, and whitespace elimination can be solved in  $\mathcal{O}(n)$ , the simple heuristic can be solved in  $\mathcal{O}(n)$ , which makes it suitable to be used in an interactive context, as detailed in Chapter 2, and for large packing problems.

The simple heuristic produces very good results if none of the regions are stackable inside a row [DLH+23]. I.e., if the region heights are very similar and no region has two neighbors in one direction that have an aggregated height smaller than the region itself. Hence, in case of non-stackable regions, the simple heuristic produces drawings with high scale measure, as detailed in Section 4.6, while keeping the ordering of regions in a row stable and controllable. Moreover, the simple heuristic always maintains the reading direction.

The simple heuristic, however, has the following problems as reported by SCCharts experts and experienced by myself: First, if the rectangle sizes vary such that we have few big regions and many small ones, as it is the case for the SCCharts regions if one or a few concurrent regions are expanded, the static width approximation is quite bad as seen in Figure 4.1a compared to Figure 4.1b on page 36. Since the required width is in this case always over-approximated using the static formula, the drawing is too wide resulting in a bad scale measure and therefore low readability when enforcing model order. Second, when eliminating the whitespace by increasing the region width and height, the drawing looks even worse since regions become abnormally formed, i.e., very slim or very high, as seen in Figure 4.2b on page 37. Third, if one showed Figure 4.1a on page 36 to a human, she could easily come up with a better compacted drawing by stacking the regions, as seen in Figure 4.11, making the simple heuristic ONO. At the same time, the only way to control the packing is to change the desired aspect ratio. Such ONO packings with limited control undesired and may give automatic layout a bad image. If a user has the impression that she could easily come up with a better solution by manual layout, tedious manual layout may be used instead of automatic layout. Hence, I propose a more advanced heuristic to create stable automatic layouts that show intended secondary notation without threatening readability in the following.

## 4.4. Rectpacking Heuristic



**Figure 4.11.** The simple compaction algorithm only creates stacks (red) of regions to fit them into a given aspect ratio (dashed-black) [Luc18]. Therefore, the reading direction is compromised. The right and down reading direction alternate, making it harder to read through the regions and to recognize their ordering. Without the marked stacks one would most likely read the regions left-to-right with top-to-bottom as secondary reading direction.

## 4.4 Rectpacking Heuristic

Although simply stacking regions, as seen in Figure 4.11, compacts the solution of the simple heuristic, the resulting packing violates the reading direction and shows bad secondary notation, as explained in Section 4.1.3. Placing a region to the right of the previous one should have priority over placing it below to make the region order recognizable and consistent as part of desired secondary notation.

To solve the reading direction issue and the readability issues in the simple heuristic, one needs an algorithm that approximates the target width of the packing using the actual size and the order in which elements might occur. Such an algorithm should preserve the reading direction and therefore the model order and secondary notation of the textual model. Moreover, this algorithm should avoid ONO packings shown in Figure 4.1a by producing a compact drawing that fits the DAR, and should prevent abnormally formed regions during whitespace elimination.

Hence, I present a heuristic that produces a compact region packing without compromising the reading-direction based on “Revisiting Order Preserving, Gap-Avoiding Rectangle Packing” [DLH+23]. The rectpacking algorithm enforces model order as the reading direction to increase stability and secondary notation in the resulting packing without threatening readability. The resulting packing has a high scale measure such that SCCharts regions and their inner content are still readable when packed into a desired aspect ratio given by the diagram view the SCCharts will be displayed in.

The algorithm is divided into four subproblems:

## 4. Model Order in Rectangle Packing

*width approximation* to get a desired target width and transform the problem into a strip packing problem in Section 4.4.1,

*placement* to approximate the height for each row the regions will be placed in Section 4.4.2,

*compaction* to stack regions inside a row in Section 4.4.3, and

*whitespace elimination* to fill the gaps that might be created in Section 4.4.6.

### 4.4.1 Rectpacking Width Approximation Step

To handle the inherent complexity of the region packing problem, the first step reduces the problem to a strip packing problem by approximating the desired target width of the packing.

If one assumes regions with varying width and height and a fixed ordering, a static width approximation only based on area, aspect ratio, and available regions is not sufficient to create a good scale measure, as seen in Figure 4.1 on page 36. Here, Figure 4.1a on page 36 overestimates the target width since the approach vastly overestimates the whitespace created by the placement resulting in a low scale measure. Hence, the regions in Figure 4.1a are much smaller compared to Figure 4.1b on page 36. Moreover, the actual aspect ratio does not match the given desired aspect ratio. Hence, if this SCChart would be drawn in a diagram view, such as the one displayed in Figure 2.1 on page 8, the regions may be very small since the resulting SCChart does not fit the aspect ratio of the diagram view.

Sorting elements by size<sup>5</sup> would result in a much lower target width and potentially in a better scale measure, as seen in Figure 4.12a compared to Figure 4.12b. As a result, the drawing becomes much more compact compared to the order preserving packing in Figure 4.12b. Since SCCharts developers, however, want to preserve the ordering, this effect has to be taken into account when approximating the target width to prevent drawings with low scale measures and bad readability.

---

<sup>5</sup>Dowsland and Dowsland recommend sorting elements by size for space efficient drawings [DD92].

---

**Algorithm 1:** Greedy Width Approximation Heuristic

---

**Input:** Regions  $rs$ , spacing  $s$ **Output:** Placed regions  $ps$ , width  $w$ 

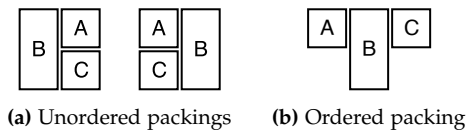
```

1   $ps := \{r_0\}$  // The already placed regions
2   $drawingWidth := w_0$ 
3   $drawingHeight := h_0$ 
4   $localWidth := w_0$ 
5   $localHeight := h_0$ 
6   $lastRegion := r_0$ 
7  for ( $r \in rs \setminus r_0$ )
8      // Calculate and evaluate possible placements.
9       $LR := ps \cup \{r | x(r) = localWidth + s, y(r) = y(lastRegion)\}$ 
10      $LB := ps \cup \{r | x(r) = localWidth, y(r) = localHeight + s\}$ 
11      $DR := ps \cup \{r | x(r) = drawingWidth + s, y(r) = 0\}$ 
12      $DB := ps \cup \{r | x(r) = 0, y(r) = y(drawingHeight) + s\}$ 
13
14     // Greedily get the best placement based on scale measure,
15     // area, aspect ratio.
16      $ps := getBestPlacement(LR, LB, DR, DB)$ 
17
18     // Update drawingWidth, drawingHeight, localWidth,
19     // localHeight and lastRegion based on chosen placement
20      $drawingWidth := \max(drawingWidth, x(r) + w(r))$ 
21      $drawingHeight := \max(drawingHeight, y(r) + h(r))$ 
22      $localWidth := x(r) + w(r)$ 
23      $localHeight := y(r) + h(r)$ 
24      $lastRegion := r$ 
25  $w := drawingWidth$ 

```

---

#### 4. Model Order in Rectangle Packing



**Figure 4.12.** Disregarding the model order can produce a more compact drawing. By disregarding the model order, as shown in the two packings in Figure 4.12a, regions can be stacked, which is not possible when preserving model order, as seen in Figure 4.12b.

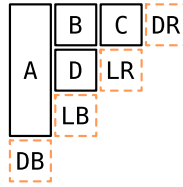
Since the final drawing should take order into account to improve stability and create desired secondary notation, so should the width approximation step. Specifically, I do not recommend ordering regions by size since SCCharts region sizes change drastically when expanding or collapsing regions. Hence, rather than doing static width approximation, I simulate the region placement with the greedy  $\mathcal{O}(n)$  heuristic in Algorithm 1.

The *greedy width approximation heuristic* packs region in four possible positions, as visualized in Figure 4.13. A new region may either be right of the current node (LR), which simulates a placement in the same row or subrow, under the current node (LB), which simulates a stack, right of the whole drawing (DR), which simulates creating a new stack in the current row, or under the whole drawing (DB), which simulates the creation of a new row. After each placement step, the algorithm greedily chooses the best of the possible placements based on scale measure with area and aspect ratio difference as a secondary and tertiary criteria [DLH+23]. After determining the best placement the algorithm continues with the next region. This approach relaxes the row structure and reading direction and only fulfills the basic ordering requirement (see Equation 4.1.1) to approximate the target width.

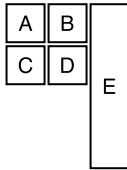
These four positions simulate how a real algorithm might pack the rows. However, Algorithm 1 is not perfect and sometimes overestimates the width since the greedy packing is not very “smart,” as seen in Figure 4.14 and 4.15. Since each incremental step in the greedy algorithm optimizes the scale measure, Figure 4.14a packs regions A to D to form a square and does not consider that region E might be high enough to stack them, as



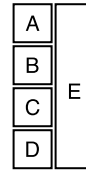
#### 4.4. Rectpacking Heuristic



**Figure 4.13.** The four candidate positions for placing region E after regions A to D were already placed. The candidate positions are visualized in dashed-orange: right of the last region (LR), below the last region (LB), right of the drawing (DR), and below the drawing (DB).



(a) Greedy placement



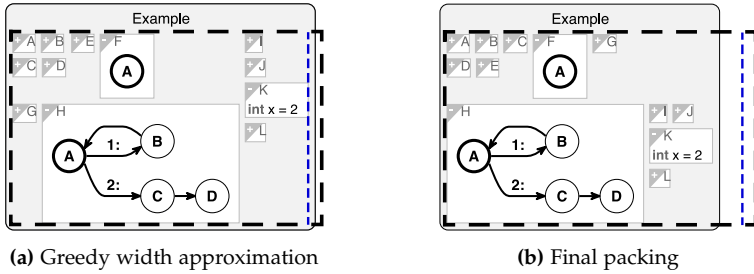
(b) Final packing

**Figure 4.14.** The greedy nature of the width approximation step may overapproximate the target width.

visualized in Figure 4.14b. For the running example, the required width is also overestimated since region G is greedily placed below the drawing, as seen in Figure 4.15a. For other cases, such as Figure 4.16, the greedy width approximation algorithm underestimates the required width since it relaxes the row structure and reading direction constraints. In Figure 4.16a the greedy width approximation groups elements of similar shape by violating the reading direction. As a result, region n2 and n4 are next to each other grouped together with the slim regions while n3 is placed at the bottom together with the wide regions.

The solution to over- or underestimated width is control. Since users typically desire control—especially if complicated algorithms are at play—the user has the option to specify the target width directly such that the width approximation step can be skipped. Controlling the target width and with it the aspect ratio of the final drawing is one way to express

#### 4. Model Order in Rectangle Packing



**Figure 4.15.** Width approximation of the running example. The desired aspect ratio is marked in dashed-black, the approximated target width is marked in dashed-blue. The approximation in Figure 4.15a overestimates the target width as seen in the final packing in Figure 4.15b. This may lead to drawings that are too long.

secondary notation, even though it is not very direct. Hence, Section 7.2 further illustrates how controlling the target width can be done interactively.

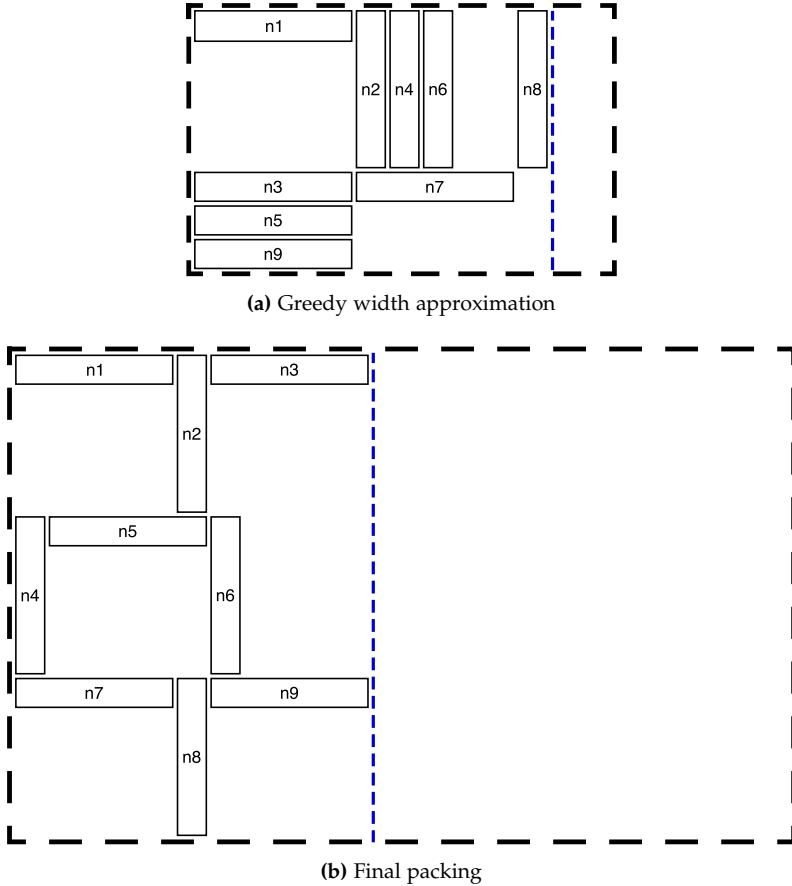
#### 4.4.2 Rectpacking Placement and Grouping Step

The width approximation step reduced the region packing problem to a strip packing problem. The following placement step further reduces the problem complexity by approximating the target height for each row. Moreover, the placement step groups regions into blocks of similar sized regions allowing space efficient placement into subrows.

Figure 4.17 shows the packing after the placement step. Algorithm 2 places regions next to each other (see regions A to G and line 6) until the next region would exceed the target width and a new row has to be created, which is the case for regions H and K. Region F, H, and K, which dominate their respective rows, define the target height of their rows. In the following compaction step, this row height determined by the dominant elements will only be reduced to limit the complexity of the algorithm and improve its runtime.

In addition to approximating the row height, the placement step evaluates the height difference of regions and groups regions with “similar height” (line 7) that fit in the same row into a *block*, which is a collection of

#### 4.4. Rectpacking Heuristic



**Figure 4.16.** Greedy width approximation of the worst case example compared to the final packing. The width approximation step in Figure 4.16a underestimates the target width. Hence, the final drawing in Figure 4.16b is too high.

#### 4. Model Order in Rectangle Packing

---

**Algorithm 2:** Simple placement algorithm for the rectpacking heuristic [DLH+23].

---

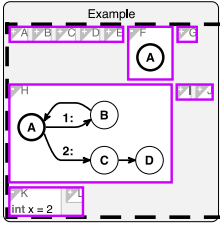
**Input:** Regions  $rs$ , width  $w$

**Output:** Regions  $rs$  placed into rows  $rows$  to approximate the row height. Rows are grouped into stacks, stacks into one block consisting of one subrow with all regions of a block based on height similarity.

```
1 rowWidth, rowHeight, rowLevel = 0
2 currentBlock =  $\emptyset$ 
3 currentStack.create(currentBlock)
4 currentRow.create(currentStack)
5 for (  $r \in rs$  )
6     if (  $r_w + rowWidth \leq w$  )
7         if ( fitIntoBlock( $r$ , currentBlock) )
8             // Place region in current block.
9             currentBlock.add( $r$ )
10        else
11            // Place region in new block.
12            currentBlock.create( $r$ )
13            currentStack.create(currentBlock)
14    else
15        // Create new row.
16        currentBlock.create( $r$ )
17        currentStack.create(currentBlock)
18        currentRow.create(currentStack)
19        rows.add(currentRow)
20        rowLevel = rowLevel + rowHeight
21        rowHeight, rowWidth = 0
22    // Add region to the new row.
23     $r_x = rowWidth$ 
24     $r_y = rowLevel$ 
25    rowWidth +=  $r_x$ 
26    rowHeight = max(rowHeight,  $r_h$ )
```

---

## 4.4. Rectpacking Heuristic



**Figure 4.17.** In the placement step, regions are placed as suggested in the simple heuristic, while grouping similar sized regions into blocks (magenta). Region F dominates the height of the first row and will be its approximated target height. Region H dominates the second row and determines its height. The approximated target width is marked in dashed-blue, and the desired aspect ratio in dashed-black.

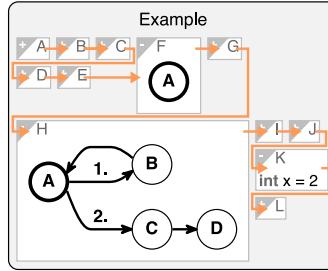
subrows in a stack. Regions have similar height if their height is 50% more or less than the average height of the block. This is important since assigning the regions of a block with similar height into the same subrow is space efficient and improves the scale measure resulting in a better readability. Additionally, grouping similar sized elements may often be intended since they might be similar. Hence, a grouping into blocks potentially improves secondary notation.

After placement, the packing structure consists of rows, which consist of stacks of one block, which consists of one subrow of regions. Hence, the blocks (magenta) of Figure 4.17 are also the stacks and subrows of the packing. In addition to the grouping into blocks, the minimum, current, average, and maximum height and width of stacks, blocks, and subrows are calculated during the placement step, as part of the create and add methods in Algorithm 2. Hence, the following compaction step updates these values rather than recomputing them from scratch. The grouping into blocks and the pre-computation and continuous update of its size will make it easier to compute possible shapes of a block in the following compaction step ensuring that the tool the algorithm will be employed in remains reactive.

### 4.4.3 Rectpacking Compaction Step

With the target width and the row height approximated, the rectangles can be compacted and finally be placed. This compaction is the main step to improve the scale measure and with it the readability of a packing compared to the simple heuristic.

#### 4. Model Order in Rectangle Packing



**Figure 4.18.** The packing utilizes the four available region positions during the compaction step. The next region is marked with orange arrows.

The compaction step uses the four different positions for a new region as defined in Equation 4.1.2 to 4.1.5 and visualized in Figure 4.18.

Rule 1: Right of the preceding region, in the current subrow. (4.4.1)

Rule 2: Right of the preceding region, in a new stack. (4.4.2)

Rule 3: In a new subrow. (4.4.3)

Rule 4: In a new row. (4.4.4)

E.g., region B is placed in the same subrow (Rule 1, Equation 4.4.1) as A. C is also placed in the same subrow (Rule 1) as B. D is placed in a new subrow (Rule 3) and E is again right of D in the same subrow (Rule 1). F begins a new stack (Rule 2, Equation 4.4.2), as does G. Region H begins a new row (Rule 4, Equation 4.4.4) while region I begins a new stack (Rule 2). J is in the same subrow (Rule 1) as I and K is in a new subrow (Rule 3, Equation 4.4.3) as is L.

Note that the packing is only structured by rows, stacks, and subrows. One cannot see blocks based on the region placement. Hence, it is not distinguishable by the drawing alone whether region I to L are in the same block or in several.

Algorithm 3 depicts the compaction algorithm that works based on these rules to create a left-to-right reading direction, which works as follows. For each block inside a row, the algorithm decides how the current block should be placed based on the next block.

---

**Algorithm 3:** Compaction algorithm for the rectpacking heuristic [DLH+23].

---

**Input:** Regions placed into rows and grouped into stacks, blocks, and subrows with one block per stack and one subrow per block  $rs$ , width  $w$

**Output:** Compactly placed regions grouped into stacks, blocks, and subrows  $rs$

```

1 rows = getRows(rs)
2 for ( row ∈ rows )
3     // Initialize row
4     for ( block ∈ row.blocks )
5         Block next = block.nextBlock()
6         // "Steal" regions from the next block.
7         block.addRegions(next)
8         if ( fitBelow(block, next) )
9             block.stack.add(next)
10        else if ( fitRight(block, next, w) )
11            block.stack.drawInRowHeight()
12            block.placeRight(next)
13        else
14            block.stack.drawInRowWidth(w)

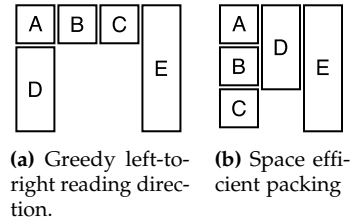
```

---

First, the algorithm checks whether the current block can “steal” regions from the next block (line 7), which might be necessary since the placement step might split similar height regions at the row border. This has to be checked again during compaction since compaction creates more available space in a row. Note that a region can only be stolen if the current and next block are not in the same row, and if the remaining row width and height allow to fit the stolen regions. E. g., a block cannot steal a region if it would exceed the row height. Moreover, the remaining width and height must fit the new regions. The pre-calculated stack, block, and subrow minimum and maximum bounds may already determine whether the new regions will fit into the current row. If this is not the case, a simple strip packing

#### 4. Model Order in Rectangle Packing

**Figure 4.19.** Preferably placing regions left-to-right may waste space, as seen in Figure 4.19a. Figure 4.19b may be more space efficient, but its reading direction is top-to-bottom and therefore undesired.



algorithm that uses the remaining width determines whether the addition of the stolen regions would fit the row height. Note that first checking whether the whole block can be stolen significantly reduces time for the “stealing” step since this prevents checking each region individually.

Next, one decides how to place the current block. Fitting a block below (line 8) places the current block as flat as possible, which emphasizes the preferred left-to-right reading-direction and therefore improves secondary notation. However, placing a block preferably below may not be ideal in terms of space efficiency and scale measure, as seen in Figure 4.19. Here, Figure 4.19a shows how a flat block followed by a slim block may waste a lot of space. Disregarding the reading direction, as shown in Figure 4.19b, may produce a more compact drawing at the cost of secondary notation and stability since Figure 4.19b does not make it as evident as Figure 4.19a that region C is before region D. Also note since the compaction algorithm places the current block as flat as possible before fitting a block below it, the heuristic will not create packings where the current block has two subrows when fitting the next block below.

If the next block cannot fit below the current one, which may be the case if their total minimum height exceeds the row height, or if fitting it below would exceed the row width, the next block is placed to the right in a new stack. If the next block fits right (line 10) of the current block, the current block is drawn as high as possible in the row height minimizing its required width and potentially creating a more compact packing. E. g., in Figure 4.18, regions A to E cannot fit region F below. Hence, the algorithm places regions A to E such that they fully utilize the row height and form two subrows.

Fitting the current block into the row height requires solving a vertical



#### 4.4. Rectpacking Heuristic

strip packing problem while maintaining the left-to-right reading direction during `fitInRowHeight`. Since one does not know the width of the current block (regions A to E), one has to approximate the block width using the minimum and maximum width and height as well as a search algorithm for the correct packing to fit the row height. I. e., the algorithm tries different packings using different target widths and use the result to decide which target width to try next. The region widths and heights can here be used to find the correct block width without using binary search on all real numbers between the minimum and maximum width. E. g., the heights of regions A to E do not allow stacking three of them below each other. Hence, we only need to try how to place them into two subrows, which boils down to the choice whether the block would be wider if region C would be placed in the first or second row. This means that one has to determine whether the aggregated region width of regions A to C is smaller than the aggregated region width of regions C to E.

If the next block does not fit below or to the right of the current block, it must be placed in the next row (line 13). This may be the case if a region is too high to fit in the row or if there is not enough width available as it is the case for region H in Figure 4.18. If the next block is in a new row, the current stack is drawn to fully utilize the remaining row width to create a left-to-right reading direction. This has linear runtime since the desired width for the horizontal strip packing problem is known, the region order is fixed, and regions are placed in subrows. Note that dynamically adjusting the row width may be necessary to find the optimal solution regarding scale measure if the width is underestimated. However, this is not done as part of the base algorithm but I discuss this as a possible optimization in Section 4.4.5.

The resulting drawing after compaction can be seen in Figure 4.20. This drawing has a stable and consistent reading direction, i. e., one primarily reads from left to right, and shows the desired secondary notation given by the model order. Moreover, it is compact and has a high scale measure such that the packing fits the desired aspect ratio visualized in dashed-black.

I want to highlight the differences between this drawing and the one in Figure 4.11 on page 47 in terms of readability, stability, secondary notation, and control. Figure 4.11 orders the regions A to E using a top-to-bottom read-

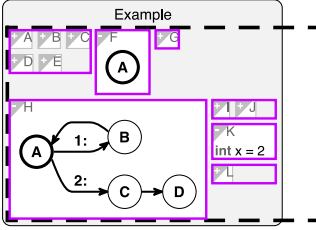
#### 4. Model Order in Rectangle Packing

ing direction, which potentially threatens stability and secondary notation. In Figure 4.20, one primarily reads through the regions from left-to-right, and secondary reads them from top-to-bottom. Hence, a user might get confused by this primary reading direction change in Figure 4.11 since it is inconsistent with the left-to-right reading direction in the rows. Users may assume a different order of regions and perceive a different grouping of regions, as reported by SCCharts developers and experienced by myself. E.g., the height of region F dominates the row height and visually creates a row. The regions of A to E are stacked against F and visually create subrows that are not really there. Since the regions have the same height, they create alignment and visual subrows while being placed in separate stacks. Region A to E seem to be ordered A, C, E, B, D if one considers a left-to-right reading direction. In Figure 4.20, however, regions A to E create subrows rather than stacks such that their intended ordering is consistent. Since the user is familiar with the left-to-right drawing style, it is always clear where to continue if rows and subrows are primarily used, with a few exceptions presented in Section 4.7.3.

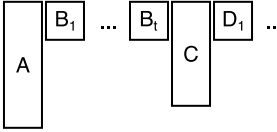
The downside of a consistent reading direction is, however, that forming stacks, as in Figure 4.11, is far easier than forming subrows since one only needs to solve trivial  $\mathcal{O}(n)$  vertical strip-packing problems without considering the reading direction. Therefore, such a compaction algorithm is quicker and may even result in a slightly better scale measure and readability at the cost of stability and secondary notation. To create subrows one needs to search for the correct stack width. Searching for the correct width, as it is necessary when forming subrows with `fitInRowHeight`, requires an  $\mathcal{O}(t \log t)$  search algorithm with  $t$  being the size of the block. This is required since subrows should be evenly populated while reducing the width and still fitting into the row height.

Both heuristic, however, might suffer from a lack of control. Since model order is one-dimensional, but a region packing has two dimensions, model order can only control the order of regions. However, model order cannot control when a new row, stack, or subrow should be created and cannot control the desired target width. Hence, layout constraints might be necessary in addition to using model order to increase the control users might have on the drawing, as detailed in Section 7.2.

## 4.4. Rectpacking Heuristic



**Figure 4.20.** The final packing after the compaction step emphasizes the left-to-right reading direction compared to the packing in Figure 4.11 with blocks marked in magenta and the desired aspect ratio in dashed-black.



**Figure 4.21.** Rectpacking takes longest if the drawInRowHeight step that searches for a correct strip-packing width executes for as many regions as possible.

### 4.4.4 Worst Case Rectpacking

Before continuing with the algorithm, I want to quickly sketch the inaccuracies of the proposed rectpacking heuristic. As stated above, the rectpacking heuristic makes several assumptions during compaction and placement, which might be suboptimal. In the following, I want to highlight steps that may increase the runtime or result in inefficient use of space decreasing the scale measure and with it the readability. Moreover, I present how the heuristic may deal or cannot deal with cases that produce these packings.

#### Worst Case Runtime

Figure 4.21 shows how the costly drawInRowHeight function (see line 11) may increase the runtime of the algorithm. Here, region A dominates the row height, the current block B consists of arbitrary many small regions  $B_i$ , and block C is high such that it cannot fit below B. If this pattern of current and next block continues, every second block has to be slimmed and has to search for a correct target width with maximum runtime of  $\mathcal{O}(t \log t)$  with  $t$  being the size of the block. This results in the worst case runtime of the rectpacking heuristic of  $\mathcal{O}(n \log n)$  if all but the first region are part of the same block while the first region is big.

Note that pre-calculating the maximum, minimum, and average width

#### 4. Model Order in Rectangle Packing

and height of blocks and regions in them may be used to speed up the search. If the row height exceeds the maximum height, the width is bounded by the maximum region width. The average width and height allows to guess a potentially good target width using the formula of the simple heuristic, which works well for regions of similar size. Additionally, the regions' widths and heights and their packing into subrows can be used to find sensible values to slim or widen the subrows similarly to how the row width might be revised, as illustrated in Figure 4.24.

##### **Worst Case Space Efficiency**

In terms of region area relative to required area, alternating flat and slim regions produce the worst result if ordering constraints are followed, as seen in Figure 4.22a. If the slim regions have a width of  $\varepsilon > 0$  and a height approaching infinity and the wide regions having a height of  $\varepsilon > 0$  and a width approaching infinity, then the region area per required area would be approaching zero. Hence, the drawing of such a packing would consist almost entirely of whitespace, which can only be prevented by relaxing the reading direction, the order constraints, the row structure, or the dominant element constraint.

Figure 4.22a visualizes additional shortcomings of the heuristic. The target width in Figure 4.22a is underestimated since the approximation step depicted in Figure 4.22b has more freedom to place the regions and may group them by shape to greedily optimize the scale measure. This is the case since the greedy width approximation step does not order regions inside subrows but only adheres to the simple ordering criteria. A succeeding region must be either vertically or horizontally before the preceding region.

Not considering row-dominant elements and not creating subrows of similar sized regions can also improve the packing. In Figure 4.22c, the slim regions  $n_2$  and  $n_4$  can be placed compactly next to each other since there does not have to be a dominant element. Moreover, placing regions  $n_4$  to  $n_6$  in the same subrow disregarding their height difference further compacts the drawing. However, even though this produces a more compact packing, the ordering of elements is no longer recognizable and violates the model order. Specifically, it is not recognizable whether  $n_7$  should be before  $n_8$  or

## 4.4. Rectpacking Heuristic

the other way around based on their placement. Hence, a row-dominant element is necessary to make the reading direction recognizable.

To summarize, even with a dominant element and a clear row-structure, Figure 4.22a can still be improved if width approximation would be more accurate or the target width would be revised (see Section 4.4.5), as seen in Figure 4.22d. However, note that the heuristic would never create the drawing in Figure 4.22d since it would always place  $n_2$  in the same row as  $n_1$  if the width permits it. The rectpacking heuristic does hence need control mechanisms to create the packing in Figure 4.22d, as detailed in Section 7.2.

### Worst Case Row Height Approximation

The height approximation step that is executed during the placement step (see Section 4.4.2), might again produce arbitrarily bad results, as visualized in Figure 4.23.

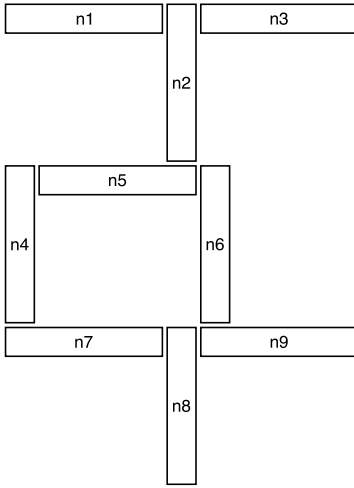
If a high region is followed by several small regions that are first placed next to each other, the target width may be fully utilized by the first placement, as seen in Figure 4.23a. During compaction, the small regions are compacted such that there should be enough width available to fit the next blocks inside the row. If, however, the height of the next region is higher than the row height, the drawing will be too slim, as seen at the placement of the  $B_i$  and  $D_i$  nodes in Figure 4.23b.

Since the height of the first row is defined by A, region C does not fit in the first row. Assuming that the small regions are arbitrarily flat and many, this pattern may produce an arbitrarily bad packing if the row height is not revised to fit C, which is one of the problem I tackle in the following.

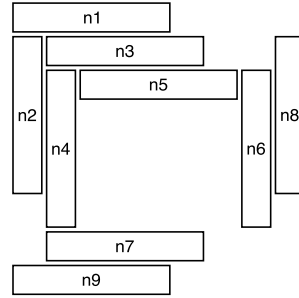
### 4.4.5 Improving the Compaction

The rectpacking heuristic presented above might create packings with a suboptimal scale measure because of four limitations of the heuristic. First, the approximated target width might be wrong. Second, the approximated row heights might be wrong. Third, the packing restrictions might be too constraining. Or, fourth, the grouping into blocks might be suboptimal.

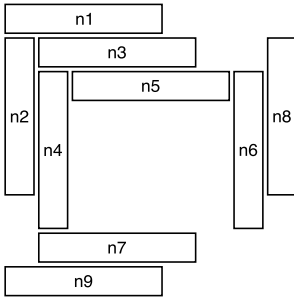
#### 4. Model Order in Rectangle Packing



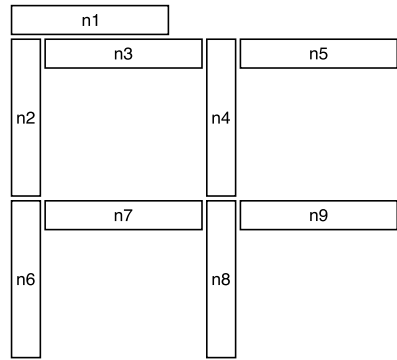
(a) Rectpacking solution



(b) Rectpacking width approximation



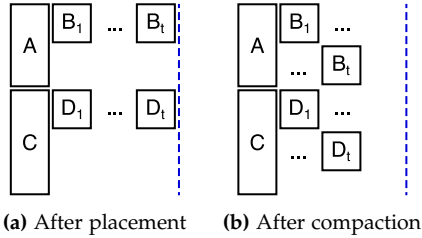
(c) Optimal packing without row-dominant elements



(d) Optimal packing with row-dominant elements

**Figure 4.22.** In Figure 4.22a, the alternating flat and slim regions create much unused whitespace. At the same time, the required width is underestimated since the greedy width approximation does not consider dominant elements in rows, as seen in Figure 4.22b. The same also applies to the optimal packing without considering row-dominant elements in Figure 4.22c. Even with row-dominant elements, the optimal solution in Figure 4.22d is much better.

#### 4.4. Rectpacking Heuristic



**Figure 4.23.** The fixed row height after placement may result in a bad packing. The placement step in Figure 4.23a fully utilizes the target width. During compaction, each row is compacted and might have space for new regions, which cannot be added if they exceed the row height, as seen in Figure 4.23b.

In the following, I present how one can solve the first two issues to improve the readability of a packing by revising the target width and the row height. This may happen at the cost of computation time if time is not critical or models are small to not threaten reactivity of a diagram view this algorithm is used for. For SCCharts these compaction steps seem promising since analysis of SCCharts models showed that region packing problems are typically small, as detailed in Table 4.1 on page 84.

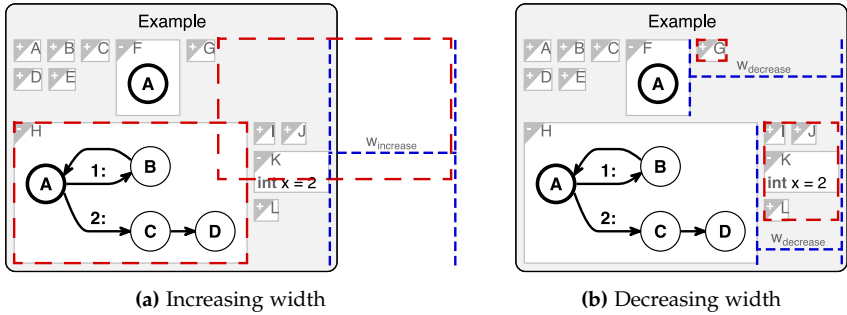
#### Revising the Approximated Width

The greedy width approximation algorithm described in Section 4.4.1 may not yield the desired target width because it is greedy and does not utilize the row structure. Since the compaction step, however, utilizes the row structure, multiple iterations of the compaction step can be used to approximate the correct target width to increase the scale measure to improve readability, as seen in Figure 4.24.

Figure 4.24a illustrates how the target width can be adjusted if it was underestimated and the resulting aspect ratio is too low. In this case, each row calculates the width  $w_{\text{increase}}$ . The width  $w_{\text{increase}}$  corresponds to the width increase if the first stack of the next row would be moved into the current row, as visualized in dashed-red for region H. Increasing the target width by the minimal additional width  $w_{\text{increase}}$  will hence create a slightly wider packing and should hence be the new target width for the compaction step.

Overestimating the target width or having a too high aspect ratio, requires us to find a suitable reduced target width, as seen in Figure 4.24b. Similarly to the case described above, reducing the width by the minimal

## 4. Model Order in Rectangle Packing



**Figure 4.24.** Target width revision visualized. A second compaction step can use the already existing stack, block, and subrows to revise the target width. Here, the width  $w_{increase}$  and  $w_{decrease}$  (dashed-blue) marks the target width change for each row based on additional stacks (dashed-red) from the next row.

$w_{decrease}$ —the minimal width reduction that would result in a change of the overall width—finds the next target width that may produce a better scale measure.

The compaction process ends if an increase or decrease of the target width does not result in a better scale measure and a potentially local minimum is found. Additionally, one can restart the compaction after the increase or decrease failed with another decrease or increase attempt if this yields a potential new target width between the current width and the old target width.

Since revising the target width may execute the potentially costly compaction step multiple times, it may drastically increase the runtime. Hence, revising the target width is only advised if time is not an issue or the region packing problems posed by a modeling language are small.

### Revising the Row Height

Similarly to the target width, the row height can be revised by repeating compaction steps to improve the scale measure and with it the readability of a packing.

As proposed in Section 4.4.2, the row height has an upper bound after



#### 4.4. Rectpacking Heuristic

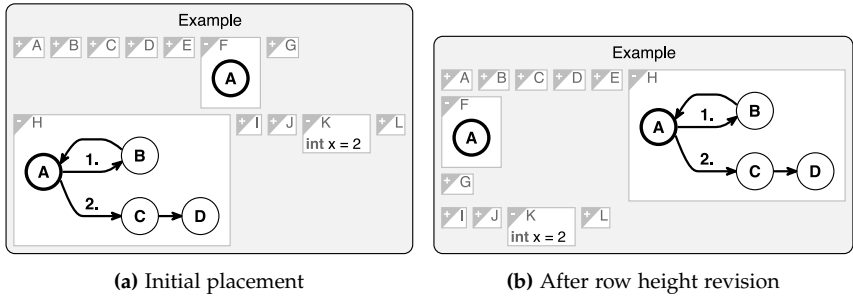
the placement step to reduce the runtime of the algorithm. This has the advantage that stealing regions or blocks from the next row does not require compacting and placing already placed blocks of the current row. A region that is too big to fit into a row cannot be stolen during compaction. However, as a result packings may become arbitrarily bad, as presented in Section 4.4.4. Hence, I propose to improve Algorithm 3 to optionally revise the row width to improve the packing. Each time a region from the next row might not be added to the current row because it would fit into the row width but would exceed the row height, we add the region and compact the row using the height of the new dominant element as the row target height. However, this means, all already placed regions have to be compacted again.

Recalculating the row height allows to fully utilize the additional row height and prevents ONO packings at the cost of additional computation time. However, if one high and several small regions alternate, as it is the case in Figure 4.23, one revises the row height every second block. In the packing in Figure 4.23, this also repeats the costly `drawInRowHeight` compaction step every other second block. Additionally, in a packing where the height doubles with each region, all regions have the same width, and the target width is twice the region width every rectangle requires a row height revision.

Figure 4.25 visualizes the row height revision. The placement step in Figure 4.25a does not leave space for region H in the first row. Hence, region H cannot be moved into the first row without revising the row height, even though one can easily imagine a packing that would compact regions A to G that would leave enough width to do so. Revising the row height, as seen in Figure 4.25b, allows placing region H in the first row and produces a better scale measure if the desired aspect ratio is 1.9. As a side effect, row height revision allows utilizing the target width more effectively. E. g., in Figure 4.25a region H might fit into the width of the first row but is limited by its height.

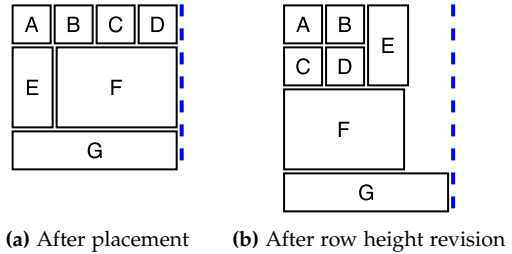
Note that Figure 4.25b additionally illustrates a downside of the rectpacking heuristic and its compaction step. Since the block consisting of regions A to E is drawn as flat as possible a lot of space is wasted since regions F and G are stacked and not put in the same subrow because of

#### 4. Model Order in Rectangle Packing



**Figure 4.25.** The initial placement in Figure 4.25a only allows to move region H in the first row if the row height is revised, as seen in Figure 4.25b.

**Figure 4.26.** The problem of row height revision visualized with dashed-blue target width. Figure 4.26a shows the packing after the placement step, which would corresponds to the final packing without row height revision. Revising the row height results in Figure 4.26b.



their height difference. Here, row height revision produces a worse scale measure when comparing Figure 4.25b to Figure 4.20 on page 61 with a desired aspect ratio of 1.3. Therefore, this option should only be activated if approximating the row height create a ONO packing or if this is desired by a user.

Additionally, revising the row height may produce ONO packings, as seen in Figure 4.26. Since region E is moved into the first row, only the height of the first row and with it the drawing height increases while the drawing width is not reduced, which creates a smaller aspect ratio and potentially a worse scale measure. Hence, this improvement step is not as powerful as the compaction iterations presented above.

#### 4.4.6 Rectpacking Whitespace Elimination Step

Whitespace elimination is the last step of the rectpacking heuristic. After the drawing is sufficiently compacted, the whitespace elimination algorithm may fill gaps between the regions and eliminate the whitespace, as seen in the SCChart in Figure 4.6c. Compared to the whitespace elimination of the simple heuristic in Figure 4.2, the rectpacking heuristic additionally aims to prevent abnormally formed regions as a result of whitespace elimination.

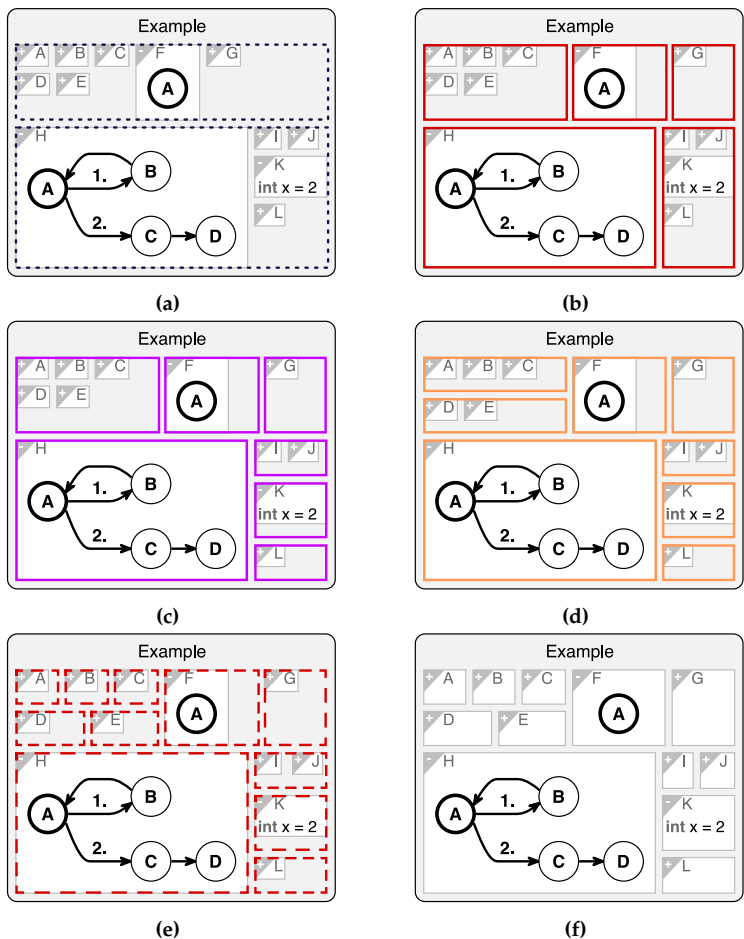
In the following, I present an algorithm that utilizes the row structure given by the rectpacking problem and equally distributes the available whitespace using the row structure. Moreover, I present a more general approach to eliminate whitespace if the packing uses the four candidate position described in Section 4.1 to be used for packing algorithms without explicit rows, stacks, and subrows.

The rectpacking heuristic eliminates the whitespace by increasing the width and height of regions while potentially moving regions to the bottom right to accommodate previous regions that increase in size. Doing so does not change the reading direction of elements such that stability and secondary notation are not threatened. Figure 4.27 visualizes the whitespace elimination algorithm with a runtime of  $\mathcal{O}(n)$ .

Here, one divides the additional row height evenly between all rows, which is skipped in Figure 4.27 since there is no additional row height. Additionally, one increases the width of each row (dotted-black) to the maximum row width resulting in additional width for the row, as visualized in Figure 4.27a. Figure 4.27b visualizes the whitespace elimination for stacks (red). Since one knows the row height and width, increasing the stack height of every stack in every row to the row height and equally assigning the additional width of each row to the stacks distributes the stacks equally in each row. This is the first step that makes sure that regions do not become abnormally formed and therefore increases readability.

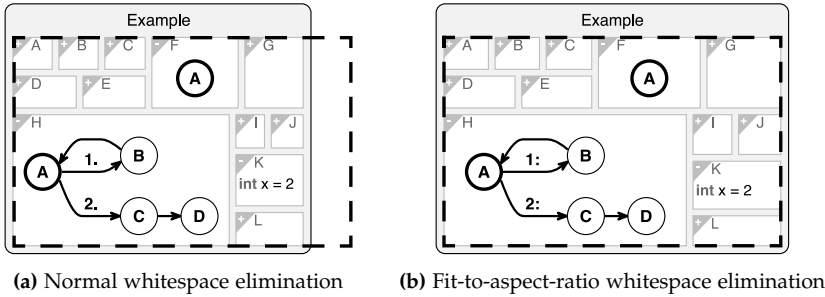
Figure 4.27c and 4.27d visualizes whitespace elimination for blocks (magenta) and subrows (orange). Depending on whether stacks consist of blocks or directly of subrows, both increase their width to the block width. The potential additional height of the stack is again evenly distributed among blocks or subrows to prevent abnormally formed regions.

# 4. Model Order in Rectangle Packing



**Figure 4.27.** Figure 4.27a shows how rows are expanded. Figure 4.27b shows how the stacks are moved to equally divide the whitespace. Figure 4.27c, Figure 4.27d, and Figure 4.27e show how the whitespace is divided among blocks, subrows, and regions until all whitespace is eliminated in Figure 4.27f.

#### 4.4. Rectpacking Heuristic



**Figure 4.28.** Whitespace can also be eliminated to let a packing fit into a desired aspect ratio (dashed-black).

Figure 4.27e shows how the additional subrow width is evenly distributed to each region (dashed-red) and how the region height increases to the subrow height, as seen in the final result in Figure 4.27f. Moving whole subrows, blocks, stacks, and rows based on the row structure allows moving regions during whitespace elimination additionally to resizing them. As a result, all regions may increase their width and height and not only the regions without direct neighbors, as it is the case in Figure 4.2 on page 37. E. g., in Figure 4.27a, region B and D fully enclose region A. Using the approach of the simple heuristic, one could only resize region D and E but not the regions in the first subrow above it. If the whole subrow is, however, moved, the algorithm equally distributes the available height among all regions A to E without disrupting the row structure.

This whitespace elimination algorithm can also be used to resize the whole drawing to the desired aspect ratio while distribution additional width and height evenly, as seen in Figure 4.28. This is achieved by setting the drawing width and height accordingly, which potentially results in additional row width or height.

#### Whitespace Elimination without a Row Structure

If no explicit row and subrow structure exists, a more general whitespace elimination approach has to be used, as seen in Figure 4.29. Here, it would

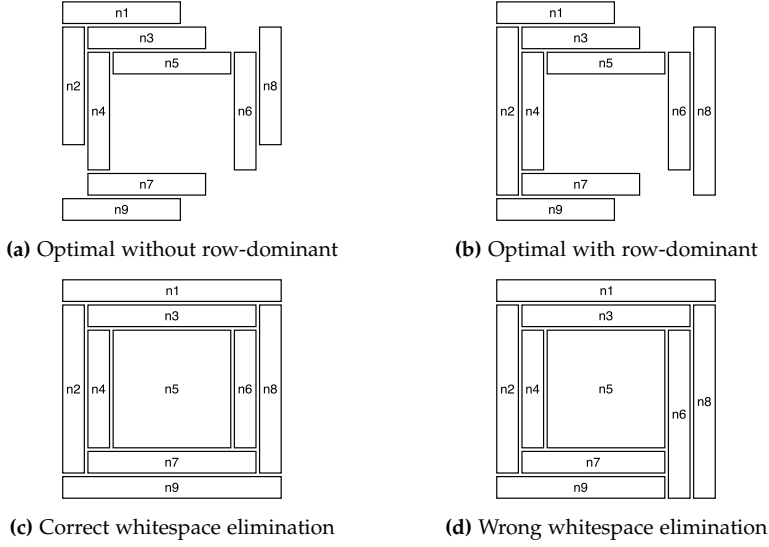
#### 4. Model Order in Rectangle Packing

only be possible to equally assign the available whitespace if one calculates the rows, stacks, and subrows of the placement first. Hence, I propose an algorithm that is able to eliminate the whitespace without a row structure.

By iterating over the regions in reverse order and increasing the region width and height as much as possible without causing overlaps or exceeding the drawing bounds, the whitespace of Figure 4.29a or its row-dominant variant in Figure 4.29b is eliminated, as depicted in Figure 4.29c. Handling the regions in reverse order is here essential. By handling regions in order, the resulting packing may create wrong alignments and subrows that imply a wrong reading direction and hence threaten stability and secondary notation, as seen in Figure 4.29d. Here,  $n_6$  compromises the height of a subrow, which consists of  $n_4$ ,  $n_5$ , and  $n_6$ , and the height of the row consisting of  $n_2$  to  $n_8$ . Hence, the subrow and row are no longer recognizable. If we handle regions in reverse order, they cannot cross a subrow boundary. The previously empty space was already filled since all region to the bottom right are succeeding regions, which were already expanded. These succeeding regions were, hence, already resized to fill potential gaps. However, since every region needs to check all succeeding regions to find out how much its width or height can increase, the runtime of the general whitespace elimination algorithm is in  $\mathcal{O}(n^2)$ . If a row structure is present, whitespace elimination can be solved in  $\mathcal{O}(n)$ .

Note that calculating the row structure in advance can be done in  $\mathcal{O}(n)$ . One can create a row structure in post-processing by iterating over all nodes in order. Depending on their alignment, regions are assigned to the same subrows. Here, it is essential to utilize the order of nodes to recognize when a new stack starts to prevent the alignment anomaly described in Section 4.7.3 from occurring. A region is in the same row or subrow if it aligns horizontally with its predecessor. If a region is below its predecessor, it must either be in the same stack in a new subrow or in a new row, which can be recognized by the already created rows or stacks. A region that is right of its predecessor and above it must be in a new stack and aligned at the current row level. Note that this always produces a row structure if one already exists in the placement. However, a packing that only consists of rows without subrows is not differentiable from a packing that consist only of one stack with several subrows without row dominant elements.

## 4.5. Solving Region Packing Optimally



**Figure 4.29.** Whitespace elimination without an explicit row structure is possible by traversing the regions in reverse order and increasing their height and width as much as possible. Figure 4.29a shows a potential packing without row-dominant element and Figure 4.29b with row-dominant elements. Whitespace elimination for both models works by traversing regions in reverse order can be seen in Figure 4.29c. Figure 4.29d illustrates how traversing regions in order may produce a misleading row structure.

## 4.5 Solving Region Packing Optimally

The proposed heuristics in Section 4.3 and Section 4.4 are not perfect in terms of readability but rather aim for stability, secondary notation, and computation time.

To be able to evaluate their quality based on readability, I propose to compare them to the optimal solution of the region packing problem. Hence, I present the region packing problem as a maximization problem that can be used to find the optimal solution based on the scale measure using the actual width and height  $w_a$  and  $h_a$  as well as the desired aspect ratio with

#### 4. Model Order in Rectangle Packing

Equation 4.5.1 as the goal function:

$$SM = \min\left(\frac{DAR}{w_a}, \frac{1}{h_a}\right) \quad (4.5.1)$$

To be comparable, the optimal packing should still prioritize the reading direction and create a row structure, use the four region candidate positions, and consider a row-dominant element as defined in Section 4.1.3. Hence, I need to describe the potential position for each region such that they are placed in row, stacks, and subrows with row-dominant elements and a stable and consistent ordering. For this I introduce the following variables.

*s* The spacing between regions. This is the same for all regions. The algorithms for width approximation, placement, compaction, and whitespace elimination also consider the spacing. The presented algorithms, however, omit it since it is not relevant to the solution. Since it does make a small difference regarding the optimal solution, I include it here for completeness.

*currentHeight<sub>i</sub>* This describes the current maximum height of the drawing after placing regions  $r_1$  to  $r_i$ , including the spacing. This is hence required to place an element in a new row (Rule 4).

*rowLevel<sub>i</sub>* This describes the top baseline of a row, which is 0 for the first row. The current row level is necessary to place regions according to Rule 2 in a new stack.

*startXStack<sub>i</sub>* This describes the  $x$ -coordinate at which the current stack begins. Placing elements in a new subrow in a stack (Rule 3) does hence need this information.

*endXStack<sub>i</sub>* This describes the end of the current stack in  $x$  direction including the spacing  $s$ . This is equal to the sum of *startXStack<sub>i</sub>* and the maximum subrow width of the stack. Hence, this coordinate together with the *rowLevel<sub>i</sub>* determine where a new stack would be placed (Rule 2).

*endYSubrow<sub>i</sub>* This describes where a subrow ends in  $y$  direction including the spacing. Hence, this is necessary to place regions in a new subrow



## 4.5. Solving Region Packing Optimally

together with  $startXStack_i$  (Rule 3).

$decision_i$  The decision determines the used rule and can hence also be used to further constrain the packing and improve the reading direction. A rule with a smaller number and smaller decision value means a region conforms more to a left-to-right reading direction. Hence,  $\min(\sum_{i=1}^n decision_i)$  should be used as a secondary criterion to the goal function to ensure that the left-to-right reading direction is preferred if it does not threaten readability.

These variables specify the position  $(x_i, y_i)$  for each region  $r_i$  based on the position and size of  $r_{i-1}$  and a start-position of  $x_1 = y_1 = 0$  for  $r_1$  while keeping track of the existing row structure and the chosen candidate positions for all regions. I. e., I describe the position of each region for the four candidate positions.

Placing a region  $r_i$  to the right of the region  $r_{i-1}$  (Rule 1, Equation 4.4.1) most prominently conforms to the left-to-right reading direction since the region is directly to the right of its predecessor and aligns at the top. Hence,  $decision_i$  is here lowest. Here, the end of the current stack in  $x$  direction, the end of the subrow in  $y$  direction, and the current height  $currentHeight_i$  potentially update based on the placement of the region and its size. Since Rule 1 (Equation 4.1.2) creates no new stack or row, the start of the stack and the row level do not change. This results in Equation 4.5.2 that describe the placement of region  $r_i$  based on Rule 1.

$$\begin{aligned}
 decision_i &= 1 \\
 x_i &= x_{i-1} + w_{i-1} + s \\
 endXStack_i &= \max(endXStack_{i-1}, x_i + w_i + s) \\
 startXStack_i &= startXStack_{i-1} \\
 rowLevel_i &= rowLevel_{i-1} \\
 endYSubrow_i &= \max(endYSubrow_{i-1}, y_i + h_i + s) \\
 currentHeight_i &= \max(currentHeight_{i-1}, endYSubrow_i) \\
 y_i &= y_{i-1}
 \end{aligned} \tag{4.5.2}$$

Placing a region in a new stack (Rule 2) also places it to the right of

#### 4. Model Order in Rectangle Packing

the preceding region creating a left-to-right reading direction, but does not align the region with the preceding one. The region is placed at the current row level after the end of the last stack. Consequently, the x-coordinate of the new stack is set to the end of the preceding stack. Moreover, the new stack ends after the current region. The subrow ends in  $y$  direction after the current region. However, the row level remains unchanged. The current height updates if the new region exceeds the height of all already placed stacks in the current row. This results in Equation 4.5.3 that describe the placement of region  $r_i$  based on Rule 2.

$$\begin{aligned}
 decision_i &= 2 \\
 x_i &= endXStack_{i-1} \\
 endXStack_i &= x_i + w_i + s \\
 startXStack_i &= endXStack_{i-1} \\
 rowLevel_i &= rowLevel_{i-1} \\
 endYSubrow_i &= rowLevel_i + h_i + s \\
 currentHeight_i &= \max(currentHeight_{i-1}, endYSubrow_i) \\
 y_i &= rowLevel_i
 \end{aligned} \tag{4.5.3}$$

Placing a region in a new subrow (Rule 3) requires knowing the start of the stack as well as the level of the new subrow, which corresponds to the end of the last subrow in  $y$  direction. As a result, the start of the stack and the row level remain unchanged. The end of the stack and the end of the subrow update if the size of the region exceeds the stack width or the subrow height. The current height may also increase if the current stack containing the new region now exceeds the height of all preceding stacks in the current row. This results in Equation 4.5.4 that describe the placement of region  $r_i$  based on Rule 3.

$$\begin{aligned}
 decision_i &= 3 \\
 x_i &= startXStack_{i-1} \\
 endXStack_i &= \max(endYCurrentStack_{i-1}, x_i + w_i + s) \\
 startXStack_i &= startXStack_{i-1} \\
 rowLevel_i &= rowLevel_{i-1}
 \end{aligned}$$

#### 4.5. Solving Region Packing Optimally

$$\begin{aligned}
 endYSubrow_i &= \max(currentSubrowEnd_{i-1} + h_i + s) \\
 currentHeight_i &= \max(currentHeight_{i-1}, endYSubrow_i) \\
 y_i &= endYSubrow_{i-1}
 \end{aligned} \tag{4.5.4}$$

Placing a region in a new row only requires knowing the current height of the drawing. Here, the  $x$ -coordinate and consequently the start of the new stack is zero. The end of the stack conforms to the width of the region and the spacing, the last current height defines the row level, which defines the end of the subrow and the new height of the drawing together with the region height and the spacing. This results in Equation 4.5.5 that describe the placement of region  $r_i$  based on Rule 4.

$$\begin{aligned}
 decision_i &= 4 \\
 x_i &= 0 \\
 endXStack_i &= w_i + s \\
 startXStack_i &= 0 \\
 rowLevel_i &= currentHeight_{i-1} \\
 endYSubrow_i &= rowLevel_i + h_i + s \\
 currentHeight_i &= endYSubrow_i \\
 y_i &= rowLevel_i
 \end{aligned} \tag{4.5.5}$$

Since each row needs a dominant element that defines the row height to emphasize the rows, as seen in Figure 4.22c compared to Figure 4.22d on page 64, I add the constraint in Equation 4.5.6 for each region  $r_i$ :

$$rowLevel_{i-1} \neq rowLevel_i \Rightarrow \bigvee_{j < i} y_j = rowLevel_{i-1} \wedge rowLevel_i = y_j + h_j + s \tag{4.5.6}$$

For each region that is placed in a new row, there exists a previous region in the last row that has the same height as the row. Note that the last row uses Equation 4.5.7 instead, which is true if there exists one region aligned on the row level of the last row, which dominates the height of the last row:

$$\bigvee_{j \leq n} y_j = rowLevel_n \wedge currentHeight_n = y_j + h_j + s \tag{4.5.7}$$

## 4. Model Order in Rectangle Packing

Also note that checking for a dominant element significantly increase the runtime of the maximization problem since it can only be known whether a row has a dominant element after placing all regions in a row. Hence, a constraint solver such as CP optimizer<sup>6</sup> cannot immediately cut off potential search branches without row dominant elements before solving the equations for all regions. A packing with no dominant element in the last row may invalidate a fully optimized packing that otherwise adheres to all placement constraints mentioned above.

Additionally, the actual width  $w_a$  and the actual height  $h_a$  are defined as  $\max(x_i + w_i)$  and  $\max(y_i + w_i)$  for all regions  $r_i$ . To ensure that a deterministic solution and that the approach primarily adheres to the reading direction, the rule priority can be enforced by minimizing  $\sum_i decision_i$  as a secondary criterion. This may also be used for presentation purposes as a tie-breaker between equal packings.

The optimal solution regarding scale measure for the running example can be seen in Figure 4.30. Figure 4.30a shows the result already presented in [DLH+23] without using a row-dominant element. Figure 4.30b and 4.30c illustrate the optimal row-dominant solution without or with whitespace elimination. One can see that the layouts have a slightly worse scale measure, as a result of the row-dominant element constraint.

Moreover, note that Figure 4.30b cannot be created by the heuristic since region F fits below the block A to E. The packing in Figure 4.25b on page 68 will be produced instead. This is a deliberate decision in the rectpacking heuristic since it greedily prioritizes the left-to-right reading direction, while the optimal solution considers the solution that primarily optimizes the scale measure and only secondary considers the reading direction. Specifically, the heuristic does not try fitting region F right of the block A to E since F already fits below block A to E creating a new subrow.

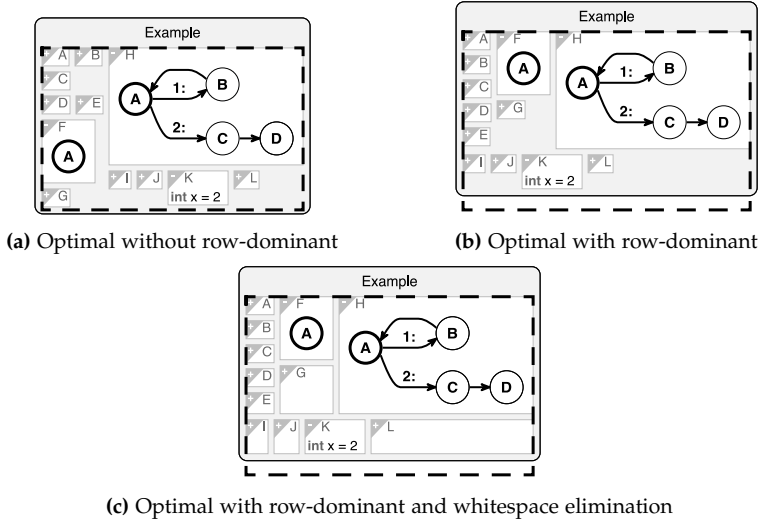
## 4.6 Evaluating the Region Packing Problem

With the optimal solution to the region packing problem, I can evaluate the proposed rectpacking algorithm using the optimal solution as the best case

---

<sup>6</sup><https://www.ibm.com/products/ilog-cplex-optimization-studio/cplex-cp-optimizer>

## 4.6. Evaluating the Region Packing Problem



**Figure 4.30.** Considering row-dominant elements lowers the scale measure. Figure 4.30a presents the optimal solution without considering row-dominant elements, with the desired aspect ratio marked in dashed-black. Figure 4.30b shows the optimal solution with row-dominant elements. Figure 4.30b presents whitespace elimination using the whitespace elimination algorithm without an explicit row structure defined in Section 4.4.6.

and the simple heuristic as the baseline.

I evaluated the following algorithms and configurations based on scale measure and aspect ratio using the GrAna tool [Rie10] with a desired aspect ratio of 1.3 and a region spacing  $s$  of 1 to be used for the SCCharts region packing problem.

- B* The simple heuristic (see Section 4.3), which was included in previous publications [DLH+21; DLH+23], is not included here, since the simple heuristic is clearly worse in cases that are not captured by the RB or RX approach.
- R* The rectpacking heuristic without optimizations. Here, width approximation uses the approach in Section 4.4.1, placement determines the

#### 4. Model Order in Rectangle Packing

row height, and the compaction step uses only one iteration without revising the target width.

- RB* The rectpacking heuristic, which first checks whether regions are stackable, i. e., whether a high region has two preceding or succeeding regions that have an aggregated height smaller than the height of the high region, and uses the simple heuristic if it is not the case. This is included since previous research by Domrös et al. [DLH+23] showed that the simple heuristic outperforms the rectpacking heuristic if regions are not stackable.
- RR* The rectpacking heuristic with row height reevaluation described in Section 4.4.5.
- R2* The rectpacking heuristic with a second compaction run described in Section 4.4.5.
- RX* The rectpacking heuristic that uses the simple algorithm if regions are not stackable and otherwise may revise the row height and do arbitrary many compaction runs. I. e., this is to the best possible packing using both the simple heuristic and the rectpacking heuristic.
- C* The optimal solution regarding scale measure calculated with CP optimizer limited to one hour per problem. The optimal packing adheres to the row-structure and dominant element constraint, as presented in Section 4.5. The final result includes packing problems that exceeded the time limit, which were excluded in [DLH+23].

Since the region packing problem solved by the proposed algorithms is inspired by SCCharts regions, the stimuli should resemble SCCharts region packing problems. To get a sufficient number of non-trivial region packing problems, I generated region packing problems that conform to the SCCharts region packing characteristics using three different problem classes.

The *same height* (SH) class consists of normal regions, which are regions of height 20 and varying width with a mean of 100. These regions correspond to collapsed SCCharts regions (see Section 2.2) for which the height is

## 4.6. Evaluating the Region Packing Problem

uniform, and the region label determines the minimum width. The second *one big node* class mocks a packing with one expanded region that shows its inner content and many collapsed regions. Here, the one big node has a height and width between 300 and 1000. The third *big nodes* (BN) class does not contain one but two to five big regions. This mocks SCCharts models with several interesting regions instead of only one.

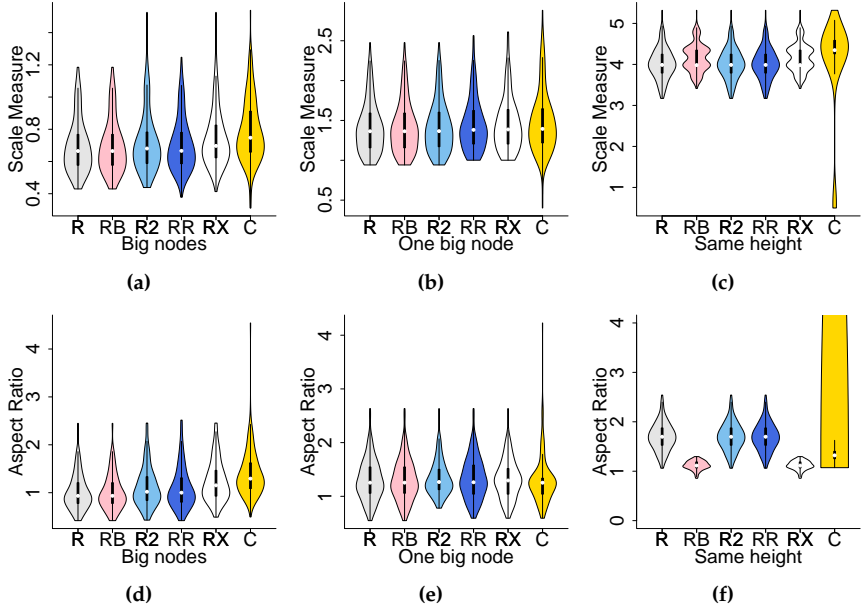
Note that I did not evaluate the algorithms using only big regions since real SCCharts region are rarely stackable, as seen in Table 4.1 on page 84, if there are only expanded regions. Hence, the same height class also covers region packings of only expanded regions.

The number of graphs per class is limited to 200 since the CP optimizer runs, which were limited to one hour per problem, took on average 29 minutes for the same height class, 17 minutes for the one big node class, and 14 minutes for the big nodes class, resulting in more than eight days of constant computation on using two Intel(R) Xeon(R) CPUs E5540 with 4 cores (8 threads) and 2.53 GHz. Hence, I advise to only repeat this evaluation if one thinks that repeating the experiments might result in new insights.

The resulting scale measure, e. g., how much the resulting drawing needs to be scaled to fit a  $\text{DAR} \times 1$  rectangle and the resulting aspect ratio of each packing can be seen in Figure 4.31. As a reminder, a bigger scale measure shows that a drawing fits the desired aspect ratio better than a lower scale measure. Hence, a bigger scale measure means regions can be drawn bigger such that their inner content becomes more readable. The aspect ratio shows whether the desired aspect ratio of 1.3 is over- or underestimated and hence shows potential improvements or disadvantages of certain algorithm configurations. Compared to [DLH+21; DLH+23], the evaluation presented in this work uses new generated packing problems, does not exclude the packing problems that exceeded the runtime, and uses an optimal solution that adheres to the row structure and the dominant element constraint.

I first compare the optimal solution (C) to the best heuristic (RX) that utilizes all possible improvements steps of the rectpacking heuristic. The optimal solution (C) is sometimes worse than the heuristics because of the one-hour execution time limit. In these cases, CP optimizer will always find the solution that puts all regions in one row next to each other or a

#### 4. Model Order in Rectangle Packing



**Figure 4.31.** The different algorithm configurations for each problem class compared by scale measure scaled by 1000 and actual aspect ratio without excluding CP optimizer runs that were terminated after one hour.

solution that is too wide for the desired aspect ratio. This happens since the optimal solution packs regions preferably to the right and tests such a packing first. For more than one big node (BN), the optimal solution (C) was in 6 instances worse than the fully optimized heuristic (RX) because of the one-hour time limit and yielded the same solution 75 times out of the 200 graphs. For one big node (OB), C was worse 5 times and as good as RX 134 times. Finally, for regions of the same height (SH), C was worse 21 times and as good as RX 24 times. Hence, the fully optimized heuristic (RX) can find the optimal solution in nearly half of the packings with at least one big node.

The different improvements of the rectpacking algorithm improve differ-



## 4.6. Evaluating the Region Packing Problem

ent problem classes. For the big nodes class, a second compaction iteration (R2) yields the most improvements, as seen in Figure 4.31a. Both R2 and RX create better results than the other approaches. Trying the simple heuristic first (RB) does not improve the result for BN since the size of the regions has a lot of variation, as detailed in Section 4.3. Hence, R and RB are identical in Figure 4.31a since the simple heuristic cannot improve the packing. Reevaluating the row height (RR) often improves the result but sometimes produces worse results. This may happen if one greedily increases the row height creating a row that may fit the first big node but not the second one following it, as visualized in Figure 4.26 on page 68.

For the one big node class (OB) the best heuristic (RX) is on par with the optimal solution (C) in more than half of the cases. As seen in Figure 4.31b, the heuristic improves by doing the row height reevaluation (RR) and slightly improves further by reevaluating the width during compaction (RX). The row height reevaluation can here not create worse results since the anomaly visualized in Figure 4.26 cannot occur. Trying the simple heuristic first (RB) can again not yield improvements since the one big node disturbs the static width approximation. Only doing a second compaction step to find a better width (R2) does also not improve the basic heuristic (R).

The same height class is better solved by the simple heuristic rather than the rectpacking heuristic [DLH+23], as seen in Figure 4.31c. Hence, RX will always choose the simple heuristic instead of the other improvements.

The actual aspect ratio in Figure 4.31d to 4.31f shows how well the required width is approximated. Specifically the actual aspect ratio shows whether the desired aspect ratio was over- or underestimated. For the big nodes class, the basic heuristic (R) tends to underestimate the required width, which yields a smaller aspect ratio. Hence, a potential way to improve the readability would be to use a higher target width or to use the target width more as a guideline allowing small overlaps. R is improved by R2, RR, and their combination RX. For only one big node, R2 and RR both limit the number of heavily underestimated solutions and produce better scale measures. If all nodes have the same height, the rectpacking heuristic overestimates the required width. The simple heuristic underestimates it instead, as seen at RB and RX. This is the case since the heuristic does only static width approximation based on the regions sizes. It does not consider

## 4. Model Order in Rectangle Packing

**Table 4.1.** Layout problems per depth  $d$  in the used SCCharts models. Here, “#” is the number of instances on each hierarchy level. “mean nodes” and “mean edges” are the mean number of nodes or regions or the mean number of edges for the respective depth. “¬simple?” marks all rectpacking problems that are actually stackable that should be solved by the rectpacking heuristic instead of the simple heuristic. “revise width?” counts problems that could be improved by revising the target width.

$d$	algorithm	#	mean nodes	mean edges	¬simple?	revise width?
1	layered	3130	3.77	5.71		
2	rectpacking	2001	1.5	0	105	32
3	layered	675	3.59	3.5		
4	rectpacking	542	1.36	0	7	0
5	layered	131	9.65	15.45		
6	rectpacking	126	1.08	0	3	0
7	layered	33	9.58	13.39		
8	rectpacking	33	2.27	0	2	0
9	layered	25	2.32	3.4		
10	rectpacking	14	3.42	0	11	11
11	layered	12	1	0		
12	rectpacking	2	6	0	0	0
13	layered	1	3	2		
14	rectpacking	1	4	0	1	0

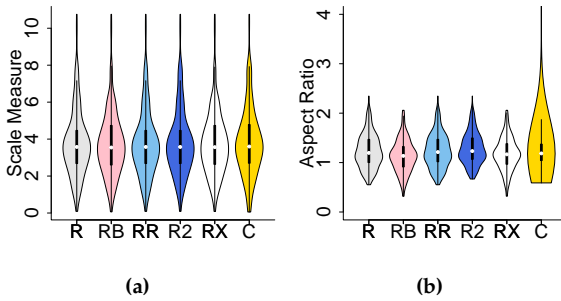
that slightly adjusting the target width may fit more regions into the same row.

### 4.6.1 Region Packing in Hierarchical Graphs

The generated packing problems do not capture the hierarchical nature of SCCharts models and its effect on scale measure and hence readability. Hence, I additionally investigated how each strategy performs using 372 hierarchical SCCharts models with only expanded regions that specify 2719 packing problems. Table 4.1 visualizes the hierarchy levels the region packing problems are in as well as the size of the SCCharts models taken from publications [SHM+18; SSM19], student projects such as the railway project ’14 [SMS+15] and ’17 [SMS+19], and weekly exercise solutions by students.

One can see in Figure 4.32 that locally optimal solutions and the vari-

## 4.6. Evaluating the Region Packing Problem

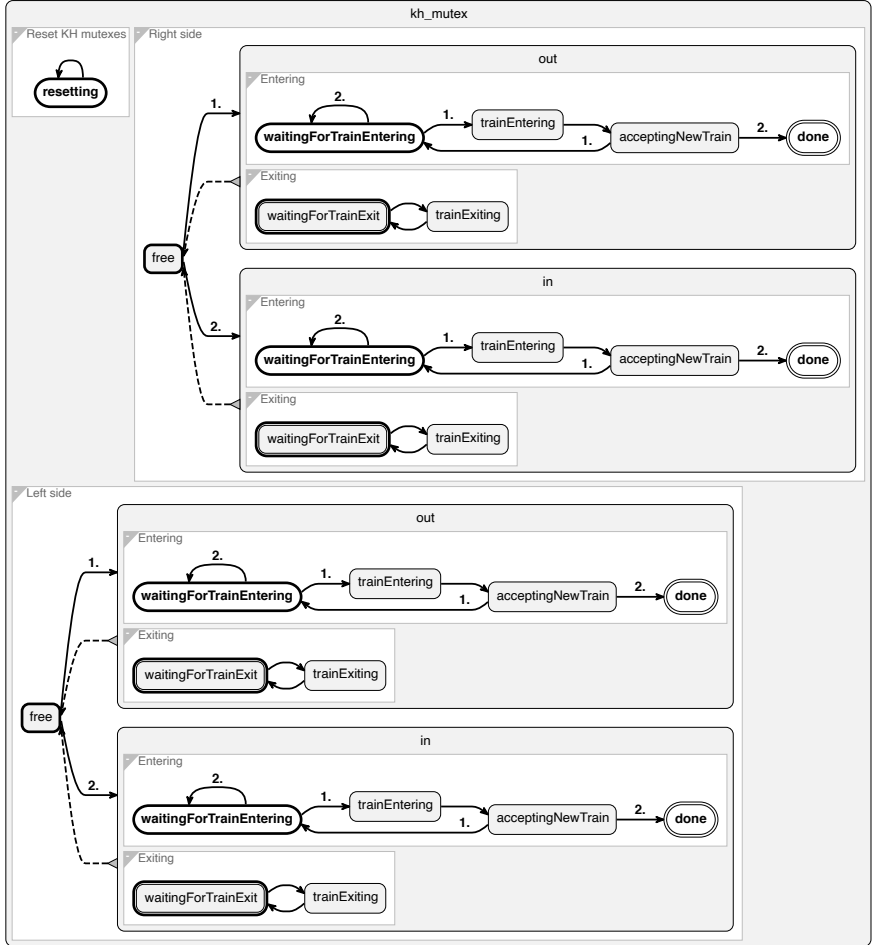


**Figure 4.32.** The scale measure and aspect ratio of 372 SCCharts models created for papers, documentation, student projects and exercises.

ous rectpacking compaction improvements mostly become irrelevant if we consider hierarchical graphs with only expanded regions [DLH+23]. The global scale measure in Figure 4.32a looks nearly the same for all models, even though the optimal solution outperforms the heuristics in the same height (SH) class (see Figure 4.31c) to which SCCharts with only expanded regions corresponds to. This is a case since a locally optimal solution may globally be bad and reduce the overall scale measure and with it the readability. This phenomenon occurs since bottom-up layout cannot determine the correct desired aspect ratio. E. g., since the SCChart in Figure 4.33 is layouted bottom-up, each region packing problem needs a fixed aspect ratio that cannot be approximated based on the top-level elements. Setting this to a fixed value—here 1.3—may result in suboptimal packings. Since the three top-level regions inside `kh_mutex` are packed higher than wide, the innermost packing problem consisting of two regions inside out and in could be drawn far more wide than high by placing `Entering` next to `Existing`. However, since the algorithm layouts bottom-up without backtracking, a globally optimal solution regarding scale measure cannot be found.

Table 4.1 also shows what kind of packing problems SCCharts typically deals with. Many models only have one or two regions per hierarchy level, making the underlying packing problem trivial. Moreover, expanding all regions makes most models solvable by the simple heuristic without wasting space. As a result, the width can only be revised to improve the result of 43 packing problems from the 2719 available problems. Hence, for SCCharts region packing, the rectpacking heuristic with a check for stackability to use

#### 4. Model Order in Rectangle Packing



**Figure 4.33.** The `kh_mutex` SCChart from the Railway Project '14 [SMS+15].

## 4.7. Remaining ONO Region Packings

the simple heuristic if appropriate is advised. Revising the target width or the row height may, however, be useful to control the layout together with the options presented in Section 7.2 if a particular packing is undesirable.

### 4.7 Remaining ONO Region Packings

While the previous section illustrated the effect of the different algorithm configurations on readability, I want to focus on the potential problems that the repacking heuristic (see Section 4.4) may create. More specifically, I want to present all ONO packings, i. e., all packings a user might think that could be trivially improved, that may occur and how they may or may not be preventable using the repacking heuristic.

#### 4.7.1 Greedy Placement Hinders Compaction

The placement step and the resulting approximated target height for each row might have serious impact on the drawing. The placement step sets bounds for what the compaction step can do. E. g., even if one sets the target width high enough to accommodate a big region in the row, compaction might steal regions that we do not want to have in the current row, as seen in Figure 4.34. Here region I is stolen and moved to the first row splitting the block consisting of region I and J in two. This is possible since region I has a smaller width and fits into the first row, while J is slightly wider. As a result, region I looks out of place and users do not understand why J is not stacked on top even though their width looks very similar.

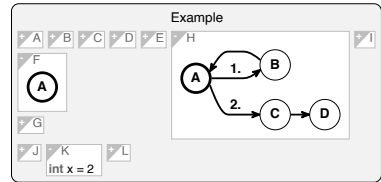
Additionally, whitespace elimination will elongate region I abnormally while the three remaining regions in the last row become abnormally wide. This can only be solved by reworking the placement and compaction steps to be less greedy.

#### 4.7.2 Greedy Block Creation hinders Compaction

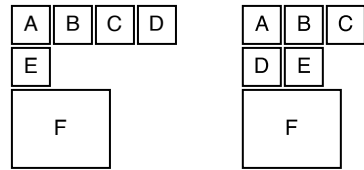
Placement not only approximates the row height, it also begins to group regions with a similar height into blocks, as detailed in Section 4.4.2. Group-

#### 4. Model Order in Rectangle Packing

**Figure 4.34.** The compaction step may steal regions we do not want in our current row if the placement step determines a row height that allows this. Region I is moved into the first row because the regions A to G can be compacted so much that I will always fit the row if H should also be in there.



**Figure 4.35.** Greedy block splitting creates ONO packings, as seen in Figure 4.35a. Figure 4.35b solves this by splitting regions A to E more evenly. In this case, the placement step splits the block and greedily assigns regions A to D to the first row.



(a) Bad block splitting (b) Good block splitting

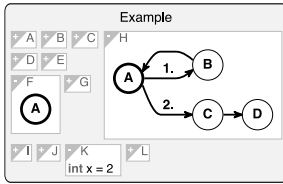
ing regions greedily into blocks based on similar size and the available target width sometimes decreases the scale measure or otherwise visually impair the drawing.

E. g., the packing in Figure 4.35a looks ONO since the algorithm did not evenly distribute the small regions among the first two rows, as it is the case in Figure 4.35b. Instead, the heuristic keeps the similar sized regions together. Even though the scale measure would not improve when moving region D to the second row if we consider a desired aspect ratio of 1.3, moving D would still optimize the used area and visually improve the packing.

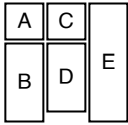
Figure 4.34 illustrates another disadvantage of the block packing policy. Since the block A to E is drawn as slim as possible to fit the block consisting of F on top, which again fits the block consisting of G on top, the drawing is too wide and looks ONO. Splitting block A to E in two subrows and moving G into a separate stack would improve the drawing, as seen in Figure 4.36. Moreover, trying to fit region F right of block A to E would also improve the packing, as seen in Figure 4.30b.

The algorithm should hence consider splitting blocks to create more subrows if this would improve the packing visually by creating a balanced

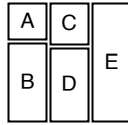
## 4.7. Remaining ONO Region Packings



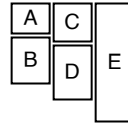
**Figure 4.36.** A different block creation improves the packing. Splitting the similar sized regions A to E more evenly into two blocks is here space efficient. Moreover, assigning the differently sized regions F and G to the same block and placing them in the same subrow is here also better.



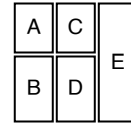
(a) Wrong region alignment before whitespace elimination



(b) Wrong region alignment fixed by whitespace elimination



(c) No region alignment before whitespace elimination



(d) Wrong region alignment after whitespace elimination

**Figure 4.37.** Even though regions are correctly placed in rows, stacks, blocks, and subrows, they may still align with regions and create orderings that do not conform with the model order. In Figure 4.37a two regions align between different stacks, which is solved by whitespace elimination in Figure 4.37b. In Figure 4.37c two regions between different stacks do not align, however, whitespace elimination creates alignment in Figure 4.37d.

packing or if doing so would improve the scale measure or area. Additionally, one could consider a compaction mode with backtracking that tries fitting blocks to the right if fitting it below is possible but seems ONO.

### 4.7.3 Wrong Stack Alignment

Packings of four regions, such as region A to D in Figure 4.20 on page 61, can also occur with a top-to-bottom instead of the primary left-to-right reading direction.

This can occur if two stacks such as the ones in Figure 4.37a are placed next to each other. Since regions A and B and regions C and D have a different height, they may not be placed in the same subrow by the rectpacking heuristic as a result of the greedy block creation. Instead, they are placed in the same stack if the row height allows this. This leads to an unintentional

## 4. Model Order in Rectangle Packing

horizontal alignment of regions A and C and regions B and D. Hence, the packing visually creates a subrow that does not exist and visually compromises the reading direction and hence the model order of the packing. Figure 4.37b solves this problem by eliminating the whitespace.

Whitespace elimination may, however, not always be the solution but can also be the problem. While whitespace elimination may solve the first issue, it may also cause wrong alignment and visually create subrows, as seen in Figure 4.37c and 4.37d. By enlarging all regions equally A and C align as well as B and D visually creating subrows.

None of these two cases can be prevented by the rectpacking heuristic. The compaction or whitespace elimination step can, however, be made aware of this problem by checking whether subrows would align and otherwise offset the next subrow. Doing so would require balancing subrow alignment against scale measure if preventing visual subrows might increase the size of the packing. The optimal solution (see Section 4.5) can consider this by adding the constraint in Equation 4.7.1 for all regions  $r_j$  on the same row level as the current region  $r_i$ .

$$(x_j + w_j + s = x_i \wedge y_j = y_i) \implies (j = i - 1) \quad (4.7.1)$$

This means that only the directly preceding region may be directly left of the current region and align vertically with the current region.

## 4.8 Summary

Considering the evaluation in Section 4.6 and the remaining ONO packings in Section 4.7 illustrates how the rectpacking heuristic handles readability, stability, secondary notation, and control (*EVAL*) while Section 4.4 shows how model order can be integrated into rectangle packing as the reading direction (*IMPL*).

The application for model order in rectangle packing is that model order should constrain the order of rectangles and hence create stability by providing a consistent reading direction. Hence, both the simple heuristic (see Section 4.3) and the rectpacking heuristic (see Section 4.4) produce stable packings compared to general rectangle packing algorithms that do not



consider order. Specifically for the rectpacking heuristic a consistent reading direction is important since stacks of rectangles might create misleading rectangle alignment, as shown in Figure 4.37. Hence, the rectpacking heuristic uses a row and subrow structure (see Section 4.1.3) and prefer creating subrows with a left-to-right reading direction over creating stacks with a top-to-bottom reading direction.

While the simple heuristic (see Section 4.3) is stable, it is not very space efficient and typically produces packings with low readability if the size of the rectangles varies, which is improved by the rectpacking heuristic to make stable rectangle packings viable. Adhering to the rectangle model order and creating a consistent reading direction by following packing guidelines established by comic book panels happens at the cost of readability measured by the scale measure (see Section 4.1.1). Additionally, a row structure is also essential to allow for whitespace elimination to make rectangle packings more aesthetically pleasing, as detailed in Section 4.1.2. The row structure, the dominant element constraint (see Section 4.1.3), and the given ordering of the rectangles all constrain the packing and hence allow for less freedom when using the rectpacking heuristic compared to the optimal solution presented in Section 4.5. Specifically, the target width of the drawing and the target height of each row is typically suboptimal. Hence, rectpacking should be enhanced to revise the target width or row height if no adequate solution regarding scale measure can be found to improve readability, as presented in Section 4.4.5. For hierarchical SCCharts that typically consist of regions of similar size, the rectpacking heuristic and simple heuristic perform similar to the optimal solution formulated as a maximization problem, as detailed in Section 4.6.1. Hence, selecting optimizations for the rectpacking heuristic is not essential and doing so should consider the available time and the desired scale measure.

If one assumes that the reading direction conforms with the intended secondary notation, the rectpacking heuristic creates good secondary notation. Using the proposed algorithm, regions are ordered and similar size regions are grouped, which is typically desired. However, the rectpacking and simple heuristic lack control to express more fine-grained secondary notation.

Model order cannot completely control a region packing, as illustrated

#### 4. Model Order in Rectangle Packing

in Figure 4.20 on page 61. Rectpacking cannot control where rows, stacks, and subrows end by using the one-dimensional model order. Moreover, it cannot set an appropriate target width. Model order can only constrain the order of regions. Hence, diagram interaction may be necessary to add additional layout constraints, as detailed in Section 7.2.

# Model Order in Layered Layout

*“If the first sentence is ‘A graph consists of a set of nodes and a set of edges’, the horse has already left the barn.”*

*“Mit dem ersten Satz ‘Ein Graph ist eine Menge von Knoten und eine Menge von Kanten’ ist das Kind schon in den Brunnen gefallen.”*

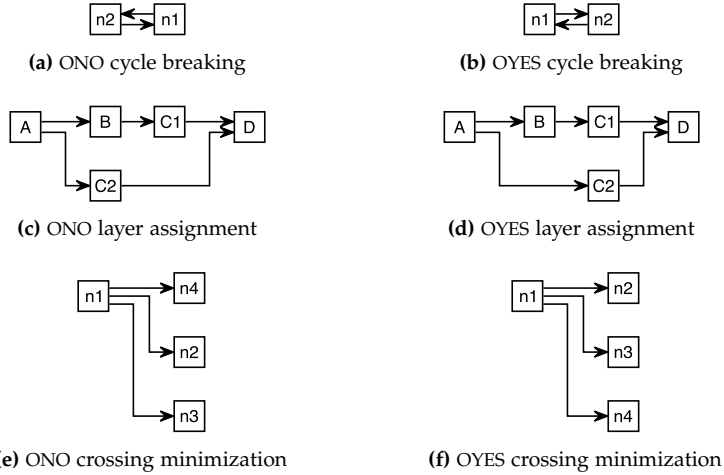
*Reinhard von Hanxleden, RTSYS Christmas Party, 2024,*

While previous chapter looked into model order for rectangle packing, this chapter focuses on model order for the layout of directed graphs using the so called layered or Sugiyama algorithm [STT81].

Nearly every subproblem of the layered algorithm—consisting of the topological phases cycle breaking, layer assignment, and crossing minimization and the geometrical phases node placement and edge routing—is NP-hard when optimizing the corresponding readability criteria. I propose much simpler model order algorithms that, instead of solely optimizing readability criteria, create better secondary notation by utilizing the model order. Additionally, I propose solutions and highlight how the core concepts of these model order algorithms can be integrated into existing heuristics improving the secondary notation without compromising readability.

The idea of this work follows the quote by Jünger and Mutzel [JM04] that “[t]he input data can be expected to determine the choice of such reversals”, which originally refers to the cycle breaking step of the layered algorithm. They argue that the input data—if it exists—is inherently ordered and typically determines the flow of a model. Hence, the model order should be used to make a graph acyclic. The input data, i. e., the model order, is, however, also relevant for the other phases of the layered algorithm, as seen

## 5. Model Order in Layered Layout



**Figure 5.1.** ONO layouts for the cycle breaking, layer assignment, and crossing minimization step that can be solved by using model order. Nodes are numbered by their model order.

in Figure 5.1, with nodes numbered by model order.

Here, Figure 5.1a and Figure 5.1b have the same number of edge reversals, which I call *backward edges*, hence they are equivalent based on this readability criterion. If we, however, assume that the model order expresses intention, Figure 5.1b displays good secondary notation controlled by model order. This makes Figure 5.1a ONO and Figure 5.1b OYES. Similarly, the secondary notation of the layering in Figure 5.1c can be improved by controlling the position of C2 creating desired alignment. Finally, Figure 5.1e and Figure 5.1f are identical despite their node order. If we again assume that the model order expresses intention, Figure 5.1f expresses the correct secondary notation controlled by model order.

Hence, I investigate in this chapter how model order can be utilized inside the layered algorithm (*IMPL*) not only for cycle breaking but for all topological phases, namely cycle breaking (see Section 5.3), layer assignment (see Section 5.4), and crossing minimization (see Section 5.5). Moreover, I discuss how model order might affect the geometrical phases

node placement (see Section 5.6) and edge routing (see Section 5.1.6). Using this information, I deduct how model order interacts with readability, stability, secondary notation, and control in Section 5.9 (*EVAL*), which leads to concrete layout configurations for the example languages SCCharts and Lingua Franca presented later in Chapter 9.

## 5.1 Layered Layout

Following the introductory quote of this chapter, I include order in the graph definition for model order aware layered layout.

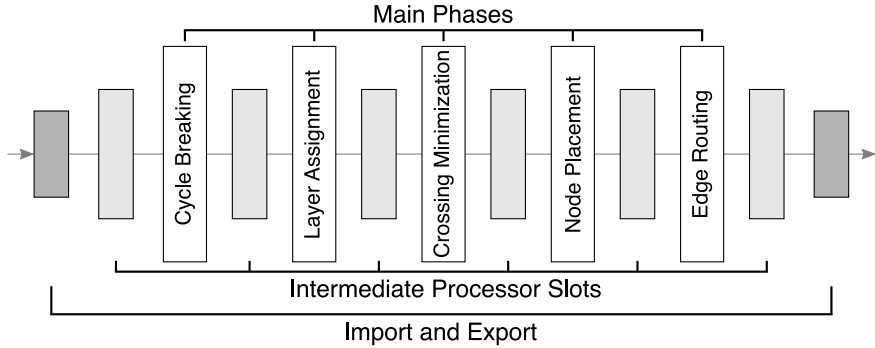
I define a directed graph, which I henceforth call graph,  $G = \langle V, E \rangle$  as the pair of an *ordered* set of  $n$  nodes  $V = \langle v_1, \dots, v_n \rangle$  and an *ordered* set of  $m$  edges  $E = \langle e_1, \dots, e_m \rangle$  with  $e_i = (v_i, v_j)$  and  $v_i, v_j \in V$ . Additionally, I enhance edges by ports, the anchor points of edges on their source and target. The port model order should also be considered during layout since many languages employ such anchor points as part of their model instead of or together with edges. E. g., Lingua Franca (see Section 2.3) models only define ports in form of connected actions, triggers, or timers on reaction instead of directly specifying an edge between reaction and action.

**(Ordered) (Proper) Layered Graph** As a consequence of the graph definition, a (proper) layered graph used during layered layout should also include order.

A *layered graph* is a graph  $G = \langle V, E, L \rangle$  that additionally has a layer assignment or layering  $L$  such that each node  $v \in V$  is assigned to a layer  $L_i$  with  $L(v) = i$ . I call such a layered graph *ordered* if the nodes and edges as well as the layer  $L$  are an ordered set. Such a layered graph is *proper* if edges only exist between neighboring layers, and edges between nodes of the same layer, so called *in-layer edges*, are forbidden. I. e., for all  $e \in E$  with  $e = (v, w)$   $L(v) = L(w) - 1$  holds for all proper layered graphs.

To summarize, I use the standard layered graph definitions with ordered sets instead of unordered sets.

## 5. Model Order in Layered Layout



**Figure 5.2.** The five phases (white) of ELK Layered with intermediate processor slots (light gray) before, between, and after each phase with an enclosing import and export step (gray) [SSH14].

### 5.1.1 Layered Algorithm in the Eclipse Layout Kernel

Since I want to present concrete model order algorithms for layered layouts (*IMPL*), I need to make a more fine-grained division of the different layout phases based on its concrete implementation in ELK.

Traditionally the layered algorithm consists of three phases: layer assignment, crossing minimization, and coordinate assignment. However, the ELK layered algorithm, which I refer to in this work and which holds the implementation for all proposed strategies, is divided into five phases [DHS+23], as seen in Figure 5.2. ELK divides the layouts phases into the topological phases cycle breaking, layer assignment, and crossing minimization, which determine the relative ordering of nodes, edges, and ports, and the geometrical phases node placement and edge routing, which assign concrete coordinates to nodes and ports and routes to edges.

Since I refer to a concrete implementation, part of the problems solved and discussed in the following sections are motivated by the inherent complexity and design decisions of ELK, which is essential to illustrate potential ordering pitfalls when designing layout algorithms.

ELK was implemented over several years by a number of researchers, as depicted in Figure 5.3. This resulted in several strategies for the lay-

## 5.1. Layered Layout

out phases and processors to further divide the problem into manageable chunks, as shown in Figure 5.4. The processor infrastructure visualized in Figure 5.4 is necessary to support a variety of strategies. Additionally, the processor infrastructure allows pre- and post-processing of layout strategies and multiple processors for one single feature. E. g., cycle breaking, which reverses edges, requires the `REVERSE_EDGE_RESTORER` to be executed after routing all edges. The partitioning implementation is divided into a `PARTITION_PREPROCESSOR`, a `PARTITION_MIDPROCESSOR`, and a `PARTITION_POSTPROCESSOR`.

The contracts between these processors do not always consider order since the algorithm designers considered a layered graph to be unordered, which might also be the case for other modular layered layout implementations. In ELK layered, the ordering in a layer is only fixed after crossing minimization has executed, as defined by the original Sugiyama layout algorithm [STT81]. I argue that this should, however, not give all previous processors and strategies the freedom to change the order as they please since this potentially creates the ONO layouts in Figure 5.1. E. g., after layer assignment, the `LONG_EDGE_SPLITTER` breaks up *long edges*, which are edges between non-neighboring layers, into multiple short ones with dummy nodes between them to create a proper layered graph. These new dummy nodes need to be somehow inserted into the existing layers between other real nodes while considering the ordering and its effect on readability, stability, and secondary notation. If this is not the case, the dummy insertion may compromise the model order since the iteration order over a layer changes, which may result in a local minimum in terms of edge crossings (readability) or may reorder edges (stability, secondary notation). Therefore, a processor or phase implementation that wants to adhere to the model order must sort nodes and edges if previous processors or phase implementations might compromise the initial ordering, as further elaborated in Section 5.5.2 for crossing minimization.

Instead of sorting the whole layer, one can also directly insert the new nodes at the correct position, which is far more efficient given the case that the node and edge order are intended, as further elaborated in Section 5.8. After that, crossing minimization will define the ordering inside a layer, which is already acknowledged by the existing strategies. Since such a

## 5. Model Order in Layered Layout

significant change to the algorithm structure is not necessarily desired or possible in terms of the required software engineering effort, the following sections provide independent solutions that assume, if not stated otherwise, that previous processors and strategies did not care about order. However, before doing presenting concrete model order algorithm, I first present the five layered phases together with their readability goals.

### 5.1.2 The Cycle Breaking Problem

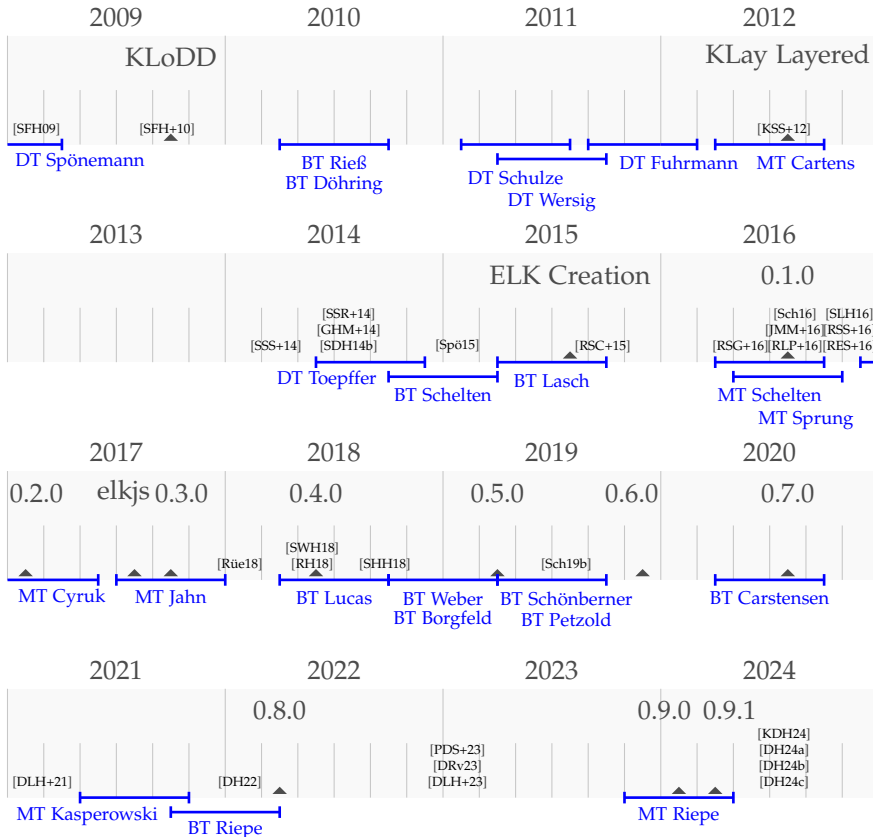
The cycle breaking or cycle removal phase is the first part of the original layer assignment phase by Sugiyama. Its input is a potentially cyclic graph. By reversing edges cycle breaking creates an acyclic graph. The main readability criterion for cycle breaking is the minimal number of backward edges, i. e., edges that have to be reversed for the graph to become acyclic. Minimizing the number of edges to reverse corresponds to the NP-complete [Kar72] minimal feedback set (FS) problem. Hence, one commonly employs heuristics to minimize the number of edges to reverse.

However, for SCCharts and other modeling languages, the number of backward edges is not that interesting. For models that have intention such as the one considered in this work, users are less interested in the number of backward edges but in reversing the *correct* edges, as seen in Figure 1.1 on page 3. Here, both layouts have the same number of backward edges, edge bends, edge length, and edge crossings and have hence the same readability. Only the model order can express that Figure 1.1c is the desired solution since this information is not part of an unordered graph.

Additionally, I argue that one does not necessarily want to change the order of Source and Through if a second edge from Through to Source would be introduced. Figure 1.1c with two backward edges is still the desired solution since Figure 1.1c visualizes the correct flow of the underlying model, which I further elaborate on in Section 5.3.

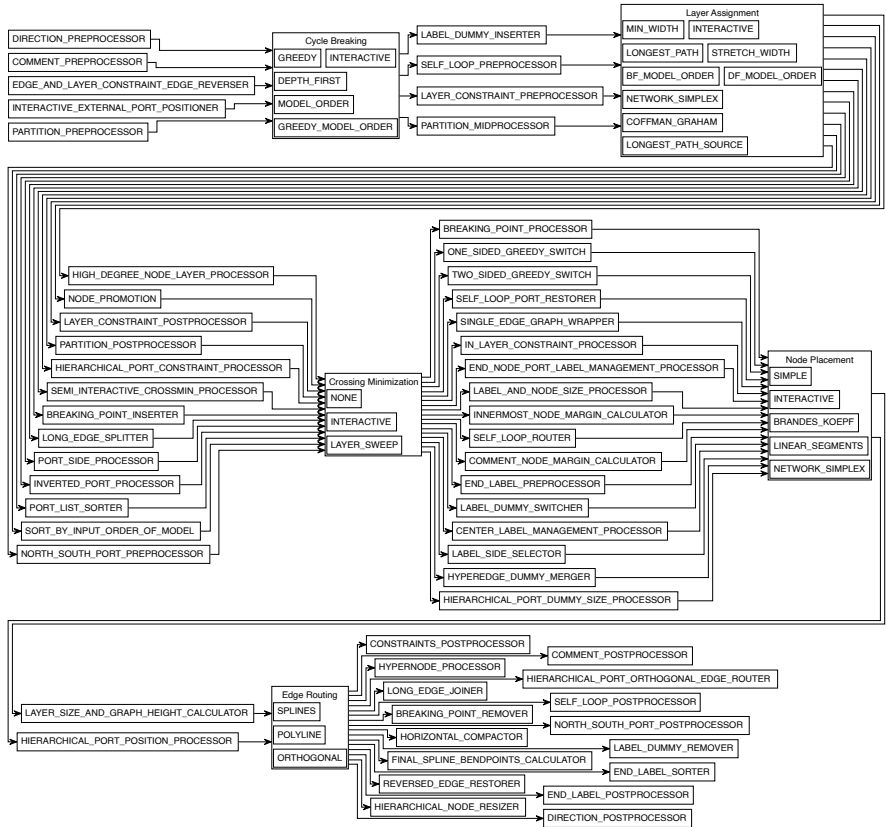


## 5.1. Layered Layout



**Figure 5.3.** The ELK development timeline [DHS+23]. Selected diploma theses (DT), bachelor theses (BT), and master theses (MT) are marked in blue, software releases are marked with a gray triangle, and relevant publications are cited.

## 5. Model Order in Layered Layout



**Figure 5.4.** All ELK Layered processors, ordered by their execution order, and layout strategies, layouted with ELK Layered and styled with KLightD [DHS+23].

### 5.1.3 The Layer Assignment Problem

During layer assignment, one aims to minimize the width and the height of the layering to increase readability. Finding a minimal width and height layering that optimizes the compactness and minimize the edge length of a layout is NP-complete [ES90]. However, a good layer assignment for languages such as SCCharts, Lingua Franca, or System Theoretic Process Analysis (STPA) control structures [PH23] does not necessarily optimize compactness but rather creates a layering that visualizes the desired secondary notation by creating alignment and grouping.

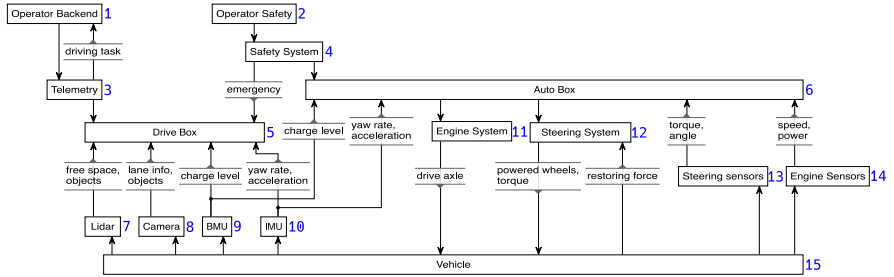
Nodes should be layered such that the layering groups them to create helpful secondary notation, as seen in Figure 5.5. Here, Figure 5.5a optimizes the compactness while Figure 5.5b aligns all actuators and sensors of the displayed control structure of an autonomous vehicle in the same (here horizontal) layer. Here, model order highlights the relation between the different nodes at the cost of readability, which corresponds to compactness in the layer assignment step, as further elaborated in Section 5.4.

### 5.1.4 The Crossing Minimization Problem

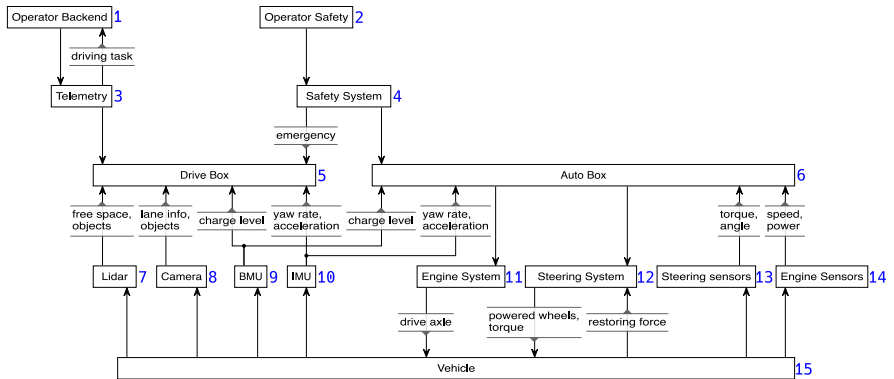
Crossing minimization tries to minimize the edge crossings to improve readability, which is an NP-hard problem even for bipartite graphs [GJ83]. Hence, heuristics such as the barycenter heuristic [STT81] are employed. Since minimizing edge crossings is the most important aesthetic and hence readability criterion [Pur97], crossing minimization algorithms typically aim to minimize edge crossings, hence the name. However, since the crossing minimization phase determines the order of nodes in a layer and the order of ports on a node, this phase is also critical when expressing secondary notation and to achieve stable layouts. Therefore, crossing minimization should minimize the crossings **and** maintain the model order to express secondary notation and increase stability.

Since model order and a crossing minimal solution are often conflicting, I further explore how goals can be considered during the layout and how model order interacts with readability, stability, secondary notation, and control in Section 5.5.

## 5. Model Order in Layered Layout



(a) ONO decision during layer assignment. The number of layers is minimal, however, the relationship between the different states, which is partly expressed by model order, is lost.



(b) OYES decision during layer assignment. The actuators and sensors Lidar, Camera, BMU, IMU, Engine System, Steering System, Steering Sensors, and Engine Sensors are in the same layer.

**Figure 5.5.** STPA control structure [PH23] with downward layout direction of an autonomous vehicle. The node model order is marked in red to the right of each node.

### 5.1.5 The Node Placement Problem

The fourth phase, node placement, assigns concrete y-coordinates<sup>1</sup> to nodes, which potentially aligns nodes or ports based on the topology provided by the first three phases. Hence, node placement minimizes the edge bends, creates alignment, and compacts the drawing to increase readability depending on the concrete strategy used, such as Brandes and Köpf node placement [BK02] or network simplex node placement by Gansner et al. [GKN+93]. During node placement, alignment may express desired secondary notation. However, if the node model order already determines the flow during cycle breaking and the vertical alignment during layer assignment, and the edge and node model order determine vertical order during crossing minimization, the node and edge model order cannot additionally determine horizontal alignment. Hence, model order can not be utilized to a similar extent during node placement. Nevertheless, in Section 5.6 I sketch how model order could still be employed to control the alignment of nodes disregarding how the node model order might be used by preceding phases.

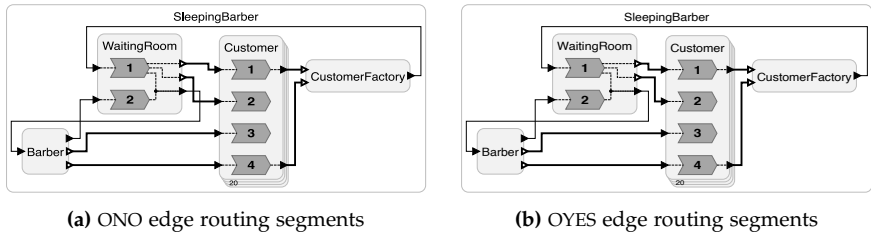
### 5.1.6 The Edge Routing Problem

Edge routing determines the x-coordinates of the nodes and the bend-points for edges. Here, the drawing width can be optimized by minimizing the number of different vertical segment levels to increase readability, as seen in Figure 5.6. Between `WaitingRoom` and `Customer` the horizontal order of the vertical edge segments may increase the drawing width if they are ordered incorrectly or may create unnecessary edge crossings or make the drawing unnecessary wide, as seen in Figure 5.6a, which could be drawn crossing free, as seen in Figure 5.6b. Moreover, different edge routing strategies can create different edge styles such as the orthogonal edge depicted in Figure 5.6 or the splines used for `SCCharts` such as the one in Figure 4.33 on page 86. Hence, the application of model order seems limited. Therefore,

---

<sup>1</sup>If we assume a left-to-right or right-to-left layout direction, node placement assigns y-coordinates. If we assume a top-to-bottom or a bottom-to-top layout direction, node placement assigns x-coordinates. Hence, edge routing assigns the other coordinate and routes the edges.

## 5. Model Order in Layered Layout



**Figure 5.6.** The SleepingBarber Lingua Franca model [MCL+24] visualizes the vertical edge routing segments minimization during edge routing.

I will not further speculate about possible ways to control edge routing by model order.

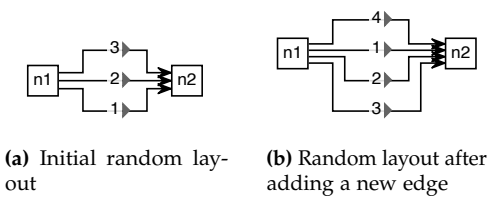
### 5.1.7 Random Decisions in Automatic Layout

After presenting the traditional goals of each layout phase, I want to highlight one of the main reasons for ONO layouts before continuing with related work and the model order for the first four phases of the layered algorithm.

A number of layout algorithms propose randomization to find a good solution. However, random decisions decrease stability and control while only randomly increasing readability and potentially secondary notation, which I illustrate in the following.

In its implementation in ELK preceding 2021, the ELK layered algorithm made random or uncontrollable decisions since the main readability criteria were inconclusive or randomization was used to prevent local minima. E. g., actual random functions were called, processors and strategies uncontrollably reordered nodes and ports, and iterating over Java HashMaps resulted in random iteration orders.

Especially calling a random function is problematic. If automatic layout cannot solve a problem—since there is no clearly optimal solution—a random decision is always worse than a deterministic one since a random decision might be inconsistent. Even if a random decision might increase readability for a specific instance, e. g., by finding a layout with fewer edge crossings, it leaves no room to select between other solutions with the same



**Figure 5.7.** When incorporating random decisions in a layout algorithm, the layout might change based on seemingly unrelated changes to the model.

readability and it is not guaranteed that the same decision will be made for a different subgraph. Moreover, if a random decision creates an ONO layout, one can only try changing the random seed or randomly reorder nodes and edges to try for a different outcome. Hence, a layout becomes uncontrollable if random decisions are employed.

Consequently, I argue that if one has to decide between multiple solutions that seem equally good, it is always better to make a deterministic and controllable decision. E. g., just choosing the first solution is better than flipping a coin and hoping for the best. This is especially important since a random decision might compromise layout stability and with it the mental map, as seen in Figure 5.7, while a deterministic decision can be controlled and can be comprehended by a potential user. E. g., in Figure 5.7, the layered algorithm completely changes the order of the edges and potentially the secondary notation in Figure 5.7a when introducing an additional edge, as seen in Figure 5.7b.

To summarize, since one of the goals of this work is to increase stability and add a way to control the secondary notation of a layout, I aim to eliminate random and uncontrollable decisions in the following algorithms for the layered layout phases.

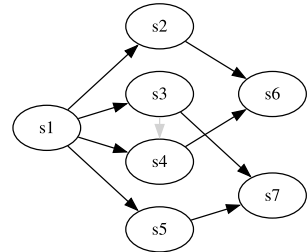
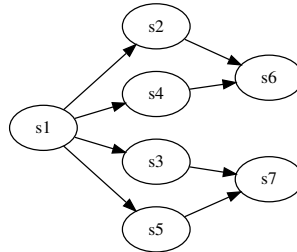
## 5.2 Related Work

Before diving into concrete algorithms, I want to present related work on model order for layered algorithms.

Since layout constraints are often used to control the layout, I shortly discuss their relationship with and model order in Section 5.2.1, which I further present in Section 7.1. Furthermore, I present algorithms that utilize

## 5. Model Order in Layered Layout

```
digraph G {  
  rankdir="LR"  
  s1,s2,s3,s4,s5,s6,s7  
  
  s1 -> s2  
  s1 -> s3  
  s1 -> s4  
  s1 -> s5  
  
  s2 -> s6  
  s3 -> s7  
  s4 -> s6  
  s5 -> s7  
  //{  
  //rank="same"  
  //edge  
  //color="lightgray"  
  //s3 -> s4  
  //}  
}
```



**Figure 5.8.** Dot uses model order per default as a tie-breaker, as seen in Figure 5.8b, but can use layout constraints, as seen in Figure 5.8c.

order in Section 5.2.2 including the library GraphViz and discuss model order for sequence diagrams.

### 5.2.1 Layered Layout Constraints and Model Order

A traditional way to control the layout is by adding layout constraints. These constraints can be either added with a WYSIWYG editor or as annotations to the textual input [PDS+23], which I focus on in this thesis, as seen in Figure 5.8a on the example of GraphViz dot.

In Figure 5.8c, the order of s3 and s4 is constrained by adding an invisible (here light-gray) edge. As a result, nodes in Figure 5.8c are ordered by model order given by the textual ordering in Figure 5.8a making the constraint redundant. This illustrates that model order is, in most cases, an alternative to layout constraints while having the advantage that model order can either



be utilized as a tie-breaker or as a constraint offering different degrees of control.<sup>2</sup>

Moreover, textual layout constraints have the disadvantage that they might become invalid if the model is edited. E. g., if *s3* is renamed or a node between *s3* and *s4* is added, the constraining edge in Figure 5.8c must be reconsidered and potentially updated, as detailed by Petzold [PDS+23] and further discussed in Section 7.1. Using model order such inconsistencies cannot occur.

Moreover, model order is much simpler than layout constraints. Modelers express intention only once by intuitively expressing secondary notation and with it intention in the textual model. The layout remains stable and can be configured to serve as a tie-breaker. E. g., if enforcing the model order creates too many edge crossings, model order crossing minimization can be configured to partly disregard the model order. Additionally, developers do not need to learn how to apply constraints, which requires knowledge about the underlying layout algorithms, but just apply best practices for textual modeling. If model order strategies result in a *ONO* layout, this typically points to a bad ordering in the textual model, as experienced for *SCCharts* in Section 5.9.1 and Section 5.9.5 and reported by *Lingua Franca* experts in Section 5.9.6. Hence, I argue that rather than fixing the symptoms of a bad textual model with constraints, developers should change the textual model to tackle the problem at its root.

Finally, layout constraint can benefit from model order since they typically need a reference layout [PDS+23]. This reference layout should be created with model order to get a stable and controllable reference. Otherwise, e. g., when using random decisions, the layout may become unstable and constraints will not achieve the desired result without constraining the whole graph, as visualized in Figure 7.3 on page 218. Using model order produces a stable initial layout that most likely already does what we want to do without using constraints. Additionally, one has the option to constrain certain parts of the layout to express additional secondary notation, which I further elaborate on in Chapter 7.

---

<sup>2</sup>Although most implementations do not support this, layout constraints can also be optional, as detailed by Graf [Gra92].

## 5. Model Order in Layered Layout

### 5.2.2 Order in Layout Algorithms for Node-Link Diagrams

The concept of utilizing the ordering of a model is not new [JM04]. Especially simple solutions for cycle breaking or engineering centric solutions, such as the work of Zielasko et al. [ZTH+22], typically enforce the model order.

If one has a meaningful ordering, model order should indeed determine the choices. Only if the structure of a graph and its order might be conflicting, or the model order does not match the intention, e. g., if nodes are randomly generated, one needs more elaborate strategies that only use model order as a tie-breaker.

#### GraphViz Dot and Model Order

One layered algorithm that also employs simple model order strategies as a default is the GraphViz dot algorithm [GN00].

Per default, GraphViz respects the model order of nodes, as seen in Figure 5.8. The first occurrence in the textual model determines the node model order, which serves as a tie-breaker, as seen in Figure 5.8b. However, the node model order can only be enforced using constraints, as seen in Figure 5.8c. Additionally, the order of incoming and outgoing edges can be enforced, as seen in Figure 5.9c compared to Figure 5.9b.

When focusing on the flow, i. e., cycle breaking, GraphViz disregards the edge model order entirely and only partially uses the node model order, as seen in Figure 5.10, since it employs a depth-first search beginning with the first node by model order. Hence, GraphViz dot creates the flow using the node model order as the secondary visiting order but requires constraints to select a specific edge to reverse.

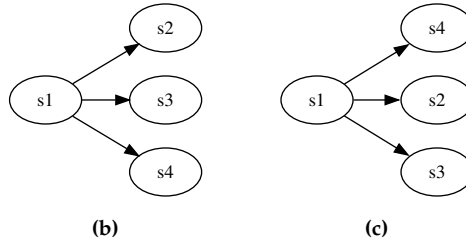
To constrain crossing minimization, GraphViz has the option to enforce the node model order by adding constraints in the form of invisible edges, as shown in Figure 5.8c with the unconstrained graph in Figure 5.8b. The rank option can additionally be used to partition the graph if necessary and to control the flow of a layout, which GraphViz cannot do using the model order alone.

GraphViz dot hence only utilizes the node model order as a tie-breaker during crossing minimization and during cycle breaking as the depth-first visiting order. For anything else, GraphViz utilizes constraints to control the

## 5.2. Related Work

```
digraph G {
  rankdir="LR"
  s1 // [ordering="out"]
  s2,s3,s4

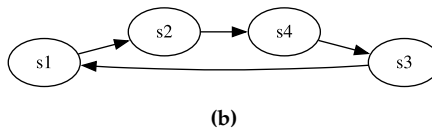
  s1 -> s4
  s1 -> s2
  s1 -> s3
}
```



**Figure 5.9.** Dot uses the node model order for vertical ordering as a tie-breaker. The edge order usually disregarded but with the option to enforce it, as shown in line 3.

```
digraph G {
  rankdir="LR"
  s1,s2,s3,s4

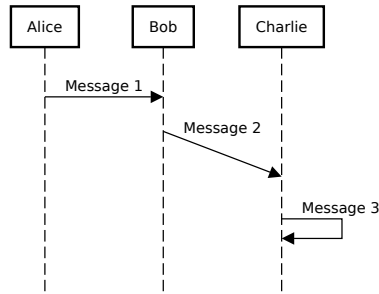
  s2 -> s4
  s1 -> s2
  s3 -> s1
  s4 -> s3
}
```



**Figure 5.10.** Dot uses depth-first search to determine the flow of a graph with the node order as the visiting order.

layout. Even though these are good design decisions for a layered layout algorithm, I propose to go further. In my work, I want to propose model order configurations that extract the intention in the textual model and utilize this information to get the desired layout without layout constraints, which requires options to use the node, edge, or port model order to cater to requirements of several modeling languages.

## 5. Model Order in Layered Layout



**Figure 5.11.** A sequence diagram with life-lines Alice, Bob, and Charlie created with <https://sequencediagram.org/>.

### Model Order for Sequence Diagrams

In contrast to a layered layout, which would typically be employed for flowcharts or state machines, I want to sketch how sequence diagrams typically employ model order.

A sequence diagram consists of life-lines and edges between them, as depicted in Figure 5.11. A textual language that describes a sequence diagrams can employ model order in a very natural way and can control the layout of the whole sequence diagram with it:

“If the sequence diagram is specified through a textual language, this order conforms to the user’s mental map,” Schulze, page 195f [Sch19b].

According to Schulze, the order of life-lines in a sequence graphs should be the same as the life-line declaration in the input model, i. e., the node model order, if life-lines should not be reordered to potentially reduce edge crossings [Sch19b]. Messages between life-lines should be ordered by the edge model order if life-lines have no explicit start and end times.

A straightforward use of the model order is possible since the structure of the sequence diagram allows us to utilize the one-dimension orderings for only one-dimension. As a consequence, textual sequence diagram editors, e. g., <https://sequencediagram.org/>, typically employ model order. Here, the order of nodes determines the order of life-lines and the order of edges the order of messages. For the layered algorithm, however, the order of nodes

or edges may determine the horizontal ordering or the vertical ordering, which I investigate in the following.

### 5.3 Model Order and Cycle Breaking

After presenting related work in form of the layout constraints, GraphViz dot, and sequence diagram layout, I want to present how model order can be used during cycle breaking, the first phase of the layered algorithm.

The goal of cycle breaking should not be to find the minimal number of edges to reverse but to find the correct edges to reverse and hence to determine the correct horizontal order between connected nodes, as explained in Section 5.1.2. I argue that if a modeling language conforms with the editing paradigm in Section 2.1, model order can be used to identify these correct edges.

In the following, I present multiple ways to utilize model order during cycle breaking (*IMPL*). Section 5.3.1 presents how the strict model order cycle breaker enforces model order. Section 5.3.2 and Section 5.3.3 show how model order can be used as a tie-breaker when integrating it into existing cycle breaking algorithms.

#### 5.3.1 Strict Model Order Cycle Breaking

If a textual model already expresses the desired flow with its node model order, the node model order should be used to enforce cycle breaking creating a desirable, stable and controllable layout with good secondary notation, as shown in Figure 1.1.

Hence, I propose the *strict model order cycle breaker* that only utilizes the model order to make decisions that controls the desired flow instead of optimizing backward edges. Additionally, the strict model order cycle breaker is compatible with constraining nodes to the first or last layer, which are common constraints for modeling languages to visualize inputs, outputs, or initial and final states, which is also used for the layout of SCCharts. The strict model order cycle breaker is depicted in Algorithm 4 where

$s(e)$  returns the source node of an edge,

## 5. Model Order in Layered Layout

$t(e)$  returns the target node of an edge,

$o : V \cup E \cup P \rightarrow \mathbb{N}$  is the model order function that returns the model order of an edge, node, or port or  $\perp$  if the element has no model order, and

$c : V \rightarrow \{-1, 0, 1\}$  is the constraint group ordering function defined as follows:

$$c(n) = \begin{cases} -1, & n \text{ shall be in the first layer} \\ 1, & n \text{ shall be in the last layer} \\ 0, & \text{otherwise} \end{cases} \quad (5.3.1)$$

---

### Algorithm 4: The strict model order cycle breaker

---

**Input:** A digraph  $G = \langle V, E \rangle$   
**Output:** An acyclic digraph

```

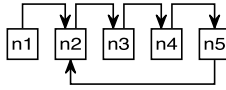
1 for (  $e \in E$  )
2   if (  $c(s(e)) > c(t(e))$ 
3      $\vee (c(s(e)) = c(t(e)) \wedge o(s(e)) > o(t(e)))$  )
4      $E := (E \setminus e) \cup (t(e), s(e))$ 
5 return  $G$ 
```

---

The strict model order cycle breaker assigns each node to a group based on the constraint it might have, which acts as a primary criterion. Sorting the nodes inside each constraint group by model order creates an acyclic graph when reversing all edges going against the model order together with all edges going to smaller constraint groups, as visualized in Figure 5.12. Each edge in Figure 5.12 that goes from left to right is a forward edge. Each edge that goes from right to left is a backward edge. Hence, the runtime of the strict model order cycle breaker is in  $\mathcal{O}(|E|)$  making it the fastest possible cycle breaking algorithm.

The strict model order cycle breaker allows developers to fully control the flow of their model using the node model order to express desired secondary notation and keeping the flow stable. This is possible, the strict model order cycle breaker can deterministically create any kind of edge reversal for unconstrained node orders by sorting the nodes topologically.

### 5.3. Model Order and Cycle Breaking



**Figure 5.12.** Strict model order cycle breaking visualized based on a total linear order, assuming model order corresponds to the node numbering.

Note that it is also possible to use the edge model order to enforce the flow of a graph. I.e., if one identifies all existing cycles, the last edge in each cycle can be reversed by model order. However, for such a *strict edge model order cycle breaker*, recalculation of the cycles after each reversal is necessary to prevent a cyclic solution. This recalculation of the cycles, however, makes a strict edge model order cycle breaker impractical and computationally heavy since finding cycles in requires running Tarjan’s algorithm [Tar72] with runtime  $\mathcal{O}(|V| + |E|)$ . Moreover, I could not find a language that employs an edge ordering that is not based on the node order. Hence, since a strict edge model order cycle breaker seems to be inefficient and does not have a clear use-case, I do not discuss strict edge model order cycle breaking in this work.

#### 5.3.2 Greedy Model Order Cycle Breaking

If the model order should not be enforced during cycle breaking, it can be used as a tie-breaker in existing algorithms, which I illustrate based on the greedy cycle breaker in the following and the depth- and breadth-first cycle breaker in Section 5.3.3.

Using model order as a tie-breaker is a trade-off between control over secondary notation and readability criteria such as the number of backward edges. E. g., in many languages, the node that should be the first node in the diagram may also be the first node in the textual model. However, if multiple model order groups (see Section 5.7 on page 153) exist, as it is the case for Lingua Franca, textual convention rather than intention might determine the node model order. If one nevertheless enforces the model order, the layout may be undesirable, as depicted in Figure 5.13. Here,  $n_4$  is ordered such that structure and model order create a conflict. Blindly

## 5. Model Order in Layered Layout



**Figure 5.13.** The greedy cycle breaker and the model order cycle breaker.

enforcing the model order during cycle breaking may therefore create the layout in Figure 5.13b instead of the desired layout in Figure 5.13a.

Such a “wrong” model order typically falls into four categories. First, novices may have created the model employing bad textual secondary notation [Pet95]. If this is the case, I argue that the bad textual model should be corrected rather than choosing a less controlling layout algorithm to mask the problem. Second, the language itself may restrict the model order, e. g., since the order represents priority or scheduling order, as it the case for Lingua Franca. Here, one would like a less constraining cycle breaker to make up the restrictions imposed by the modeling languages. Third, the model may be very big or very connected such that it was not properly maintained resulting in a bad textual model. In this case, I again argue that one should rather correct the bad textual model such that it remains maintainable in the future rather than choosing a less controlling strategy. I even argue that a well maintained textual model is even more important for big than for small models. Fourth, there might be other factors such as semantically different elements ordered by convention, as presented in Section 5.7.1. In such a case, model order should be used as a secondary criterion or as a tie-breaker, which I illustrate based on the greedy cycle breaker, the depth-first cycle breaker, and the breadth-first cycle breaker.

Algorithm 5 shows the *greedy cycle breaker*, which was inspired by the work of Eades et al. [ELS93] with the following additional functions.

*indegree and outdegree* calculate the number of incoming and outgoing edges.

*removeFirst* removes the first element of an ordered set.

*updateNeighbors* removes all occurrences of a node from a graph including



### 5.3. Model Order and Cycle Breaking

edges from or to the node.

*findMaxOutflow* determines the node with the maximum degree.

*chooseNode* randomly chooses a node from an ordered set.

*chooseModelOrderNode* chooses the node with the minimal model order from a given set.

The goal of the greedy cycle breaking heuristic is to minimize the number of edge crossings. It iterates over all nodes and removes all sinks and sources until only nodes that are part of cycles remain. If the graph is acyclic, no nodes will remain to be processed and the algorithm terminates. If the graph is cyclic all edges leading to the node with the highest *degree*, the difference of outdegree and indegree, are reversed. If there are multiple nodes with the same degree, *chooseNode* in line 19 may randomly choose a node to reverse.

Figure 5.14 visualizes the resulting problem that the random decision and usage of the degree poses. Here, the greedy cycle breaker removes all sources and sinks from Figure 5.14a, as visualized in Figure 5.14b. The remaining graph only consists of two nodes *n2* and *n3* with an edge from *n2* to *n3* and an edge from *n3* to *n2*. Hence, the greedy algorithm that optimizes the number of backward edges makes a random decision to reverse any of these edges.

The usage of random decisions in automatic layout is, however, flawed, as explained in Section 5.1.7. For the same two nodes, the decision might be consistent with a fixed random seed but when adding a third or fourth node, the random decision might randomly and unpredictably change, threatening layout stability and the mental map. I further argue that the decision which edge to reverse should not only be deterministic but also controllable, which is the case if we utilize model order as a tie-breaker.

Instead of utilizing random decisions by using *chooseNode*, I propose the *greedy model order cycle breaker* that utilizes the model order by calling *chooseModelOrderNode*. By selecting the node with the minimal model order instead of a random one, the greedy model order cycle breaker deterministically and controllably creates Figure 5.14c instead of Figure 5.14a.

## 5. Model Order in Layered Layout

---

### Algorithm 5: Greedy (Model Order) Cycle Breaker

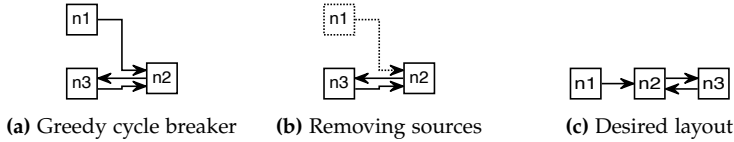
---

**Input:** A digraph  $G = \langle V, E \rangle$   
**Output:** An acyclic digraph

```

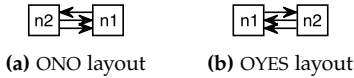
1   $G' := G, \text{sources} := \emptyset, \text{sinks} := \emptyset$ 
2   $\text{unprocessed} := |V|$ 
3  while (  $\text{unprocessed} > 0$  )
4      for (  $n \in V$  )
5          if (  $\text{indegree}(n) = 0$  )
6               $\text{sources} \cup = n$ 
7          if (  $\text{outdegree}(n) = 0$  )
8               $\text{sinks} \cup = n$ 
9      while (  $|\text{sinks}| > 0$  )
10          $\text{sink} := \text{removeFirst}(\text{sinks})$ 
11          $\text{updateNeighbors}(G', \text{sink})$ 
12          $\text{unprocessed} := \text{unprocessed} - 1$ 
13     while (  $|\text{sources}| > 0$  )
14          $\text{source} := \text{removeFirst}(\text{sources})$ 
15          $\text{updateNeighbors}(G', \text{source})$ 
16          $\text{unprocessed} := \text{unprocessed} - 1$ 
17     if (  $\text{unprocessed} > 0$  )
18          $\text{maxNodes} := \text{findMaxOutflow}(G')$ 
19          $n = \text{choose}[\text{ModelOrder}]\text{Node}(\text{maxNodes})$ 
20         // Reverse incoming edges
21         for (  $e = (v, n)$  )
22              $E := (E \setminus e) \cup (n, v)$ 
23          $\text{unprocessed} := \text{unprocessed} - 1$ 
24 return  $G$ 
```

---



**Figure 5.14.** The greedy cycle breaking algorithm randomly makes wrong decisions. Figure 5.14a illustrates the result of the greedy cycle breaker. Figure 5.14b illustrates how the greedy cycle breaker removes all sources and sinks. Figure 5.14c illustrates the desired layout that utilizes model order.

### 5.3. Model Order and Cycle Breaking



**Figure 5.15.** The greedy model order cycle breaking cannot determine the first node.

This simple change makes the algorithm deterministic and allows a certain level of control.

However, the greedy model order cycle breaker may still have trouble to identify the correct first node in a graph if multiple backward edges are desired, as seen in Figure 5.15. Since the greedy model order cycle breaker still primarily aims to reduce backward edges, it creates Figure 5.15a instead of the desired solution in Figure 5.15b.

#### 5.3.3 Depth-First and Breadth-First Model Order Cycle Breaking

While the greedy cycle breaker made it fairly obvious that a random decision might be a bad idea, I want to illustrate in the following how model order might already affect your layout algorithm implicitly on the example of the depth-first and breadth-first cycle breaker.

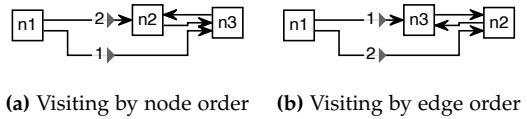
The depth-first cycle breaking heuristic [GKN+93] works by executing a depth-first search through the graph and reversing all edges to already visited nodes.

The depth-first cycle breaker does not employ random decisions model order could be substituted for. However, model order is here typically used implicitly as the visiting or iteration order of nodes. Hence, it is able to find the first node if the model order is used as the iteration order, solving the problem depicted in Figure 5.15. However, since a model might have a different node and edge model order, I propose a depth-first node model order and a depth-first edge model order cycle breaker. These cycle breakers both control the sometimes brittle depth-first search visiting order and make the implicit iteration order explicit.

The *depth-first node model order cycle breaker* uses the node model order as a tie-breaker, as depicted in Figure 5.16a. Here, the depth-first search begins with node n1 and continues with node n2 leading to a backward edge from n3 to n2. The *depth-first edge model order cycle breaker* also starts with the first

## 5. Model Order in Layered Layout

**Figure 5.16.** Depending on the visiting order, a depth-first or breadth-first cycle breaker may create Figure 5.16a or Figure 5.16b.



node by model order but uses the edge model order as the visiting order, which results in Figure 5.16b. Here, the depth-first search visits n3 after n2 since the first edge of n1 leads to n3.

Similarly to the depth-first cycle breaker, the breadth-first cycle breaker works by executing a breadth-first search through the graph and reversing all edges to already visited nodes. Again the node or the edge model order should here determine the visiting order creating the *breadth-first node model order cycle breaker* and the *breadth-first edge model order cycle breaker*.

Consequently, all layout algorithms should use model order when iterating over the graph since the visiting or iteration order often matters for the final result. Thus, pre-processing steps shall not reorder nodes and edges and data structures and algorithms shall be order preserving. E. g., a Java HashMap or a quicksort algorithm may have a seemingly random iteration order or may reorder equal elements by chance.

Adhering to the model order makes the layout more stable since algorithms possibly make fewer random decision and provide users with control over the layout, which random decisions might compromise. Additionally, if the resulting decision by model order is undesired, the input model can be adjusted to control the layout in a straightforward manner without introducing and understanding the possibly available layout constraints. For the layered algorithm, this requires preserving the model order before crossing minimization determines the final order of nodes and edges in a layer. Any strategy or processor before crossing minimization<sup>3</sup> cannot just change the order of nodes and edges in the source model, since the iteration order matters during all topological phases of the layered algorithm.

<sup>3</sup>Take a look at Figure 5.4 again to see potential steps that should happen before crossing minimization.

### 5.3.4 Summarizing Model Order Cycle Breaking

I presented the efficient strict model order cycle breaker, the less efficient strict edge model order cycle breaker, and the tie-breaking greedy, depth-first, and breadth-first node or edge model order cycle breakers.

Typically, the strict model order cycle breaker should be used to find the correct backward edges to reverse since it directly uses the input model to determine the choice of the edge reversals. The strict model order cycle breaker works best for state machines or languages with nodes that can be freely ordered and form branches of control flow that can be followed, which I will evaluate in Section 5.9. If model order should only prevent ONO edge reversals, such as Figure 5.14a, existing layout algorithm can be enhanced by using model order instead of random decisions or by using model order as the iteration or visiting order. This allows users to have more control over their layout and prevents the most common ONO layouts as a result of an uncontrollable edge reversal.

## 5.4 Model Order in Layer Assignment

While cycle breaking determines horizontal order between connected nodes, layer assignment determines vertical alignment of nodes. Since this vertical alignment is unrelated to edges, model order layerers primarily use the node model order instead of the edge model order to control the layout.

The following sections answer the question of how model order integrates into layer assignment (*IMPL*). Additionally, I discuss how layer assignment may exert control using model order and how it relates to the mental map and secondary notation together with a discussion on readability criteria. To do this, I will first discuss model order as a tie-breaker in Section 5.4.1 before presenting strict model order strategies in Sections 5.4.2, 5.4.3, and 5.4.4.

### 5.4.1 Tie-Breaking Model Order during Layer Assignment

Figure 5.17 illustrates how the node model order can be used as a tie-breaker, which aims to leave readability criteria, i. e., the average edge length and

## 5. Model Order in Layered Layout



**Figure 5.17.** Layer assignment as a tie-breaker.

the size of the layout, unchanged.

Assuming a desired breadth-first node order, the node C2 can be moved from the second layer below B (see Figure 5.17a) to the third layer below C1 (see Figure 5.17b) without changing the number of layers or the average edge length. This group all nodes of the C type in the same layer showing desired secondary notation. Hence, I argue that the drawing is improved by following the model order.

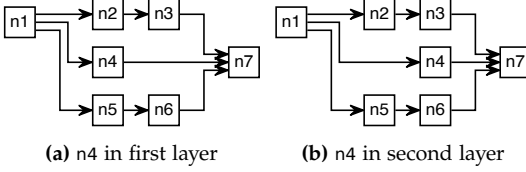
However, in Figure 5.18 one cannot find a layer for  $n_4$  based on model order since  $n_4$  would fit between  $n_2$  and  $n_5$  and also between  $n_3$  and  $n_6$ . Here, model order cannot decide whether Figure 5.18b or Figure 5.18a is the desired layout. Each potential layer for  $n_4$  has nodes with a higher or a lower model order. This may also happen when using the strict model order cycle breaker.

A tie-breaking model order layerer therefore requires a desired breadth-first node order, which requires strict model order cycle breaking, to be used effectively. If there is no breadth-first order, model order does not help in making a decision and does not increase control or stability. Hence, a tie-breaking model order layerer that keeps the readability criteria stable can only exert the necessary control to influence the layout in the trivial case shown in Figure 5.17. To exert control and increase stability during layer assignment, a strict model order layerer is necessary. However, such a strict model order layerer will potentially decrease readability by increasing the edge length and with it the size of the layout, as detailed in the following.

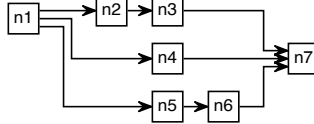
### 5.4.2 Breadth-First Model Order Layer Assignment

Since a tie-breaking strategy for model order layer assignment is often not able to exert the necessary control, as seen in Figure 5.18, I propose

## 5.4. Model Order in Layer Assignment



**Figure 5.18.** Model order layer assignment requires a consistent order. If the nodes have no breadth-first order, no decision based on model order can be made.



**Figure 5.19.** Enforced model order layer assignment.

a (strict) *breadth-first model order layerer*, which increases control over the layout by potentially increasing the required area and edge length, as seen in Figure 5.19 compared to Figure 5.18.

In Figure 5.19, aligning  $n3$ ,  $n4$ , and  $n5$  emphasizes the desired breadth-first node order. A more compact layout could be created by moving  $n5$ ,  $n6$ , and  $n7$  one layer to the left, as seen in Figure 5.18b, while reducing area and edge length at the cost of secondary notation and stability. E. g., in Figure 5.18a, a layerer that optimizes edge length would move  $n4$  one layer to the right if a second edge from  $n4$  to  $n7$  would be introduced, resulting in seemingly unrelated changes.

The strict breadth-first model order layerer, hence, creates a layering that complies with the *model order layering constraints* such that for all nodes, there exists no node with a lower model order in a succeeding layer and no node with a higher model order in a preceding layer:

$$\forall v, w \in V : (L(v) < L(w) \rightarrow o(v) < o(w)) \wedge L(w) < L(v) \rightarrow o(w) < o(v)$$

Hence, the (strict) breadth-first model order layerer in Algorithm 6 assumes a strict model order cycle breaker (see Section 5.3.1) to be used as a pre-condition such that the edge direction does not conflict with the model order. Algorithm 6 further assumes that iterating through nodes is done by model order and that the first node is in the first layer, which is also the initial current layer to work as follows.

The algorithm places the next node in the current layer per default. If

## 5. Model Order in Layered Layout

the next node has a connection to the current layer, it cannot be placed in the current layer and moves to the next layer, making the new layer the new current layer. This continues until all nodes are placed and results in an  $\mathcal{O}(|V| + |E|)$  layering algorithm.

Note that the layering step excludes adding dummy nodes to break long edges. The algorithm assumes that post-processing by the LONG\_EDGE\_SPLITTER adds the dummy nodes, as depicted in Figure 5.4.

---

### Algorithm 6: Breadth-first model order layering

---

**Input:** An acyclic graph  $G = \langle V, E \rangle$  with model order of  $v$  greater than model order of  $w$  for all edges  $(w, v) \in E$ .

**Output:** A layered ordered graph with model order of  $v$  smaller than model order of  $w$  if  $L(v) < L(w)$  for all  $v, w \in V$ .

```

1  $i := 0$ 
2  $\text{currentLayer} := L_i$ 
3 for ( $v \in V$ )
4    $\text{connectedToCurrentLayer} := \text{false}$ 
5   for ( $e \in E \mid (w, v)$ )
6     if ( $L(w) = i$ )
7        $\text{connectedToCurrentLayer} := \text{true}$ 
8   if ( $\text{connectedToCurrentLayer}$ )
9      $i := i + 1$ 
10     $\text{currentLayer} := \text{newLayer}(i)$ 
11   $\text{setLayer}(v, \text{currentLayer})$ 

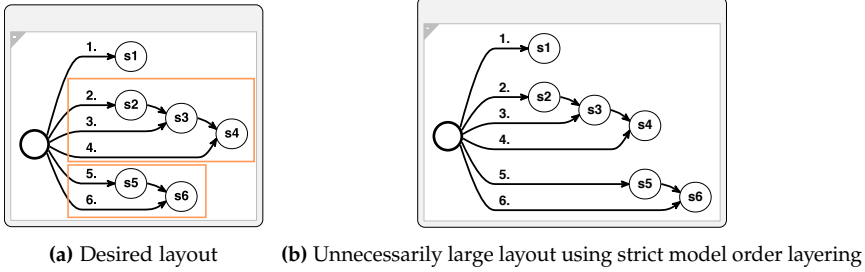
```

---

Even though a strict model order layerer can be easily implemented, as seen above, it might not be useful for all modeling languages. For state machines such as the obfuscated SCChart in Figure 5.20a, enforcing model order results in the drawing in Figure 5.20b, which might be undesired. It is a common pattern in SCCharts that states are ordered by control-flow branches marked in orange in Figure 5.20a. Here, the model order cycle breaking and the resulting backward edges already determine the desired layering. Figure 5.20b makes the drawing unnecessarily wide and only adds value if aligning  $s_4$  and  $s_5$  is good secondary notation, which is not the case for this obfuscated SCCharts model. Hence, a strict model order layer assignment algorithm might also be undesirable for other languages that model control-flow.



## 5.4. Model Order in Layer Assignment



**Figure 5.20.** Strict model order layering potentially affects the width of a layout. Figure 5.20a and 5.20b illustrate how strict model order layering affects the layout of an example state machine.

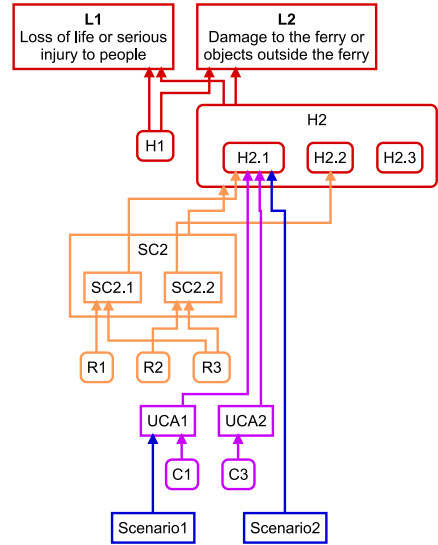
An enforcing breadth-first model order layerer might, however, be desirable for STPA control structures such as the one depicted in Figure 5.5 on page 102 or the STPA relationship diagram [PKH23] in Figure 5.21. Here, the strict breadth-first model order layerer controls the layout to align actuators and sensors.

E.g., the breadth-first model order layerer created the layout in Figure 5.5b on page 102. Here, the breadth-first model order layerer places Operator Backend in the first layer. Since Operator Safety is not connected to Operator Backend, the breadth-first model order layerer also places Operator Safety in the first layer. Telemetry, however, is connected to Operator Backend and does therefore open a new layer. This continues until all nodes are layered.

The STPA relationship diagram [PKH23] depicted in Figure 5.21 similarly utilizes the breadth-first model order layerer. Here, developers desire a breadth-first order of the nodes and the semantically different elements already partition the graph. Red losses are on top of red hazards connected to green system constraints and responsibilities. Blue unsafe control actions and controller constraints with yellow scenarios at the bottom follow the responsibilities. Hence, developers naturally order these different elements in the desired breadth-first order and also want this order to occur in the drawing.

## 5. Model Order in Layered Layout

**Figure 5.21.** The CAPTN STPA relationship diagram [PKH23] partitions the different semantic elements and can utilize breadth-first model order layer assignment for this. The layout direction is top to bottom. The losses are on top, below are the hazards, system constraints, responsibilities, unsafe control actions, controller constraints, and scenarios in breadth-first order.



### 5.4.3 Breadth-First Model Order Layer Assignment by Post-Processing

If a graph only partially adheres to a breadth-first node model order, the solution described above cannot be applied, as illustrated in Figure 5.18. Here, post-processing the layering might be the better alternative. A simple algorithm for layer assignment post-processing after inserting dummy nodes can be seen in Algorithm 7.

Here `promoteNode` moves a node to the next layer while also promoting connected nodes that would otherwise create in-layer edges.

The algorithm works as follows. One promotes a real node if

1. it has a model order (line 4-5),
2. it is not the only node in the first layer (line 7-8),
3. it has the highest model order in its layer (line 9-11), and
4. the next layer either has at least one real node with a lower model order (line 16-18) or

## 5.4. Model Order in Layer Assignment

---

### Algorithm 7: Breadth-first model order layerer by post-processing [DRv23]

---

**Input:** A layered graph  $G = \langle V, E \rangle$

**Output:** A layered ordered graph

```

1 somethingChanged := false
2 do
3   for (  $n \mid \text{!dummy}(n) \wedge \text{outdegree}(n) > 0$  )
4     if (  $\text{o}(n) = \perp$  )
5       continue
6     currentLayer :=  $L_{L(n)}$ 
7     if (  $|\text{currentLayer}| = 1 \wedge L(n) = 0$  )
8       continue
9     for (  $v \in \text{currentLayer}$  )
10      if (  $\text{o}(n) < \text{o}(v)$  )
11        continue
12    nextLayer :=  $L_{L(n)+1}$ 
13    allowsPromotion := false
14    dummyLayer := true
15    for (  $v \in \text{nextLayer}$  )
16      if (  $\text{!dummy}(v)$  )
17        allowsPromotion |=  $\text{o}(v) < \text{o}(n)$ 
18        dummyLayer := false
19      else if (  $\text{!allowsPromotion} \wedge \text{dummyLayer}$  )
20        if (  $(n, v) \in E \wedge L(\text{longEdgeTarget}(n)) - L(n) \leq 2$  )
21          dummyLayer := false
22    if (  $\text{allowsPromotion} \vee \text{dummyLayer}$  )
23      promoteNode( $G, n$ )
24      somethingChanged := true
25 while somethingChanged;
26 return  $G$ 
```

---

## 5. Model Order in Layered Layout

5. the next layer contains only dummy nodes while the current node has enough space to its connected real node such that it can safely be moved in the next layer (line 19-21).

Constraint (1) is necessary since there might be real nodes without a model order that are created by the *diagram synthesis*, which translates the model into a graph layout problem, as it is the case for the startup and shutdown triggers (see Section 2.3) in Lingua Franca [HLF+22]. Constraint (2) ensures that the source node does not promote the whole graph over and over. Constraint (3) ensures that only the node with the highest model order can be promoted. Constraints (4) and (5) describe the two different criteria for the layer the current node might move to. Either a real node with a lower model order is in the next layer, as seen in Figure 5.20a for s5 and s3, or the next layer has only dummy or dummy label nodes, as seen in Figure 5.5a, which shows an STPA control structure that is used to analyze hazardous scenarios. Here Engine System can be moved since Vehicle, its *long edge target*, which is the real node it is connected to, is far enough away to move the node and the label dummy without also moving Vehicle. Note that Safety System could be in the layer directly after Operator Safety, but Safety System aligns itself with Telemetry based on the model order to create proper alignment.

The runtime of this post-processing algorithm is no longer linear. In the worst case each of the  $i$  iterations over all  $|V|$  real nodes moves only one node and that happens at most  $|V|/2 = i$  times. Hence, we get a worst-case runtime of  $\mathcal{O}(|V|^2)$ .

However, the post-processing algorithm can be applied to a wider range of problems since the graph must only partially adhere to a breadth-first node model order. Moreover, the increased runtime should not be a problem for most visual languages since hierarchy is typically used to keep the number of nodes in one hierarchy level below 20, as seen in graph problem distribution for SCCharts in Table 4.1.

### 5.4.4 Depth-First Model Order Layer Assignment

The previous sections always assumed that the node model order is given in a breadth-first manner. Since this cannot be applied to languages such as

## 5.4. Model Order in Layer Assignment

SCCharts and potentially other state machine dialects, I also want to sketch how a *depth-first model order layerer* would work, even though I have a real world application for it.

The depth-first model order layerer—orthogonally to the breadth-first layerer—puts nodes in the next layer based on model order and uses depth-first search instead of breadth-first search to visit nodes.

---

**Algorithm 8:** Depth-first model order layerer

---

```
Input: An acyclic graph  $G = \langle V, E \rangle$ 
Output: A layered ordered graph
1 firstNode := true
2 components =  $\emptyset$ 
3 currentComponent =  $\perp$ 
4 for ( $v \in V$ )
5   if (firstNode)
6     currentComponent = new Component( $v$ )
7     components.add(currentComponent)
8   else if ( $v.incomingEdges \neq \emptyset$ )
9     inCurrent = False
10    for ( $w \in \{v.incomingEdges.source\}$ )
11      w.component.add( $v$ )
12      // Merge components if possible.
13      if (inComponent)
14        mergeComponents( $v, components$ )
15      // Mark unconnected nodes.
16      inCurrent = (w.component == currentComponent)
17      if ( $\neg inCurrent$ )
18         $v.unconnected = true$ 
19  else
20    // No incoming edges.
21    components.add(new Component( $v$ ))
22     $v.unconnected = true$ 
23  firstNode = false
24 return  $G$ 
```

---

Algorithm 8 illustrates the depth-first layering algorithm, which handles nodes by model order: The depth-first model order layerer puts the first node in the first layer. If a node is connected to already placed nodes, the

## 5. Model Order in Layered Layout

depth-first model order layerer puts the node in the next layer. If a node has no incoming edges, the algorithm creates a *new layer component*, marks the new node, and puts the new node in the first layer of that new layer component.

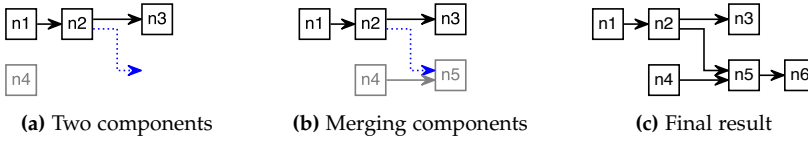
This new layer component might later be connected to the already layered components, merging the two layerings. Subsequent nodes that are only connected to the new unlayered component but not to layered nodes are marked accordingly.

If a new node that is connected to a marked node in a separate layer component and a placed node occurs, the two components merge. The marked nodes will be unmarked and placed while considering the offset from the connected marked node and the placed node to potentially shift placed or marked nodes.

Additionally, one should here maintain the model order when merging layers. E. g., in Figure 5.22a,  $n_1$  is placed in the first layer of the first layer component. Since  $n_2$  is connected to  $n_1$ ,  $n_2$  is placed in the second layer and  $n_3$  is placed in the third layer. Since  $n_4$  is not connected to the already layered nodes, it is put in the first layer of a new layer component. Figure 5.22b shows how the  $n_5$  node connects the two existing layer components. Normally,  $n_5$  would be placed in the second layer of the second layer component. However, since  $n_5$  connects to the already placed node  $n_2$  that is also in the second layer, the second component is moved with an offset of one. Finally,  $n_6$  is placed in the fourth layer resulting in Figure 5.22c.

While this algorithm creates a valid layering it has a substantial flaw. The problem with this algorithm is that it leaves no control option for model order other than the iteration order. Instead, the structure of the graph determines when a new “row” or “column”, i. e., a new layer, is created. Only the iteration order of nodes matters and the assumption of strict model order cycle breaking makes the search for layered components easier since there can only be edges leading to new nodes and no edges from new nodes to already layered ones. Therefore, an algorithm such as network simplex [GKN+93] or a simple longest path layering based on the source(s) may efficiently solve this task if the iteration order corresponds to the model order. Hence, I will not further investigate depth-first model order layering.

## 5.4. Model Order in Layer Assignment



**Figure 5.22.** Depth-first model order layering on an example model. The first layered component is marked in black and the second in gray and a potential connection of the first component to another one is marked in dotted blue.

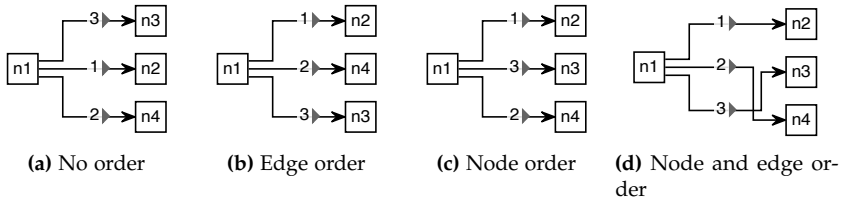
### 5.4.5 Summarizing Model Order Layer Assignment

To summarize, model order layer assignment does not have as many use-cases as its cycle breaking counterpart. It can only be done effectively if a breadth-first node model order exists. Real world use-cases are STPA control structures and relationship diagrams [PKH23] or other visual languages that value control of vertical alignment. These modeling languages are good candidates since developers have full control over the textual node order, no ordering conventions that conflict with model order exist, and the desired result is graph partitioned by node types. State machines, however, should not use model order layer assignment, since the nodes, i. e., the states, are typically not ordered breadth-first.

A tie-breaking model order layerer can be easily integrated in existing layering algorithms by considering a post-processing step to move nodes to a free layer. Again, this requires a strict breadth-first node order, making this approach hardly usable by most languages, since there often is no way to control the layout to express secondary notation. If a breadth-first node ordering is present, a strict model order layerer provides the necessary control and stability at the cost of the readability criteria area and edge length. This is again undesired by languages such as SCCharts.

Moreover, a depth-first model order layering approach cannot control the layout by model order. This approach only makes standard layering approaches easier by assuming a cycle breaking by model order but does otherwise only utilize the given connections to make layering decisions and uses model order only for the iteration order.

## 5. Model Order in Layered Layout



**Figure 5.23.** Without considering model order (see Figure 5.23a), any kind of crossing minimal ordering is optimal. Following the edge order (see Figure 5.23b) or node order (see Figure 5.23c) order may lead to different orderings. Following both may induce edge crossings (see Figure 5.23d).

## 5.5 Model Order in Crossing Minimization

To recapitulate, cycle breaking uses the node model order to decide on the direction of edges or indirectly uses both the edge and the node model order as a tie-breaker. Layer assignment uses the node model order to create vertical alignment. Crossing minimization instead determines the vertical order of elements.

Traditionally, the crossing minimization phase minimizes edge crossings to increase the readability of a graph layout [Pur97]. Using model order, I want to optimize the desired vertical order of nodes and the desired order of edges and ports instead. In addition to edge crossings, I focus on stability and secondary notation preventing that algorithms only consider readability, i. e., edge crossings, which is problematic, as illustrated in Figure 5.23.

Figure 5.23a is optimal regarding readability measured by edge crossings. However, it neither respects the node model order nor the edge model order. I argue that this is a sign that the algorithm that creates such a drawing leaves users no control over the layout, which is problematic. E. g., a user that wants to try out automatic layout might be unsatisfied once he creates a small example and is presented with the unordered solution in Figure 5.23a without any option to change this. Additionally, the layout suggests that adding an outgoing connection to n1 might again scramble the nodes and edges and potentially threaten stability. Figure 5.23b instead shows how the edge order can be configured to have precedence over the node order, while



## 5.5. Model Order in Crossing Minimization

Figure 5.23c illustrates a preferred node order. Here, anchor points of an edge, the ports, determine the ordering of the edges. Hence, edge and port model order will here control the order of ports, which implicitly defines the order of edges. Enforcing both edge and node order at the same time may induce crossings, as seen in Figure 5.23d. Hence, one should define which is more important for a given language.

In the following, I present how model order can be configured for crossing minimization (*IMPL*). I show how model order can be used as a tie-breaker in a pre-processing step and during crossing minimization in Sections 5.5.1, 5.5.2, 5.5.3, and 5.5.4. Moreover, I show how model order can be completely enforced to fully control the layout during crossing minimization in Section 5.5.5.

### 5.5.1 Leveraging Model Order for Crossing Minimization

As an introduction to model order crossing minimization, I want to present the different ways to integrated model order to prevent Figure 5.23a.

The reason algorithms may create Figure 5.23a and not Figure 5.23b or Figure 5.23c is that an algorithm might not consider the initial ordered crossing minimal solution, which I want to focus on in the following. Other reasons may be that an algorithm actually uses a set-semantic as part of a graph definition using unordered sets for nodes and edges resulting in a random order, or that the actual ordering of the model is random to begin with.

To understand why crossing minimization might choose the bad ordering in Figure 5.23a, I first present a crossing minimization algorithm in Algorithm 9. Crossing minimization typically works by sweeping through the graph to order nodes and ports [SSH14] based on barycenter or median values, as depicted in Figure 5.24. Moreover, crossing minimization uses multiple runs with different starting configurations. The first layout run begins with some the initial configuration depicted in Figure 5.24. The first layout sweep begins from the first layer and reorders the following layer based on the connections to the previous one. E. g., in Figure 5.24 the blue barycenter values 2.5 and 3.33 are calculated using the average weight (magenta) of their connected ports. E. g.,  $\frac{2+3}{2} = 2.5$  and  $\frac{1+4+5}{3} = 3.33$ . Once

## 5. Model Order in Layered Layout

---

### Algorithm 9: Crossing minimization without model order

---

**Input:** A proper layered graph  $G = \langle V, E, L \rangle$   
**Output:** A proper layered ordered graph

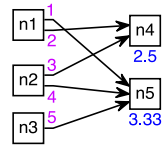
```

1  $r = \text{randomSeed}$  // A fixed random seed
2  $t = 7$  // Thoroughness
3  $\text{sweepForward} = \text{sweepDirection}(r)$ 
4  $\text{bestOrder} = \text{null}$ 
5 for  $i = 0; i < t; i = i + 1$  do
6    $G = \text{randomizeLayers}(G, r, \text{sweepForward})$ 
7   do
8     foreach  $L_i \in L$  do
9        $\text{minimizeCrossings}(L_i)$ 
10    while  $\text{improved}(G);$ 
11    if  $\text{crossings}(G) < \text{crossings}(\text{bestOrder})$  then
12       $\text{bestOrder} = G$ 
13     $\text{sweepForward} = -\text{sweepForward}$ 
14 return  $\text{bestOrder}$ 

```

---

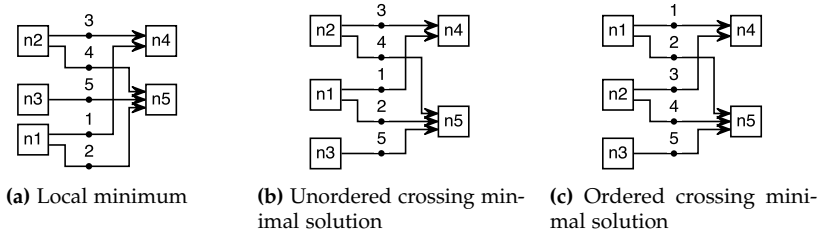
**Figure 5.24.** Barycenter heuristic visualized. The port order in the initial layer is marked in magenta, while the calculated node barycenter is marked in blue below the node.



the barycenter layer sweep reaches the last layer, the process terminates if nothing improved, as it is the case in Figure 5.24. Otherwise, the layer sweep continues in the backward direction. The first layout run requires using model order since the initial sweeping direction, the initial order in the first layer, and the order of dangling source nodes—nodes with only outgoing edges—determine the outcome of the layer sweep run. A crossing minimal solution is found by saving a crossing minimal node and port permutation after each run (see line 12) and comparing its crossings with the previous best solution. Hence, the first layout run has priority. This approach, however, has three issues.

First, the crossing minimization algorithms typically use randomization to prevent local minima, as it is, e. g., the case for ELK. If the first layer and its ports have a fixed order, there is a fixed outcome of the layer sweep, which

## 5.5. Model Order in Crossing Minimization

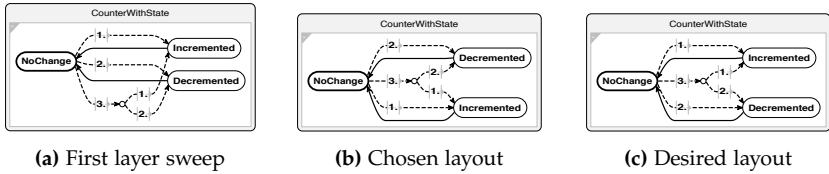


**Figure 5.25.** Dummy nodes are marked as a dot with edge order labels on top of the dummy node. Finding a local minimum during crossing minimization depends on the initial node and port order. The order in Figure 5.25a creates a local minimum with two edge crossings. Both Figure 5.25b and 5.25a are crossing minimal. However, Figure 5.25c adheres to the model order.

may occasionally lead to local minima, as depicted in Figure 5.25. Here, the barycenter heuristic [STT81] cannot find the crossing minimal solution in Figure 5.25a, while the permutation of  $n1$ ,  $n2$ , and  $n3$  in Figure 5.25b and 5.25c have minimal edge crossings. Permuting the initial layer for each layer sweep run with the number of runs bounded by the thoroughness, which is here limited to seven (see line 2), may prevent local minima by trying different permutations to start with. Hence, randomization is here essential to find the crossing minimal solution. I, however, argue that at least the first try should be constrained by model order and not be randomized. If nothing constrains the initial layer order, the result may be random and potentially undesired, as seen in Figure 5.23a. Here, the output of the first run is crossing minimal. Hence, other crossing minimal solutions such as Figure 5.23b found in subsequent layer sweep runs do not overrule Figure 5.23a even though Figure 5.23b might be the desired solution. The main issue is here that the outcome of crossing minimization cannot be controlled since the algorithm uses random decisions. E. g., before model order affected the layout in ELK, SCCharts developers reordered nodes and edges and adjusted the random seed in the hope to create a desired layout to prevent ONO layouts such as Figure 5.23a from occurring in papers or presentations.

Second, even if the initial layer order would not be randomized, one has

## 5. Model Order in Layered Layout



**Figure 5.26.** If the first layer sweep in Figure 5.26a is not crossing minimal, an unordered permutation such as the one in Figure 5.26b may be chosen instead of the desired solution in Figure 5.26c.

to make sure that the initial layer order reflects the model order, which is often not the case. Any layout phase or processor before crossing minimization (see Figure 5.4) might change the order of nodes and edges. E. g., layer assignment does typically not care about the insertion order of nodes to a layer. Instead, layer assignment may insert nodes based on the visiting order or iteration order it has, which might not adhere to the desired model order. Moreover, inserting dummy nodes to route long edges typically adds them at the end of a layer, which typically creates wrong initial edge routes. Hence, nodes and ports have to be sorted before crossing minimization starts to take model order into account to be compatible with other phase implementations.

Third, currently only the edge crossings determine whether a solution is “the best.” E. g., if an initial configuration yields a solution that still has edges crossings, a second crossing minimal layout run might improve it, and a third crossing minimal but ordered solution can no longer be selected, as shown in Figure 5.26. Here, Figure 5.26a is the initial solution that is not crossing minimal. Without using model order as a tie-breaker between layouts with equal edge crossing, the unordered solution in Figure 5.26b may be chosen over the desired solution in Figure 5.26c. Hence, I propose to also take order into account in addition to edge crossings when deciding whether a layout is “better,” as detailed in Section 5.5.4. This also has the advantage that secondary notation that typically corresponds to the model order can be measured and used as an metric to improve the layout.

In the following, I present three different ways that illustrate how one can integrate model order into the crossing minimization phase to solve the

## 5.5. Model Order in Crossing Minimization

issues in Algorithm 9 by small adjustments shown in Algorithm 10. First, I present how the initial ordering that might be compromised by previous phases can be preserved [DH22; DRv23]. Here, it is essential that one starts with a forward sweep (see line 3), sorts ports and nodes (see line 5 and line 7), and prevents unnecessary random decisions, as the one in line 11. Second, I present on the example of the barycenter heuristic, how the partial order given by the node model order can be integrated into a crossing minimization heuristic. Third, I illustrate how model order can be used as a selection criterion and a metric for the best solution, as seen in line 16.

---

**Algorithm 10:** Crossing minimization with model order

---

```
Input: A proper layered graph  $G = \langle V, E, L \rangle$ 
Output: A proper layered ordered graph
1  $r = \text{randomSeed}$  // A fixed random seed
2  $t = 7$  // Thoroughness
3  $\text{sweepForward} = \text{true}$ 
4 foreach  $v \in V$  do
5    $\text{sortPorts}(v)$ 
6 foreach  $L_i \in L$  do
7    $\text{sortNodes}(L_i)$ 
8  $\text{bestOrder} = G$ 
9 for  $i = 0; i < t; i = i + 1$  do
10   if  $i > 1$  then
11      $G = \text{randomizeLayers}(G, r, \text{sweepForward})$ 
12   do
13     foreach  $L_i \in L$  do
14        $\text{minimizeCrossings}(L_i)$ 
15     while  $\text{improved}(G)$ ;
16     if  $\text{crossings}(G) + \text{order}(G) < \text{crossings}(\text{bestOrder}) + \text{order}(\text{bestOrder})$  then
17        $\text{bestOrder} = G$ 
18      $\text{sweepForward} = \neg \text{sweepForward}$ 
19 return  $\text{bestOrder}$ 
```

---

### 5.5.2 Model Order Crossing Minimization by Pre-Sorting

As a first improvement, I propose that crossing minimization should have a controllable and deterministic initial layer order before starting layer sweeps

## 5. Model Order in Layered Layout

by doing *model order pre-sorting* as one way to add control by model order (EVAL).

This solves the issue that all previous phases and processors (see Figure 5.4) might already compromise the initial order since it is not formally part of their contract. Moreover, it makes sure that dummy nodes are also correctly ordered, which ELK otherwise adds to the end of a layer via a post-processing step. Hence, this is also the case for the model order layer assignment algorithms shown in Figure 5.4.<sup>4</sup> Moreover, even if dummy nodes would be sorted according to model order, one needs to make sure that their placement does not introduce ONO layouts. Hence, nodes and ports need to be sorted before crossing minimization can start, beginning with the first layer and the first node in it.

Sorting nodes and ports requires an ordering to adhere to. The model order provides three different potentially conflicting orderings to consider: the node model order, the edge model order, and the port model order. Note that for simplicity, I consider the port and the edge model order to be the same. Both order ports in a specific way and only their origin differs. Hence, I will for the sake of clarity only use the edge model order instead of the port model order and highlight when they might be used differently.

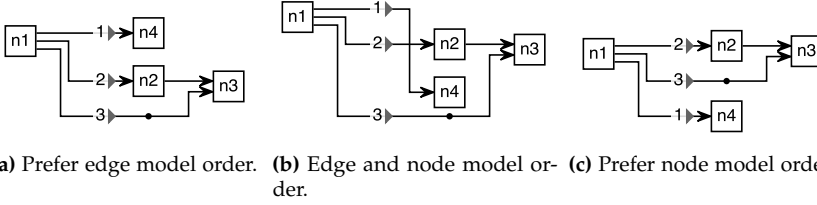
Using the node model order and the edge model order, I propose three different pre-ordering strategies, as depicted in Figure 5.27. Either edges determine the order of nodes and ports, and the node order is only considered if no connection to a previous layer exists, as seen in Figure 5.27a, nodes determine the order of real nodes and edges determine the order of ports and dummy nodes, which results in crossings if both orderings conflict, as seen in Figure 5.27b, or the order of nodes determines the order of nodes and ports, as seen in Figure 5.27c.

All three pre-ordering strategies iterate over all layers beginning with the first one and reorder all nodes and their ports one after the other. To sort nodes, one can either use their model order or their connections to an already sorted previous layer to transitively find sensible orderings. Consequently, either edge or node model order determines the order of outgoing ports, which—in this phase of the layered algorithm—are all edges

---

<sup>4</sup>However, I present an algorithm that only uses model order in Section 5.8 that considers dummy nodes.

## 5.5. Model Order in Crossing Minimization



**Figure 5.27.** The three crossing minimization pre-sorting strategies: prefer-edges, nodes-and-edges, and prefer-nodes.

going from left-to-right since backward edges are reversed. Model order pre-sorting orders incoming ports by their connection to the already sorted previous layer.

Note that I do not determine the order of incoming ports by model order. This is the case, since ELK provides mechanisms to constrain the order of ports if desired. Moreover, constraining both the order of incoming and outgoing ports by model order might introduce preventable edge crossings that will be solved by the first layer sweep and override the order of incoming ports. Hence, no tie-breaking effect exists and there are easier ways to constrain the order by just not reordering ports. Therefore, model order pre-sorting orders incoming ports by their connections to prevent ONO layouts.

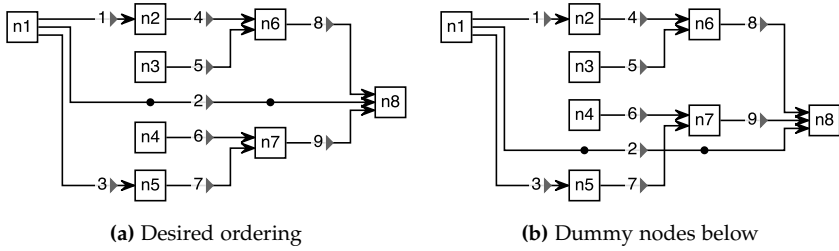
In [DH22], I presented the required ordering as a formal relation. In this work, however, I want to show that the ordering is quite simple and instead show the ordering as an enumeration of conditions for each of the three pre-sorting strategies *prefer-edges*, *nodes-and-edges*, and *prefer-nodes*.

### Preferring Edge Order for Node Ordering

When preferring edge model order, model order pre-sorting orders nodes as follows.

*Transitive order* If the transitive closure of the already compared nodes already determines the ordering of the two nodes, it should be respected.

## 5. Model Order in Layered Layout



**Figure 5.28.** Illustration of “No connection”: Dangling nodes, e. g.,  $n_3$  and  $n_4$ , are not comparable to dummy nodes since dummy nodes have no model order and dangling nodes no connection to the previous layer.

*No connection* If at least one of the two nodes has no connection to the previous layer, they have to be compared by node model order. If one of them has no node model order, they have to be statically ordered.

*Connection* If both nodes directly connect to the previous layer, we order them by the order of their connections to the previous layer.

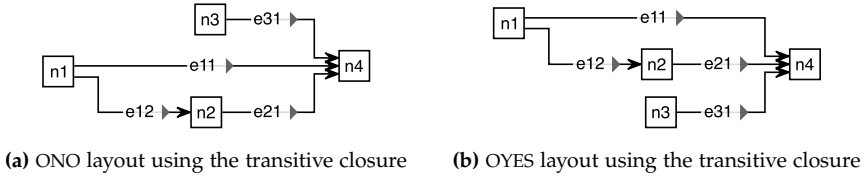
The *Connection* case is the most intuitive. It means that the order of nodes should be determined by the order of ports in the previous layer, which will already be ordered by model order. The other two cases are more interesting. They deal with nodes with no connection to the previous layer.

An example for the *No connection* case can be seen in Figure 5.28. Here, the real dangling nodes  $n_3$  and  $n_4$  cannot be compared to the dummy node in their layer. This is neither possible by their connection to the previous layer, which  $n_3$  and  $n_4$  do not have, nor by model order, which the dummy node does not have. To still get an order, a static decision has to be made to order dummy nodes either above or below dangling nodes, which may yield edge crossings in the initial solution, as seen in Figure 5.28b. Hence, the desired result in Figure 5.28a will not be created by pre-sorting alone. Therefore, not all desired layouts can be expressed by model order pre-sorting alone if a layered graph has dangling nodes and dummy nodes in the same layer.

Finally, a consistent total order is created by considering the *Transitive or-*



## 5.5. Model Order in Crossing Minimization



**Figure 5.29.** Illustration of “transitive order”: Statically sorting dummy nodes below dangling nodes creates a node model order violation given the wrong order of comparisons or a non-order when ignoring the transitive closure.

*der*, which always has priority, to prevent conflicts and undesired orderings when using the *No connection* case. E. g., if the partial order relation given by the nodes and edges are in conflict. Hence, the order of comparisons to create the transitive closure matters. The following combination of comparisons in Figure 5.29 might create a transitive closure that violates node model order if the comparisons using the two partial ordering relations are randomly applied. If the dummy node *e11* is first compared to *n3*, *n3* is statically ordered above it since both are otherwise incomparable. Choosing real nodes to be ordered below dummy nodes is here preferred since it typically sorts long edges at the bottom of the drawing. If the second comparison compares *e11* to *n2*, their connection to the previous layer orders *n2* below the dummy node, resulting in Figure 5.29a, which violates the node order. Given these two comparisons, the transitive closure needs to order *n3* over *n2*. However, when directly comparing *n2* and *n3*, which has to be done by model order, *n2* is above *n3* resulting in a potential conflict when ignoring the transitive closure or in a node model order violation by the transitive closure, a “non-order.” This can be prevented by fixing the order in which the comparisons that determine the transitive closure take place. First, comparing all real nodes ordered by model order before considering dummy nodes prevents the problem. E. g., if *n2* and *n3* are compared first, the transitive closure creates the desired result in Figure 5.29b.

The open question remains whether there may be other means to compare dummy nodes and dangling nodes such that static decisions are not necessary. E. g., maybe it might be possible to predict whether a certain or-

## 5. Model Order in Layered Layout

der might produce edge crossings or not or one could skip the comparison and let the transitive closure decide.

### **Considering Edge and Node Order or Preferring Node Order for Node Ordering**

Another option for node ordering is to primarily use the node model order instead of the connections to the previous layer, which is the case for the nodes-and-edges and prefer-nodes pre-sorting strategy.

*Transitive order* If the transitive closure of the already compared nodes already determines the ordering of the two nodes, it should be respected.

*Model order* If both nodes have a model order, they are real nodes and their node model order determines their order.

*No model order and no connection* Similar to the *No connection* case above, two nodes of which one is a dummy node and one has no connection to the previous layer can only be statically compared to each other.

*No model order and connection* If one of the nodes has no model order but both have a connection to the previous layer, their connections determine the ordering.

The prefer-nodes and nodes-and-edges strategies have the same problems with the transitive closure and dangling nodes as the prefer-edges strategy. These strategies differ from the prefer-edges strategy in the handling of nodes that have a model order and since they directly compare nodes with a model order with one another. Moreover, since this strategy utilizes both the node and the edge model order, a layout may have additional crossings, as seen in Figure 5.27b.

### **Preferring Edge Order or Considering Nodes and Edges Order for Port Ordering**

After ordering nodes, model order pre-sorting determines the order of ports. Here, it is important to not introduce unnecessary edge crossings by strictly

## 5.5. Model Order in Crossing Minimization

constraining all ports by edge or port model order, which is solved by order inheritance.

*Transitive order* If the transitive closure of the already compared ports on a node already determines the ordering of the two ports, it should be respected.

*Order Inheritance* A port should use the minimal model order of all ports that connect to the same node to bundle ports.

*Outgoing ports* If both ports are outgoing, i.e., connect to a node in a succeeding layer, the two ports can be ordered by edge or port model order.

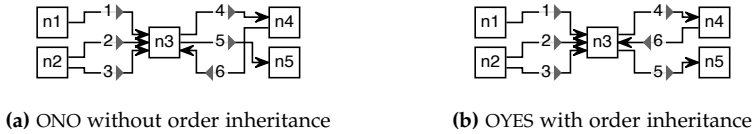
*Incoming ports* Similar to the *No connection* case for nodes, the connection to the previous layer orders two incoming ports.

*Outgoing and incoming port* Both ports can be statically ordered such that outgoing ports are before incoming ports or the other way around, since they will be on different sides of the node.

When sorting ports, I consider incoming ports to be on the left or `WEST` side of the node and outgoing on the right or `EAST` side, as depicted in Figure 5.30b. Note that the backward edge from `n4` to `n3` is also an outgoing edge since it is reversed during cycle breaking. In Figure 5.30b, `n1` and `n5` need no port sorting since they have only one port. Node `n2` can order its outgoing ports by port or edge model order such that edge `e21` is above `e22` and node `n4` orders its incoming edges by their connection to the previous layer. For `n3`, we sort the three incoming ports and the three outgoing ports independently. The three incoming ports get the ordering from the previous layer to prevent unnecessary node local edge crossings by bad port permutations.

Moreover, I sort the outgoing ports by model order and order inheritance. This means edge `e41` which has a higher model order than `e32` is nevertheless sorted above `e32` since it inherits the model order of `e31`, which has the same target as `e41`. This prevents the `ONO`-ordering depicted in Figure 5.30a. Here, order inheritance solves a potential edge crossing

## 5. Model Order in Layered Layout



**Figure 5.30.** An example for model order port pre-ordering to prevent ONO layouts.

already by pre-sorting. This is important, since a crossing in the initial order might result in crossing minimization reordering nodes and edges in an undesired way. Hence, this prevents that crossing minimization completely reorders the graph if there is a “better,” i. e., crossing minimal, solution. Hence, I here disregard the edge model order here to prevent ONO layouts and argue that edges with the same target should be bundled together.

Additionally, a common side effect of sorting ports by model order is that backward edges tend to be below normal edges. This is the case, since one typically declares backward edges after forward edges if they belong to a node declared later. Hence, backward edges often have a higher model order than forward edges and are hence sorted below normal edges since the first node and edge should be at the top.

Finally, the transitive closure has to be respected to prevent order violations and non-orders, as shown above for node orders. This is, however, only necessary, if incoming *unconnected ports* exist, i. e., if incoming ports with no edges connected to them exist.

### Preferring Node Order for Port Ordering

When preferring the node model order over the edge model order, ports should primarily be sorted by the node model order. To do this, one orders all incoming edges as in the previous case, but orders outgoing edges not by edge or port model order but by the node order of their long edge targets, e. g., the real nodes these edges eventually connect to.

*Transitive order* If the transitive closure of the already compared ports on a node already determines the ordering of the two ports, it should be respected.

## 5.5. Model Order in Crossing Minimization

*Outgoing ports* If both ports are outgoing, i. e., connect to a node in a succeeding layer, the two ports can be ordered by the node model order of their long edge target nodes. If they connect to the same node, the port or edge model order serves as a fallback.

*Incoming ports* Their connections to the previous layer orders two incoming ports.

*Outgoing and incoming port* Both ports can be statically ordered such that outgoing ports are before incoming ports or the other way around, since they will be on different sides of the node.

This concludes all crossing minimization pre-sorting strategies implemented and evaluated for this work, which are further generalized in the following.

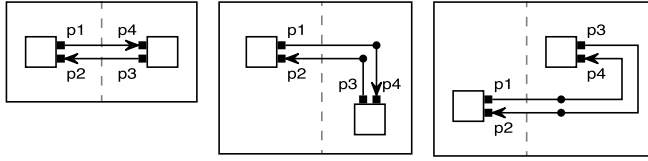
### **Generalizing Node and Port Sorting with Port Side Constraints, Feedback Edges, and Hyperedges**

Incoming ports on the WEST side and outgoing ports on the EAST side of a node are an assumption that is not always true. E. g., Schulze et al. [SSH14] proposed port side constraints, which allow incoming and outgoing ports to be constrained to any side. The port ordering presented above can, however, be easily adapted to port side constraints, which I also implemented in ELK but omitted above for clarity.

Three additional constraints have to be considered to support nodes with port side constraints. First, instead of sorting outgoing ports statically below incoming ports, I propose to sort ports by node side. Second, NORTH and SOUTH ports also require dummy nodes to order their edges, as seen in the second drawing in Figure 5.31. Third, feedback edges, i. e., edges going around a node as depicted in the last drawing in Figure 5.31, have a special kind of feedback dummy node that also has to be sorted determining the initial route of a feedback edge, e. g., below or above a node.

Figure 5.31 illustrates how ports orders of incoming ports might change based on the port sides of the source and target ports to avoid unnecessary edge crossings in during port pre-sorting.

## 5. Model Order in Layered Layout



**Figure 5.31.** Feedback edge port sorting illustrated. If the incoming ports p3 and p4 are on the NORTH or EAST side, their order is in reverse order of their source ports p1 and p2 in the previous layer.



**Figure 5.32.** Hyperedges need to consider all sources and all targets to make desired decisions based on model order.

In the first drawing in Figure 5.31, the ports p1 and p2 are ordered correctly from top to bottom with p4 and p3 being ordered based on their connection to p1 and p2. If, however, the incoming ports are on the NORTH or EAST side, p3 and p4 are placed in reverse order based on their connections to p1 and p2 to avoid crossings. The same is necessary if outgoing ports are on the WEST or SOUTH side.

Additionally, hyperedges have to be handled differently, which is currently only conceptualized and not implemented in ELK. When comparing a node with an incoming hyperedge, it might have multiple sources in the previous layer or an outgoing hyperedge might have multiple long edge targets, as seen in Figure 5.32. Here, one cannot just look at one source in the previous layer but has to consider multiple sources. Therefore, instead of comparing two nodes in the previous layer, one must count how many sources might be above or below each other or use a barycenter value to compare two hyperedges by model order. Hence, this should be implemented as part of future work on model order.

### Summarizing Model Order Pre-Sorting

The presented prefer-edges, nodes-and-edges, and prefer-nodes pre-sorting strategies create a deterministic and controllable initial order of nodes and ports before crossing minimization using model order. The main idea here is that—depending on the strategy— one sorts nodes and ports either by model order or their connections to a previous layer such that the ordering is consistent with the already sorted layers.

In the next step, the algorithm that actually minimizes the crossings, e. g., the barycenter algorithm [STT81], needs to make sure that the pre-ordering is only compromised if the developer desires it.

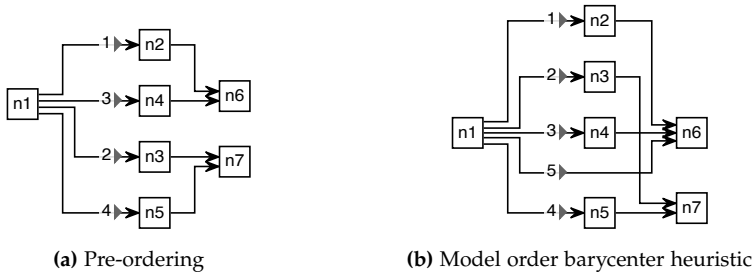
### 5.5.3 Model Order Barycenter Heuristic

If model order should be enforced while still minimizing edge crossings, model order has to be integrated into existing crossing minimization techniques such as the barycenter heuristic (see Figure 5.24). This is necessary since the barycenter heuristic reorders nodes and ports to prevent edge crossings. However, this reordering might violate the model order and may hence be undesired.

E. g., the crossing minimization with only model order pre-sorting might create a drawing such as Figure 5.33a. Here, the pre-ordered solution by nodes-and-edges is not crossing free. Therefore, the crossing minimization destroys the order of nodes during the barycenter heuristic to prevent these crossings. The model order barycenter heuristic solves this issue by enforcing the order of nodes, as depicted in Figure 5.33b, and keeping the node order stable despite a potential edge crossing. This means that the partial node model order is enforced while using the barycenter value for each node as a secondary total order. Here, enforcing the node order creates additional edge crossings and further increases the likeliness of local minima during crossing minimization. However, it also ensures stability of real nodes, which may be desired for big drawings and if the ordering conveys meaning.

Such a model order barycenter heuristic can be created as follows given a pre-ordered graph based on the node model order.

## 5. Model Order in Layered Layout



**Figure 5.33.** Pre-ordering by model order creates the result in Figure 5.33a. To create Figure 5.33b the order of nodes has to be enforced during crossing minimization.

*Transitive order* If the transitive closure of the already compared elements already determines the ordering of the elements, it should be respected.

*Real nodes* Two real nodes cannot diverge from their current order, which is already given by the pre-ordering step.

*Other nodes* If at least one of the nodes has no model order, the barycenter value has to be considered.

*Ports* Either the order of their nodes or, if both ports are on the same node, the barycenter value determines the ordering of ports.

Again, the transitive closure is necessary to not break the desired ordering, which again requires comparing real nodes first to not create a non-order since the partial node model order might conflict with the barycenter values. Other than that, real nodes do not reorder and everything else is determined by the barycenter heuristic.

Figure 5.33b shows the limitations of this approach. Conceptually unconstrained long edges, such as edge 5, are “stuck” because of the transitive ordering. If the first comparison yields that the edge 5 must be below  $n_4$ , the edge cannot move above  $n_1$ , which would be optimal, because of the transitive closure. Additionally, the  $5! = 120$  different port permutations in the first layer make it unlikely that a following random permutation of the ports actually tries the crossing minimal solution with constrained real



## 5.5. Model Order in Crossing Minimization

nodes, which places edge 5 at the top. Nevertheless, this simple addition to the barycenter heuristic keeps layouts stable by only allowing edge routes to change.

Figure 5.34 shows the Lingua Franca PacMan model with a layout that does not enforce the order of real nodes. Here, expanding the blinky ghost reorders all nodes and edges compromising stability and with it the mental map while breaking the intended order of the PacMan ghosts. As a result the order of the ghosts pinky, blinky, inky, and clyde<sup>5</sup> is again changed since showing the inner behavior constrains the hierarchical ports and reorders them.

Figure 5.34 compared to Figure 5.35 also highlights another aspect of edge crossings and stability. Namely, if edge crossings become sufficiently many, a few more do not hurt, as seen in Figure 5.35. In Figure 5.35, enforcing the order of real nodes during the barycenter heuristic keeps the drawing and the ghosts stable preserving their desired order, which might be desirable for bigger models, as seen in Figure 5.35a and Figure 5.35b compared to Figure 5.34a and Figure 5.34b. The several edge and hyperedge crossings do not matter much and can be mitigated by stability such that nothing really changes between Figure 5.34a and Figure 5.34b or by implementing edge selection highlighting, as shown in Figure 5.35 where the edge from controller to display is marked in orange. I argue that a diagram should here not be limited by layout alone but should also utilize whatever visualization technique might help to make a diagram more readable.

After presenting how an existing crossing minimization technique can adopt model order, I present how model order can work as a metric during crossing minimization in the following.

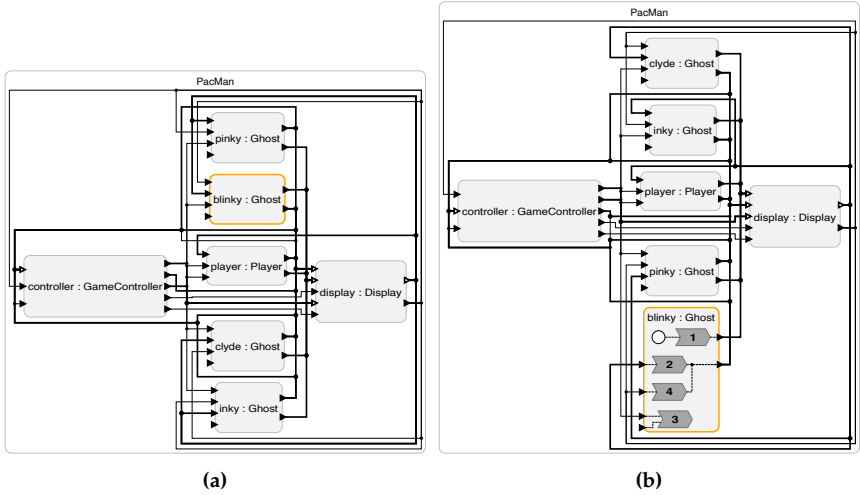
### 5.5.4 Crossing Minimization Weighted by Model Order

Edge crossings must not be the only metric that determines the “goodness” of a layout during crossing minimization. Model order can also be a metric, which has the advantage that it can measure intention, if the textual model carries intention, as detailed in Section 3.4. Since model order can measure

---

<sup>5</sup>Note that the order of the ghosts are already not preserved in Figure 5.34a since the initial pre-ordered solution was not crossing minimal.

## 5. Model Order in Layered Layout



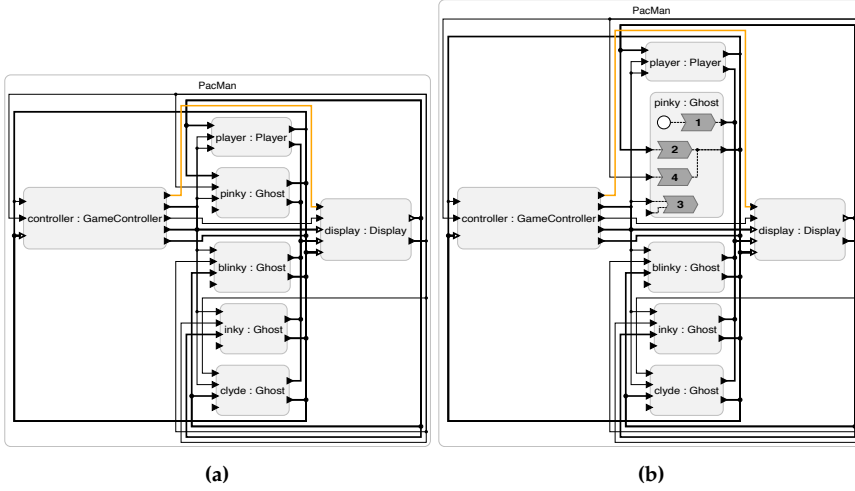
**Figure 5.34.** The PacMan Lingua Franca model with a marked blinky reactor in collapsed and expanded variant and no enforced order of real nodes during the barycenter heuristic.

intention, this metric can be used to determine a better layout: If pre-sorting does not help to get the desired solution and to capture the intention during crossing minimization and the drawing cannot be constrained, one needs an additional selection mechanism.

One example where such a selection mechanism might be necessary is Figure 5.26 on page 134. Here, the initial solution in Figure 5.26a is in a local minimum and creates crossings and is hence not crossing minimal. Trying random permutations of the free nodes and ports might randomly create Figure 5.26b before trying a starting configuration that results in Figure 5.26c. Hence, model order has to be added to the equation to achieve the desired result to overrule the first “best” solution in Figure 5.26b by Figure 5.26c.

Hence, I propose adding the model order metric described in [DH22]. In addition to the edge crossings, one counts the inversions in the node and port order when determining the “best” order during crossing minimization.

## 5.5. Model Order in Crossing Minimization

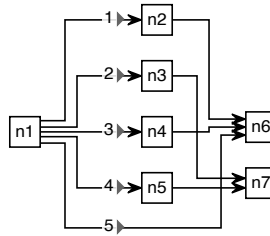


**Figure 5.35.** The PacMan Lingua Franca model in collapsed and expanded variant with enforced order of real nodes during the barycenter heuristic. A selected edge is highlighted in orange.

I.e., one counts how many of the node and port pairs are not ordered correctly and weights this against the crossings. By configuring the weight for nodes and ports, the importance of edge crossings against model order violations can be configured. Setting the weight to  $\frac{1}{c_{max}}$  where  $c_{max}$  is the maximum number of edge crossings makes the model order a true tie-breaker, meaning that it is only used as a secondary criterion. In the other extreme setting the model order weight for nodes and ports to  $c_{max}$  will always preserve the initial pre-sorted solution over possible solutions with fewer crossings since this weight will make any violation count more than the possible edge crossings.

Hence, the *weighted model order crossing minimization* finds a mostly ordered solution if the initial pre-ordered solution is not crossing minimal and results in a local minimum on the first crossing minimization run. E. g., the weighted node and port order violations as a secondary criterion to edge crossings create Figure 5.28a. This is the case even though the initial solution

## 5. Model Order in Layered Layout



**Figure 5.36.** Enforcing node and edge model order creates additional crossings. Compared to Figure 5.33, model order constrains the edge and node order to create this layout.

in Figure 5.28b is not crossing minimal and will hence be disregarded once a solution with fewer crossings is found.

### 5.5.5 Full Control Model Order Crossing Minimization

After presenting three ways to integrate model order into crossing minimization, I want to present how one can enforce the model order completely and only determine the order of nodes and ports by model order: Enforcing model order during crossing minimization works by not doing crossing minimization.

E. g., Section 5.5.3 illustrates how one can enforce the partial order of real nodes during the barycenter heuristic. To fully enforce model order, *full control model order crossing minimization* only executes the pre-sorting step before crossing minimization. Hence, full control model order crossing minimization creates the drawing in Figure 5.36 instead of the two alternatives depicted in Figure 5.33. Figure 5.36 creates additional crossings while keeping the drawing completely stable and eliminates the last source of random decision by skipping crossing minimization allowing to completely control the drawing by model order.

Hence, the full control strategy assumes that the developer always intends the ordering to be the way it was written down. I. e., it assumes that the model order matches the intention of the developer. Therefore, it

## 5.5. Model Order in Crossing Minimization

also makes the developer responsible for ONO layouts. E. g., if a model has too many edge crossings, the developer should change the model order to solve this issue, as I already argued for model order cycle breaking. Hence, a “weird” layout may point to an error in the textual model. However, this approach may require the developer to put more thought into the creation of the model to prevent drawings such as Figure 5.26a. In this example SCChart, the developer should change the edge order such that the connector state is in the middle with priority 2 instead of 3. Hence, the developer must also adjust the transition guards to not change the semantics of the SCChart model.

Here, hyperedges, as well as all cases of “static decisions” pose a problem since they cannot be fully controlled by model order. Hence, dummy nodes together with dangling nodes, unconnected ports, and feedback as well as NORTH or SOUTH edges pose a problem. E. g., Figure 5.28a cannot be created by fully controlling the model order, and an ordering of feedback edges that diverges from the base case in Figure 5.31 cannot be created without greedy post-processing, e. g., by the greedy switch heuristic proposed by Eades and Kelly [EK86].

### 5.5.6 Summarizing Model Order Crossing Minimization

To summarize, there are three different ways to use model order for crossing minimization. First, one can make sure that the initial order conforms to the model order, second, one can make sure that the crossing minimization strategy, e. g., the barycenter heuristic, uses model order as a constraint, and third, one can use model order as an additional weighted metric together with edge crossings to determine the “goodness” of a solution.

I presented three ways of using the model order depending on the importance of node model order compared to the port or edge model order. The prefer-edges strategy prefers the edge model order and primarily uses the connections to previous layers to order nodes and edges. The nodes-and-edges approach uses both the node and the edge model order which might be conflicting and hence create edge crossings. The prefer-nodes strategy primarily uses the node model order.

Another dimension of model order is how much it constrains the model.

## 5. Model Order in Layered Layout

Either one can only create an initial order by crossing minimization pre-ordering and use model order as a secondary metric to edge crossings, model order can be enforced on nodes during the barycenter heuristic, or model order can fully control the order of nodes in a layer and ports on a node, which requires the model order to match exactly with the developer intention.

In the following, I continue with a short discussion on model order node placement in Section 5.6 and its usefulness.

### 5.6 Model Order in Node Placement

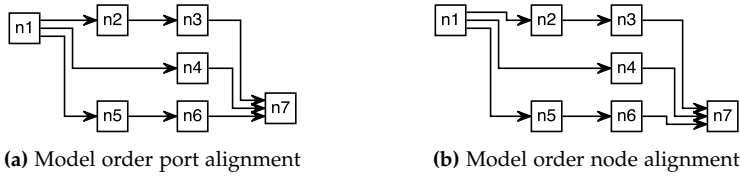
Order comes very natural to the topological phases of the layered algorithm, namely cycle breaking, layer assignment, and crossing minimization, as presented in the previous sections. Node placement, however, does not determine order but rather concrete coordinates using a given division of nodes into layers and node orderings in each layer.

Hence, node placement cannot use model order to create a desired ordering to improve the secondary notation. Instead, node placement can potentially create desired alignment of nodes and of ports to express secondary notation. In addition to alignment, node placement may create stability and enable control over the layout on a coordinate level by grouping elements together. However, model order is only one-dimensional and cannot express a grouping in addition to a desired ordering. Therefore, model order node placement will probably not be effective.

Nevertheless, I present an idea for a *model order node placement algorithm* in the following to illustrate the potential it might have. However, since I did not yet find a suitable language to test a model order node placement algorithm with, I only conceptually sketch how one might want to control the alignment of nodes using such an algorithm.

There are two ways of alignment: alignment of nodes and alignment of ports. I suggest aligning a node (or port) with a node (or port) in the previous layer if the node (or parent node in case of a port) has a model order that is directly below the succeeding node (or parent node), as depicted in Figure 5.37. In Figure 5.37a model order node placement aligns

## 5.7. Group Model Order for Layered Layouts



**Figure 5.37.** Ports or nodes align themselves based on model order.

the ports connecting n2 and n1 and the ports connecting n7 and n6 based on model order. In Figure 5.37b model order node placement aligns the nodes. However, algorithm such as Brandes and Köpf [BK02] or network simplex [GKN+93] node placer would potentially align n1 and n7 with n4 to balance the layout.

Potentially, such a model order node placer can hence be an extension to existing algorithms such as Brandes and Köpf or network simplex node placement by adding alignment constraints based on model order.

However, because of a lacking use-case, I do not further explore different possible node placement ideas other than the short sketch above.

## 5.7 Group Model Order for Layered Layouts

After presenting viable model order algorithms for the phases of the layered algorithm, I will add an extension to deal with modeling languages such as Lingua Franca.

In all previous section regarding cycle breaking, layer assignment, crossing minimization, the layout algorithms assumed that there is only one kind of node in a model, which may not reflect reality, as seen in Figure 5.38. Here, Figure 5.38b shows the desired layout and Figure 5.38c a worse layout using the strict model order cycle breaker. Since the order of the nodes in Figure 5.38a is separated into two model orders groups for actions and reactions, the strict model order cycle breaker creates undesired edge reversals based on the total model order of nodes.

Hence, I explore how such model order groups should influence cycle breaking, layer assignment, and crossing minimization strategies for mod-

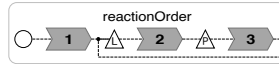
## 5. Model Order in Layered Layout

```
main reactor {  
  logical action a:char*;  
  physical action b:char*;  

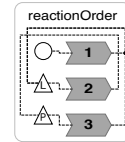
```

```
  reaction (startup) -> a {==}  
  reaction (a) -> b {==}  
  reaction (b) -> a {==}  
}
```

(a) Abbreviated textual model



(b) OYES layout



(c) ONO layout

**Figure 5.38.** Declaring actions before reactions in the textual model does not mean that actions should be before reactions in the diagram.

eling languages that consist of semantically different nodes ordered into groups in the following.

### 5.7.1 Group Model Order in Cycle Breaking

All presented model order cycle breaking strategies assume that all nodes and edges adhere to a total order. In languages such as Lingua Franca, however, multiple partial orderings may order nodes and edges. Because of textual ordering conventions, Lingua Franca declares different elements in their own model order groups, as seen in Figure 5.38. E.g., actions are declared independent of reactions, as seen in Figure 5.38a. The order between different semantic elements does here not express intention. Hence, only because actions are declared before reactions, we should neither reverse edges from reactions to actions nor reverse edges from actions to reactions.

In the following, I present three different ways of dealing with model order groups on the example of Lingua Franca based on the thesis of Riepe [Rie24] I advised. One can either aim to create a total order by ordering the model order groups, fall back to the graph structure to make decisions, or spend more time, e.g., by calculating the strongly connected components for cycle breaking, to make more informed decisions.



## 5.7. Group Model Order for Layered Layouts

### Creating a Total Order

Since group model order means that the model order is only valid in the respective ordering group, different semantic elements are not comparable by model order, as seen in Figure 5.38. Here, only reactions can be compared to reactions but comparing the reaction model order to the action model order results in the ONO layout in Figure 5.38c.

However, if one needs full control over the flow in their layout, e. g., to create Figure 5.38c, a total group model order is necessary to control the layout by model order. Hence, the trivial “solution” to multiple partial group model orders is creating a *total group model order* by ordering the different model order groups. A total group model order requires modelers to potentially violate ordering conventions in their model.

Manually adjusting the model order in Figure 5.38a to adhere to a total model order to create Figure 5.38b by violating the textual ordering conventions would be possible but may be prone to errors and places a burden on the developer. Moreover, developers would again manually layout the graph, which I criticized in WYSIWYG diagram editors. Strict group model order cycle breaking is hence not desired for Lingua Franca and possibly other languages with multiple ordering groups.

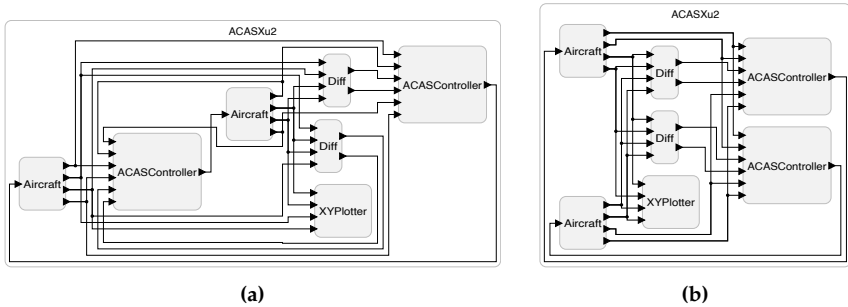
A group model order cycle breaking strategy for Lingua Franca should hence take the structure of the graph into account and not make decisions based exclusively on model order.

### Using the Structure

The structure of the graph, i. e., its edges, determines the desired flow creating Figure 5.38b rather than Figure 5.38c. This is possible by using the depth-first, breadth-first, or greedy cycle breaker presented in Section 5.3.3 and Section 5.3.2 with a total group model order as its iteration order and secondary criterion. This works, since model order is here only a secondary criterion, and combining the different model order groups to form a total order does influence fewer decisions.

However, a *depth-first group model order cycle breaker* cannot identify the correct edge to reverse to emphasize loops in very connected graphs, as seen in Figure 5.39. Figure 5.39a not only illustrates the problem of the

## 5. Model Order in Layered Layout



**Figure 5.39.** In the ACASXu2 Lingua Franca model taken from the Lingua Franca playground nearly every reactor is connected to every other reactor. Figure 5.39a utilizes depth-first model order cycle breaking while Figure 5.39b utilizes the strongly connected components to reverse edges based on the existing cycles.

depth-first cycle breaker, which it shares with the depth-first group model order cycle breaker: the drawing gets too wide and has an undesirable flow. Note that the Lingua Franca example in Figure 5.39 models only reactors but could potentially have different semantic elements. However, the underlying problem does also occur for models with different model order groups.

Figure 5.39 shows the ACASXu2 Lingua Franca model for a collision avoidance system consisting of two aircraft for which the position difference is calculated to make informed course corrections in a controller, while also plotting the path of the aircraft. In Figure 5.39a, the drawing created by depth-first cycle breaking gets unnecessarily wide. Since the controllers are connected with most of the graph, the layout in Figure 5.39a does not show the symmetric structure of the ACASXu2 model. The depth-first cycle breaker cannot create the desired solution in Figure 5.39b. Figure 5.39b instead shows that the two aircraft somehow get feedback from the ACAS-Controller reactors, which make their decision based on the Diff reactors, which calculate the position and route difference of the two aircraft.

Similarly, the breadth-first and the greedy cycle breakers create layouts similar to Figure 5.39a since all three algorithms do not take enough of the model order into account. Moreover, they are limited by structural criteria

## 5.7. Group Model Order for Layered Layouts

and by the fact that they work in linear time  $\mathcal{O}(|V| + |E|)$ . I propose to solve the lack of structure of the depth-first cycle breaker by spending more than linear time to calculate the strongly connected components and to use the model order, which captures the intended ordering.

### Considering Strongly Connected Components

Cycle breaking is an NP-hard problem [Kar72]. Hence, just spending more time to get a better layout, e.g., by calculating the strongly connected components of a graph, does not work for general graph drawing problems without threatening reactivity of the diagram tool since the runtime of such an algorithm would become quadratic or worse.

However, in the application of Model-Driven Engineering (MDE), e.g., for Lingua Franca and SCCharts, the graph sizes are typically quite small. A typical state machine has no more than 20 states, as presented in Table 4.1 in Chapter 4. A state machine with over 20 states would be considered very complex. Hence, state machines typically employ hierarchy to structure state machines by using superstates, i.e., states with inner behavior, e.g., another state machine in each state. MDE even recommends introducing hierarchy to keep the models human-readable and understandable. This does not only work for state machines but any control- or data-flow model employed in MDE and reduces the complexity of the model and the subgraphs the corresponding layout problem consists of.

Hence, I can assume few nodes to layout, and that the time complexity of the employed algorithms might not be an issue. Therefore, an algorithm may be allowed to spend more time determining the desired flow based on the group model order to create layouts with better secondary notation using model order.

For this use-case, I propose the *strongly connected component model order cycle breaker* that calculates all strongly connected components using Tarjan's algorithm [Tar72], and uses model order to reverse the desired edges in them, as shown in Algorithm 11.

In Algorithm 11, Tarjan's algorithm first computes all strongly connected components. If at least one exists, `findMinMaxModelOrder` returns the edges that should be reversed, i.e., all edges leading to the node with the minimum

## 5. Model Order in Layered Layout

total group model order or all edges originating at the maximum total group model order node, as visualized in Figure 5.40. E. g., Figure 5.40 shows two strongly connected components marked in red and orange. The maximum total group model order nodes are visualized with a dotted border. Their incoming and outgoing edges are also dotted, showing that they might be reversed. After reversing the edges, the process continues by again calculating the strongly connected components until none remain in the graph. The resulting algorithm executes in  $O((|V| + |E|) \cdot |V|)$  for executing Tarjan's with runtime  $O(|V| + |E|)$  algorithm  $|V|$  times until all potential strongly connected components are removed.

Note that finding the maximum total group model order requires knowing the desired order between different semantic elements, which I further investigated in Section 5.9.3, Section 5.9.6, and Section 5.9.7.

As an alternative to the proposed strongly connected component model order cycle breaker, one can consider a *strongly connected component degree cycle breaker*. This cycle breaker uses the degree of a node instead of the total group model order to identify the edges to reverse in a strongly connected component similar to the greedy cycle breaker (see Section 5.3.2). Here, all incoming edges to the highest degree node are reversed instead, leaving model order as a tie-breaker if nodes have the same maximum degree, as proposed for the greedy model order cycle breaker. This results in more informed decisions compared to the greedy cycle breaker and may also be used for graphs without different semantic elements, as seen in Figure 5.39.

---

**Algorithm 11: Strongly Connected Component Model Order Cycle Breaker**

---

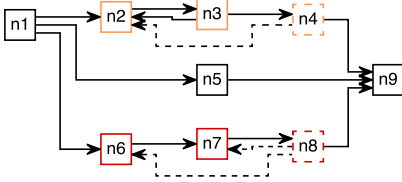
**Input:** A digraph  $G = \langle V, E \rangle$

**Output:** An acyclic digraph

```
1  $sccs := \text{tarjan}(G)$ 
2 while ( $sccs \neq \emptyset$ )
3    $backwardEdges := \text{findMinMaxModelOrder}(sccs, G)$ 
4    $G := \text{reverseEdges}(G, backwardEdges)$ 
5    $sccs := \text{tarjan}(G)$ 
6 return  $G$ 
```

---

## 5.8. Solving Layered Topology Efficiently using Model Order



**Figure 5.40.** Example model for a strongly connected components cycle breaker. The two strongly connected components are marked in orange and red. The maximum total group model order elements and its edges are marked with dotted lines.

### 5.7.2 Considering Group Model Order in Layer Assignment and Crossing Minimization

In contrast to the group model order cycle breaking, group model order layer assignment and crossing minimization pre-sorting always requires a total group model order since one cannot rely on structure given by the direction of edges.

Model order layer assignment can use a total group model order as iteration order or utilize it as a tie-breaker in layer assignment post-processing (see Section 5.4.3).

During crossing minimization, the pre-sorting has to be done with a total group model order. However, the model order barycenter heuristic can consider the group model order by only constraining real nodes belonging to the same ordering group and model order violations should only count between nodes and ports belonging to the same group. Hence, full control model order crossing minimization is not recommended for model order groups, since a total order is here necessary for the model order strategies, which enforces potentially undesired orderings.

## 5.8 Solving Layered Topology Efficiently using Model Order

The already presented model order strategies for layering and crossing minimization assume that intermediate processors (see Figure 5.4) might not consider model order. Specifically, dummy nodes are typically not sorted by model order, and layering strategies might order nodes in a layer not by model order but by the time the respective algorithm assigns the

## 5. Model Order in Layered Layout

node to a given layer.

Hence, nodes and edges have to be sorted before each model order strategy to ensure that it works as intended, which costs time. If one, however, starts on a clean slate, one can immediately make the right decisions and avoid compromising the ordering altogether to create an *efficient layered model order topology algorithm*. Based on [DH24a], I want to sketch an algorithm that solves the topology of a layered layout efficiently as a one-pass algorithm.

### Solving Cycle Breaking Efficiently by Model Order

The strict model order cycle breaker (see Section 5.3.1) with runtime  $\mathcal{O}(|E|)$  is already as efficient as it gets for cyclic graphs. Hence, efficient model order cycle breaking uses the strict model order cycle breaker. Here, I assume that node model order does indeed represent the desired flow to employ this strategy without causing undesired backward edges.

### Solving Layer Assignment Efficiently by Model Order

Creating a layering for real nodes what respects the model order was already presented in Section 5.4.2. However, the solution for layer assignment presented in Section 5.4.2 requires a post-processor to add the dummy nodes, which might violate the model order. The insertion order of dummy nodes into layers matters for the following crossing minimization step, which should be able to omit the pre-sorting step in the efficient solution.

An efficient model order layering algorithm that directly inserts dummy nodes and creates a desired order of nodes in the layer works as follows, as visualized in Figure 5.41.

The first node is placed in the first position in the first layer, as seen in Figure 5.41a. After that there are two options for succeeding nodes. Either they are not connected to any node in the current layer or they are connected. If a new node is not connected, the efficient model order layerer places it in the current layer below the last node, as it is the case for  $n_2$  in Figure 5.41b. If it is connected to the current layer, as it is the case for  $n_3$  in Figure 5.41c, the efficient model order layerer places it in a new layer

## 5.8. Solving Layered Topology Efficiently using Model Order

making the second layer the new current layer. Here, it is important to keep track of all currently not connected dummy nodes. E. g., in Figure 5.41c  $n_1$  has a connection to the currently not placed  $n_5$ . So either the edge leading to  $n_5$  can go to the second layer or it will go to a succeeding layer. If it goes to a succeeding layer, one has to find a suitable position for dummy nodes. Here, the connection from  $n_2$  to  $n_3$  determines the potential position for a dummy node. Since the edge below the outgoing edge of  $n_1$  is the edge from  $n_2$  to  $n_3$ , any edge going below  $n_3$  will produce a crossing. Hence, I put the potential dummy node above  $n_3$  to prevent that. The ports below and above our currently not connected port are here the reference and limit the number of possible dummy node positions for succeeding layers to two. Hence, the drawing can be controlled such that a long edge orients itself on the edge above it or on the edge below it if there is no edge above it. Figure 5.41d shows how  $n_4$  is added in the same layer since it is not connected to  $n_3$ . Figure 5.41e shows how an edge route for the outgoing edge of  $n_1$  is determined once the algorithm places  $n_5$  in the new layer, and Figure 5.41f shows the final layering with  $n_6$  inserted below  $n_5$ .

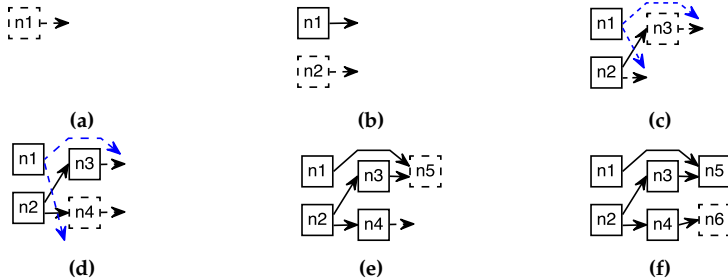
Note that there cannot be a case where neither an edge above nor an edge below exists. Otherwise, there would be no long edge. Assume that there is no neighboring edge and the current edge is a long edge. This means that the next layer can be merged with the current one without violating the model order layering constraints (see Section 5.4.2). This will either create neighboring edges, make the current edge a short edge, or the process can be repeated until one of the first two conditions is true.

Hence, continuing to place nodes and dummy nodes in the described manner creates a layered graph controlled by model order.

### Solving Crossing Minimization Efficiently by Model Order

Since efficient model order layer assignment already ordered the real nodes, the dummy nodes, and the ports correctly, as shown in Figure 5.41f, no crossing minimization is necessary. If one assumes that model order is the desired order, the topology of the edge routes and the vertical and horizontal order of nodes is already as intended. Hence, the layout can be changed by changing the model order as desired.

## 5. Model Order in Layered Layout



**Figure 5.41.** The efficient model order layering algorithm visualized. New nodes are marked in dashed-black, edge route alternatives in dashed-blue, and unconnected edges in dashed-black.

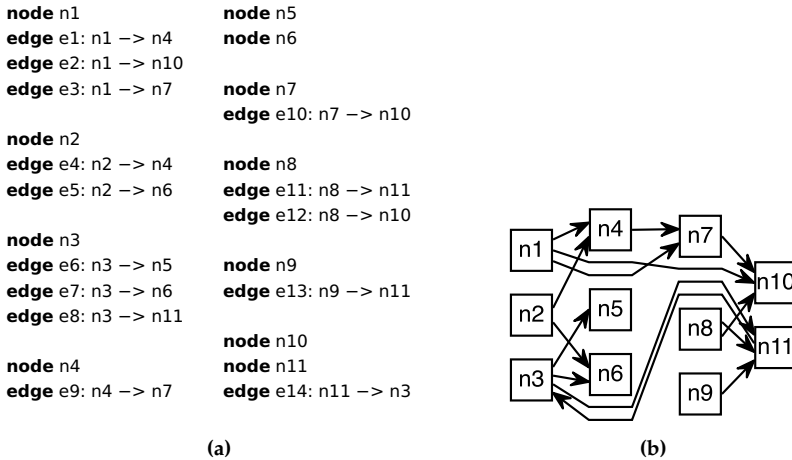
Moreover, using the efficient approach, developers have full control over the layout using the model order if the underlying model allows a breadth-first node order and edges can be freely reordered. An example for such a model can be seen in Figure 5.42. Here, all problems in the layout can be solved by changing the textual model in Figure 5.42a. Reordering the edges  $e_2$  and  $e_3$  prevents the crossing between the outgoing edges of  $n_1$ . Similarly, the order of  $n_5$  and  $n_6$  can be changed together with moving  $e_6$  below  $e_8$  to prevent the edge crossing between the outgoing edges of  $n_2$  and  $n_3$ . Additionally, the outgoing edges of  $n_8$  can be reordered preventing another crossing.

However, there might still be ONO drawings, such as ones depicted in Figure 5.43, which cannot be completely controlled by model order. Here, greedy post-processing might be beneficial, e.g., by the greedy switch heuristic by Eades and Kelly [EK86]. In Figure 5.43a model order has no way to control the route of outgoing long edges of  $n_2$  compared to the dangling node  $n_4$ . Here, the two edges might either take route A (gray) or route B (blue) but it is not possible that one takes route A and the second takes route B.

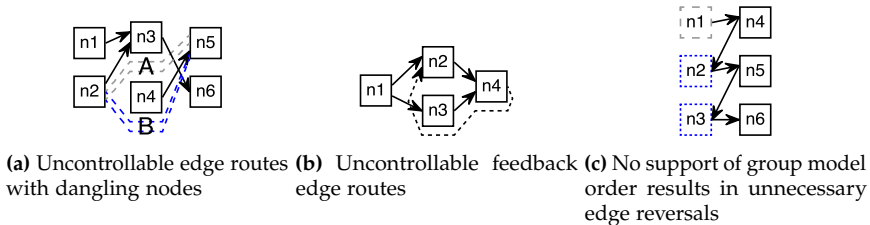
Figure 5.43b illustrates the same problem for feedback edges. Here, one cannot decide whether the feedback edge (dashed-black) should be routed below or above its source  $n_4$  without doing crossing minimization.



## 5.8. Solving Layered Topology Efficiently using Model Order



**Figure 5.42.** An example for the efficient strict model order topology algorithm.



**Figure 5.43.** ONO models using the efficient model order topology algorithm.

Additionally, the efficient model order topology algorithm requires a total model order. E. g., Figure 5.43c illustrates how control does not work as intended if there are model order groups, e. g., the dashed-gray nodes, dotted-blue nodes, and black nodes since a total model order is required for the strict model order cycle breaker. Here, the backward edge from n4 to n2 and from n5 to n3 is not intended since both nodes belong to different ordering groups.

The efficient model order topology algorithm should hence be post-

## 5. Model Order in Layered Layout

processed to capture these pitfalls. Using the algorithm without post-processing might only be beneficial if the model order is to be taken literally, or if one desires a quick, structured, and stable graph layout. E. g., an initial solution for force or stress based layout, which might otherwise use random decisions, might benefit from the efficient model order topology algorithm.

### 5.9 Evaluating the Layered Algorithm

The previous sections mostly focused on the question of how model order can be integrated into automatic layout (*IMPL*). This section instead shows the effect of model order strategies on readability, stability, and control (*Eval*) on the example of SCCharts and Lingua Franca. Moreover, this section provides the evaluation data for model order applied to different modeling languages, which will be discussed in Part III.

Most of the research and studies presented in the following, are already published in papers [DH22; DRv23; DH24b; DH24c] and student thesis working on model order [Rie22; Rie24]. Hence, this section summarizes their findings and puts them into context.

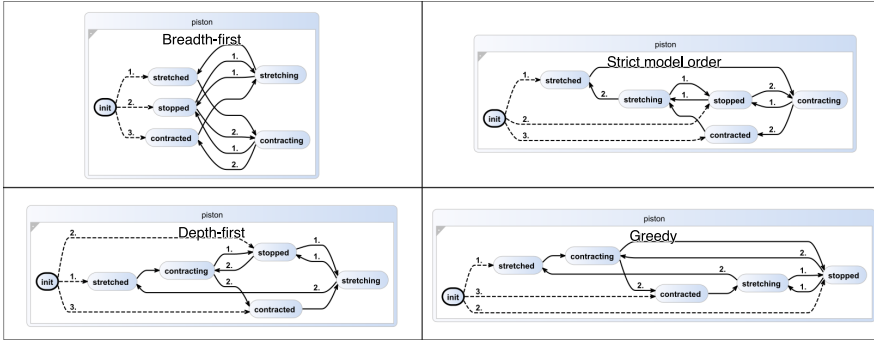
#### 5.9.1 Study 1: Qualitative Evaluation of SCCharts Cycle Breaking

This study evaluates cycle breaking for SCCharts based on qualitative feedback. It is based on the work of Riepe [Rie22] with the goal to evaluate the strict model order cycle breaker (see Section 5.3.1) for SCCharts.

Figure 5.44 illustrates the main motivation for this study. Here, the breadth-first edge model order cycle breaker creates the top-left layout, the strict model order cycle breaker the top-right layout, the depth-first edge model order cycle breaker the bottom-left layout, and the greedy cycle breaker the bottom-right layout.

Especially the top-left layout is interesting since, although it has more crossings and backward edges, modeling experts deemed it the best of the four layouts. By aligning the states, this layout shows that there are states for static positions, i. e., stretched, stopped, and contracted, and states

## 5.9. Evaluating the Layered Algorithm



**Figure 5.44.** The obfuscated piston SCChart used by Riepe [Rie22] as one of the stimuli.

for movement, i.e., stretching and contracting. Hence, this study evaluates the breadth-first edge model order cycle breaker (see Section 5.3.3), which created the interesting model in the top-left, to find out whether the breadth-first cycle breaker creates desirable layouts by highlighting the potential intention in a model.

By comparing the breadth-first cycle breaker to the strict model order cycle breaker, one can determine how often a breadth-first model order occurs in SCCharts models. At the same time, evaluating the strict model order cycle breaker may show whether the developer intention matches with the node model order for SCCharts.

Including the depth-first edge model order cycle breaker (see Section 5.3.3) in the evaluation promises a potential match of the depth-first order with the model order.

Moreover, the greedy cycle breaker (see Section 5.3.2) is included since it was the initial default cycle breaker for SCCharts, which was used when creating the evaluated SCCharts models. Hence, ordering had no controllable effect on the layout when the models were created.

The study was conducted as follows. Participants completed an online questionnaire. They were first questioned about their experience in graph drawing and their background. A total of 27 participants completed the questionnaire. Of these 27 participants 17 worked in the IT sector, 14 had

5. Model Order in Layered Layout

previous experience with any statecharts dialect, and eight had already worked with a layered algorithm. The confidence of the participants on the topic of graph drawing is depicted on a five-point Likert scale in Table 5.1. It shows that participants had varying confidence in the topic. However, their experience or confidence did not significantly change their rating of the different layouts.

**Table 5.1.** Study 1: Confidence of the participants of study 1 on the topic of automatic graph drawing on a five-point Likert scale.

No confidence	Not confident	Unsure	Confident	Very confident
9	4	6	4	4

After participants completed the short introduction, they were shown four graphs, as depicted in Figure 5.44, and were asked to answer the following questions, if applicable, on a five-point Likert scale.

*First Impression* Take a quick look (~30s) at the different layout options and rank them according to your preference, starting with the best.

*Readability I* Do you feel like the states are too crowded or too far away from each other?

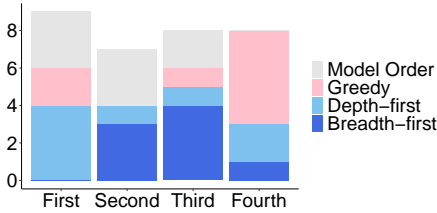
*Readability II* Evaluate for each graph: Edges are easy to follow?

*State Grouping* Take a look at the node labels. Do you feel like any layout option creates groups of nodes that are thematically related?

*Final Impression* After working with this graph repeat the evaluation of your preference.

Participants had one training tasks and then repeated the full questionnaire for the first 3 stimuli. To not prolong the questionnaire, the next five stimuli only asked for a first impression keeping the average completion time at 30 minutes. Since the expected number of participants was small, the order of graphs was not permuted. The order in which the different strategies appear was also not permuted, which should be done if the survey would be repeated.

## 5.9. Evaluating the Layered Algorithm



**Figure 5.45.** Study 1: The average performance and the resulting ranking per graph. For each graph each participant ranks a strategy either first, second, third, or fourth such that the y-axis shows how often each strategy had each total average ranking. Note that the first bar has nine and the second bar seven cases since the greedy cycle breaker and the model order cycle breaker shared first place for one graph.

The performance of each graph was determined by the number of votes a strategy got, with 1 representing the best rank and 4 the worst rank, as detailed below.

$$performance(s) = \frac{\sum_{i=1}^4 r_i \cdot i}{\text{total votes}}$$

The average performance ranking of the four drawing strategies can be seen in Figure 5.45. Here, the y-axis shows how often each strategy reached the respective ranking. Therefore, each bar should have a height of eight since eight graphs were evaluated. However, since the greedy cycle breaker and the model order cycle breaker performed identically for graph 2, the first bar has nine entries and the second bar has seven entries.

The study resulted in the best performance value for the strict model order cycle breaker with a performance of 2.11 followed by the depth-first edge model order cycle breaker with a performance of 2.17. Hence, the difference was not significant and given the sample size cannot be generalized. However, the greedy and breadth-first cycle breaker clearly performed worse.

As reasons for their ranking participants mentioned and ranking clear edges routes, crowdedness, and thematic grouping, as shown in Table 5.2. Furthermore, participants mentioned the following criteria as important in their decision-making.

1. Initial node in the first layer and final node(s) in the last layer (7 Responses)

## 5. Model Order in Layered Layout

**Table 5.2.** Study 1: Importance of aesthetic criteria

	Very Unim- portant	Unim- portant	Neither nor	Impor- tant	Very Impor- tant
Clear Edges	1	0	3	15	8
Crowdedness	2	4	8	8	5
Thematic Grouping	2	5	1	11	8

### 2. Symmetry (5 Responses)

### 3. Edge crossings (4 Responses)

### 4. Understandable node labels (3 Responses)

The interesting part here is that participants not only rated clear edge routes highly but also thematic grouping highly. Moreover, they often mentioned domain specific secondary notation such as the position of initial and final states. This explains the results of the ranking. Depth-first edge model order cycle breaking focuses on the structure, i. e., the direction of edges, and may hence produce good results based on that including the position of the initial and final states. Strict model order cycle breaking captures intention and creates grouping. Other strategies cannot control the flow of a graph in the same way. Especially the greedy cycle breaker, which relies on random decisions and performed worst, often creates ONO layouts. The breadth-first cycle breaker ranks third since it utilizes the graph structure not in way SCCharts developers think about a model. SCCharts developers think in depth-first control flow branches.

This study showed that SCCharts developers and potentially other state machine or control flow developers think and organize the flow of a model depth-first. Moreover, the node model order seems to match the intended flow in most of the cases. Overall, the most controlling strict model order cycle breaker that keeps the flow of a model completely stable was rated best by the participants. As a practical consequence of the study, SCCharts added the strict model order cycle breaker as a default.

In the following quantitative study, I discuss the influence of the strict model order cycle breaker on SCCharts. Moreover, the strict model order

## 5.9. Evaluating the Layered Algorithm

cycle breaker is again discussed and evaluated regarding suitability as the default strategy for SCCharts in Section 5.9.8.

Future work on cycle breaking could compare a sufficient number of new models created with the strict model order cycle breaker and compare them to the following results. Moreover, if I were to do the study again, I would try using the breadth-first node model order and the depth-first node model order cycle breakers instead of the edge model order variant. I suspect that this might potentially improve the result of both cycle breakers if the node model order is a better tie-breaker than the edge model order.

### 5.9.2 Study 2: Quantitative Evaluation of SCCharts Cycle Breaking

In addition to the qualitative evaluation above, Riepe [Rie22] also qualitatively analyzed how the breadth-first edge model order cycle breaker (see Section 5.3.3), the depth-first edge model order cycle breaker (see Section 5.3.3), the greedy cycle breaker (see Section 5.3.2), and the strict model order cycle breaker (see Section 5.3.1) performed in terms of area, backward edges, edge crossings, edge length, and layer count. Together with Study 1 in Section 5.9.1, this quantifies the potential effect of the strict model order cycle breaker on readability for SCCharts.

The evaluated dataset consists of 419 SCCharts taken from student solution of lectures regarding synchronous languages and embedded systems, with three to 300 nodes per model with outliers of up to 29085 nodes in a hierarchical model. These result were filtered to 265 graphs for which at least one strategy created a different result, removing all trivial cycle breaking problems.

The quantitative evaluation resulted in the following. The difference in area is not significant with a p-value  $> 0.98$ . The Kruskal-Wallis test indicates with a p-value of  $3.14 \cdot 10^{-13}$  that the breadth-first cycle breaker produces more backward edges than the other strategies. For the edge length, the Kruskal-Wallis test indicates with a p-value of 0.022 a difference between the strategies. The breath-first cycle breaker produces layouts with the smallest edge length followed by the depth-first and model order cycle breaker for which the Wilcoxon tests fails to show any significant differences,

## 5. Model Order in Layered Layout

while the greedy cycle breaker performs worst. For the number of layers, the differences are not significant with a p-value of 0.41. The number of edge crossings as a result of the chosen cycle breaking strategy also shows no significant difference with a p-value of 0.302. When evaluating against the 106 graphs with at least one edge crossing, there is also no significant difference with a p-value of 0.092. Hence, there is no significant quantitative difference for readability of the flow created by the different strategies. Moreover, the number of edge crossings, which might be more in denser graphs, and the required drawing area is not affected by the choice of the cycle breaking strategy.

Hence, the results show that the strict model order cycle breaker, which completely constrains the flow of a model allowing complete control while keeping the flow completely stable, does not significantly threaten readability for SCCharts. Consequently, one can assume that even though model order did not affect the layout at the model creation, developers naturally employed model order in their textual models to indicate flow. If this would not be the case, the strict model order cycle breaker would create significantly more backward edges. Together with the preceding study, this shows that even student models, i.e., models by novices, employ model order in SCCharts such that using the strict model order cycle breaker does not threaten readability. Hence, if one desires control and stability, the strict model order cycle breaker should be used.

### 5.9.3 Study 3: Quantitative Evaluation of Lingua Franca Cycle Breaking

In his master thesis [Rie24], Riepe again evaluated model order cycle breaking, here at the basis of Lingua Franca models created for demonstrations, tutorials, and documentation, which I consider to be models made by experts, i.e., Lingua Franca developers and researchers. This study evaluated 378 Lingua Franca models using the following cycle breakers.

*B* The breadth-first node model order cycle breaker.

*D* The depth-first node model order cycle breaker



## 5.9. Evaluating the Layered Algorithm

**G** The greedy cycle breaker.

**LA** The model order look-ahead cycle breaker. This cycle breaker was disregarded after this study and this hence not described in this work.

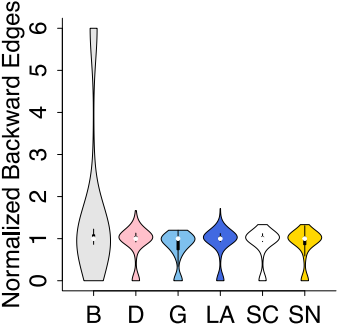
**SC** The strongly connected component model order cycle breaker described in Section 5.7.1.

**SN** The *strongly connected component node type cycle breaker*. This cycle breaker preferably reverses the edges to the reactor with the lowest model order. Hence, reactors are handled separately by the SN cycle breaker since they are deemed more important.

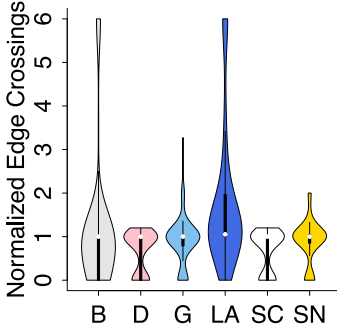
The breadth-first and depth-first node model order cycle breakers (see Section 5.3.3) were selected to see whether a simple structural solution is sufficient and to identify a possible ordering preference in Lingua Franca models. Compared to the evaluation in Section 5.9.1 and Section 5.9.2, this study uses the node model order as the depth- and breadth-first visiting order. The greedy cycle breaker serves as a baseline since it was the current default strategy. I expected that it would again perform bad in terms of intention since it does not take model order into account. The model order look-ahead cycle breaker aims to find potential backward edges by only looking at the potential connections from the same ordering group. E. g., a reaction would look at the node model order of a potential reaction connected to one of the following actions. If the model order of the connected reaction would be smaller, the edge would be reversed. The strongly connected component cycle breakers SC and SN (see Section 5.7.1) both aim to make better decisions by spending more time to find strongly connected components. Here, it needs to be evaluated if model order can even be used to control the layout of these algorithms and whether spending more time by using the strongly connected component cycle breaker is worth the additional time spend. The strict model order cycle breaker was here not included since it cannot sufficiently deal with model order groups.

Figure 5.46 to 5.49 show the resulting normalized number of backward edges and edge crossings as well as the execution time in milliseconds and the actual aspect ratio of the layout using the six different cycle breakers.

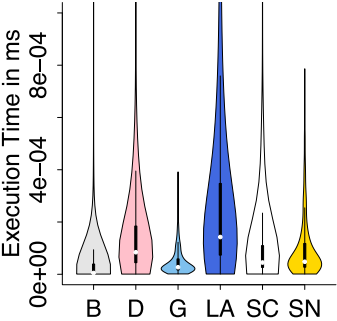
5. Model Order in Layered Layout



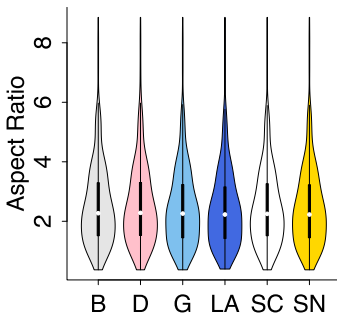
**Figure 5.46.** Study 3: Number of backward edges normalized by the mean number of backward edges [Rie24].



**Figure 5.47.** Study 3: Number of edge crossings normalized by the mean number of edge crossings [Rie24].



**Figure 5.48.** Study 3: Time to layout in milliseconds [Rie24].



**Figure 5.49.** Study 3: Aspect ratio of the resulting drawing [Rie24].

## 5.9. Evaluating the Layered Algorithm

The breadth-first cycle breaking strategy performs worst in terms of backward edges while the greedy cycle breaker reduces the backward edges most effectively, as seen in Figure 5.46.

Figure 5.47 shows the effect of the cycle breaking strategies on the crowdedness of a layer measured by the resulting edge crossings. Fewer layers often imply fuller layers with potentially more edge crossings, as seen in Figure 5.44. Here, the top-left breadth-first layout has only three layers and the most edge crossings. In terms of edge crossings, the model order look ahead cycle breaker performs worst followed by the breadth first node order cycle breaker. This is the case since both create fewer layers as the other strategies. The depth-first node model order and the strongly connected SC component cycle breaker perform best. This might be the case since both create longer layouts.

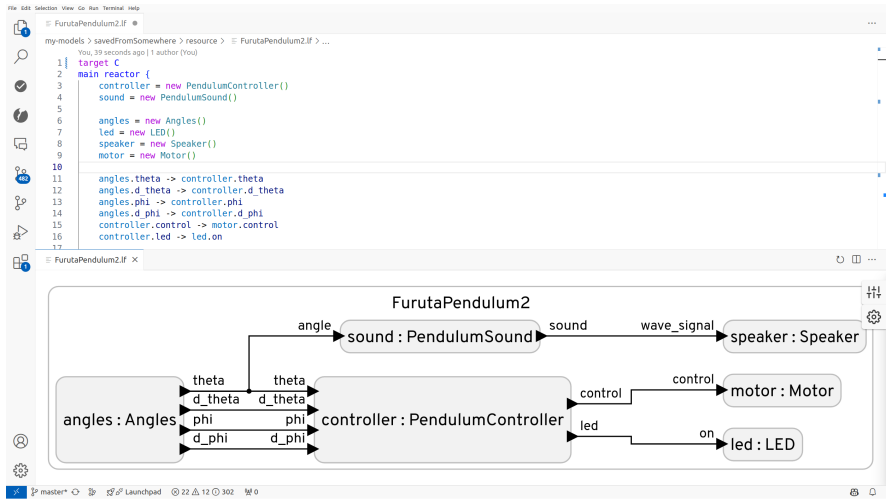
All cycle breakers only need a fraction of a millisecond to layout, but the greedy cycle breaker performs best in terms of execution time, as seen in Figure 5.48. The results show that the time does not really matter for the graph sizes Lingua Franca models typically have.

Figure 5.49 shows that the cycle breaking strategy does not alter the overall shape of a layout since each plot looks roughly the same. The actual difference in aspect ratio is not significant. However, note that the actual aspect ratio with a mean over two does not fit the editor and diagram setup described in Figure 2.1 on page 8. Lingua Franca developers would hence gain a better scale measure if they would move the diagram to the bottom, as seen in Figure 5.50. Here, the aspect ratio of the diagram window matched the actual aspect ratio of the layout.

A short qualitative comparison additionally showed that all approaches but the model order look-ahead cycle breaker produced suitable results. Additionally, the greedy cycle breaker often failed to express intention. The depth-first node model order and the strongly connected component model order cycle breaker produced good results.

However, further qualitative evaluation regarding developer intention, which I present in Section 5.9.6 and Section 5.9.7, must be done to show which strategy produces desirable layouts for Lingua Franca. Moreover, it needs to be evaluated whether large Lingua Franca models such as the

## 5. Model Order in Layered Layout



**Figure 5.50.** Lingua Franca developers tend to position their diagram below the textual editor such that the diagram window matches the shape of a Lingua Franca diagram better, as reported by experts and verified on existing Lingua Franca models.

cdn\_cache\_demo model by Magnition<sup>6</sup> can still use the strongly connected component cycle breaking without threatening reactivity. The strongly connected component node type cycle breaker, which reverses the edges to the reactor with the smallest model order, did not have a significant advantage over the SC strongly connected component cycle breaker. Hence, I only evaluated the SC approach in the following, since it allows more control and increased stability compared to the SN approach.

### 5.9.4 Study 3 continued: Quantitative Evaluation of Lingua Franca Crossing Minimization

Continuing the quantitative evaluation of Lingua Franca, which was already presented as part of the master thesis of Riepe [Rie24], I illustrate the effect

<sup>6</sup><https://www.magnition.io/>

## 5.9. Evaluating the Layered Algorithm

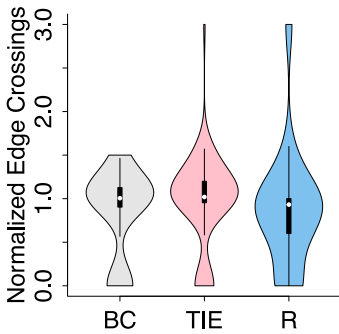
of the following model order strategies on edge crossings for the Lingua Franca models described in the scope of Study 3.

This part of Study 3 only focuses on the depth-first, the greedy, and the model order look ahead cycle breakers, since 97.1% of depth-first cycle breaking results equal the breadth-first and the strongly connected component model order cycle breaker results. Moreover, 93.4% of the greedy edge reversals equal the strongly connected component node type cycle breaking results.

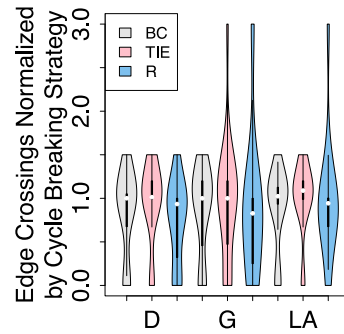
As the first strategy, I use the model order as a tie-breaker approach (TIE) presented in [DH22] using prefer-edges crossing minimization pre-sorting (see Section 5.5.2) and a node and port model order violation weight of 0.001 (see Section 5.5.4). The second strategy (R) utilizes model order only for reactions and constrains the barycenter heuristic to not reorder reactions, since this is desired by Lingua Franca developers. Hence, this approach only counts node model order violations for reactions and does not count edge order violations. The third strategy (BC) constrains the nodes by group model order. Here, nodes of the same model order group cannot change their relative order given by the model order. Hence, BC is the strictest strategy evaluated, which also adds the most stability to the drawings and gives developers more control than the others.

Figure 5.51 shows the edge crossings of the BC, TIE, and R model order crossing minimization configurations while using the greedy model order cycle breaker. Here, a value below one means that a strategy performed better than its alternative. I.e., a strategy with a value below one has fewer edge crossings than the others. Surprisingly R, which constrains the reaction order, achieves on average the best results. However, BC, which constrains the order of all model order groups and not only the reactions, is not much worse. A p-value of 0.0018 only indicates statistical differences between R and TIE, which were confirmed by a Wilcoxon test with a p-value of 0.0004. Using different cycle breaking strategies, as shown in Figure 5.52, shows that the strategies perform similar using different cycle breaking approaches since this boils down to using more models for the evaluation of crossing minimization. Moreover, this confirms that potential differences in cycle breaking, e.g., fewer layers, might influence the result in favor of the R strategy.

## 5. Model Order in Layered Layout



**Figure 5.51.** Study 3: Normalized edge crossings using the depth-first node order cycle breaker [Rie24].



**Figure 5.52.** Study 3: Edge crossings normalized by edge crossings in the respective cycle breaking category [Rie24].

The actual performance in terms of edge crossings only shows a significant difference between using the tie-breaking solution (TIE) and only constraining the reactions (R). Note that this study did not evaluate the often more important user intention and stability constraints. Hence, this is no indication that the R strategy is a suitable strategy for Lingua Franca. This just puts the three different approaches into perspective and allows estimating whether an approach potentially threatens readability. Moreover, it is known that BC constrains the graph the most and hence creates the most stable layouts followed by R and TIE. After this study, it is, however, unclear whether increasing stability based on model order is more desirable for Lingua Franca, which will be further evaluated in Section 5.9.6 and Section 5.9.7.

### 5.9.5 Study 4: Model Order Metric and its Effect on Edge Crossings for SCCharts

The following study illustrates the effect of model order crossing minimization pre-sorting (see Section 5.5.2) and weighted model order crossing minimization (see Section 5.5.4), which was already published in [DH22].

## 5.9. Evaluating the Layered Algorithm

The aim of this study is to show that crossing minimization pre-sorting has no effect on the number of edge crossings when used as a tie-breaker. Moreover, I investigated how the pre-sorting step influences readability and stability. Additionally, I investigated to what extent weighted model order crossing minimization would influence the edge crossings if weighted model order crossing minimization influences the selection of the “best” layout during crossing minimization.

I quantitatively evaluated 54 selected SCChart models that were “interesting” in a sense that they have at least three nodes and a non-trivial number of edges such that there must be at least two edges that could potentially cross each other. The result can be seen in Table 5.4 with the model order configuration encoding in Table 5.3. Table 5.3 shows that unordered layout configuration use a “U,” configurations with nodes-and-edges pre-sorting use a “N,” and prefer-edges pre-sorting uses an “E.” Additionally, the different node order violation weights  $w_n$  and port order violation weights are shown in subscript. E. g.,  $N_{0.1,2}$  is the layout configuration that uses nodes-and-edges pre-sorting with a port order violation weight of 0.1, meaning that port order violations are counted as  $\frac{1}{10}$  of an edge crossing, and a node order violation weight of 2, meaning that node order violations are twice as important as edge crossings.

As seen in Table 5.4,  $U_N$  and  $U_E$  are the configurations without model order crossing minimization pre-sorting and hence serve as a baseline for the ordering violations and crossings for the nodes-and-edges and prefer-edges crossing minimization pre-sorting approach. Because of the random permutation of the free nodes during crossing minimization (see Section 5.5.4), the edge crossings slightly increase even if there is no node or port order violation metric in use, or the node or port order weighting is too small to take effect. The number of fully ordered drawings slightly increases after adding node and port order violations as a tie-breaker. When comparing N and  $N_{0.001,0.001}$  and E and  $E_{0.001,0.001}$ , the latter has one fully ordered drawing less, while the number of edge crossings remains nearly the same<sup>7</sup>. Hence, model order pre-sorting is the main influence

---

<sup>7</sup>Here, the number of edge crossings is different since the model order pre-sorting results in different initial layer permutations. Hence, the randomly different permutation of the free nodes may result in a randomly different local minimum and hence a different number of

## 5. Model Order in Layered Layout

**Table 5.3.** Overview and encoding of the evaluated algorithmic alternatives. U (unordered), N (nodes-and-edges), E (prefer-edges). Columns differ in whether vertices or edges are prioritized, rows differ in the weights assigned to node order violations  $w_n$  and port order violations  $w_p$  relative to edge crossings, which carry a weight of 1.

$w_p$	$w_n$	U	nodes-and-edges	prefer-edges
0	0	$U_V, U_E$	N	E
$> 0$	0		$N_{w_p,0}$	$E_{w_p,0}$
0	$> 0$		$N_{0,w_n}$	$V_{0,w_n}$
$> 0$	$> 0$		$N_{w_p,w_n}$	$E_{w_p,w_n}$

on SCChart layouts in terms of order. Utilizing weighted model order crossing minimization as a secondary metric might only be useful for a comparatively small number for which the initial pre-sorting creates a local minimum.

The results of a manual comparison of the different approaches to categorize the concrete effects on the layout of model order pre-sorting using the nodes-and-edges approach with the  $U_N$  approach without pre-sorting can be seen in Figure 5.53. Of all 54 SCCharts models, 31 were *consistent models* such that node and edge model order were not in conflict. For 20 of these 31 SCCharts, the ordering clearly improved. Without using model order, backward edges have a tendency to be routed above the graph using the strategies. Hence, for the consistent models, I did not count models where only the position of backward edges changed as an ordering improvement. I argue that this is reasonable since this change might improve consistency and stability of a layout but is still minor compared to a proper order improvement. Of the 23 remaining and *conflicting models*, i. e., node and edge model order were conflicting, five models only differed in the routing of the backward edges and fifteen clearly improved the ordering compared to the approach without pre-sorting. For the remaining three models, the ordering improvement by model order was not that clear.

To summarize, model order pre-sorting does not lead to significantly more edge crossings during crossing minimization. Hence, model order pre-sorting does not threaten readability. At the same time model order pre-

---

edge crossings. Using more layout runs for crossing minimization may prevent this.

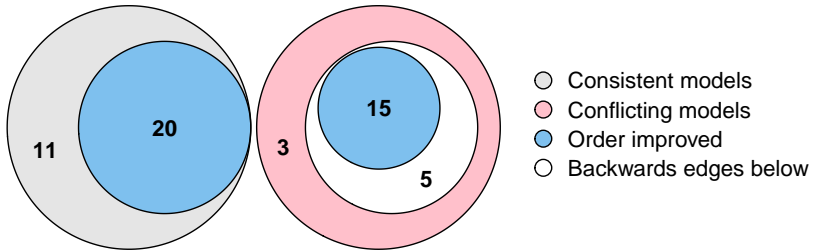


## 5.9. Evaluating the Layered Algorithm

**Table 5.4.** Edge crossings and graph order violations for 54 non-trivial SCChart models. I omitted the rows for  $N_{w_p,0}$  and  $E_{w_p,0}$  since they remained unchanged with varying port order violation weights  $w_p$ .

	Node order violations	Port order violations	Fully ordered drawings	Edge crossings
$U_N$	250	695	1	26
N	143	91	23	32
$N_{0,001,0}$	143	91	23	32
...	...	...	...	...
$N_{10,0}$	143	91	23	32
$N_{0,0,001}$	91	173	24	28
$N_{0,0,01}$	91	173	24	28
$N_{0,0,1}$	91	173	24	28
$N_{0,0,5}$	44	172	24	28
$N_{0,1}$	81	184	25	39
$N_{0,10}$	64	159	26	82
$N_{0,001,0,001}$	122	96	24	28
$N_{0,01,0,01}$	122	96	24	28
$N_{0,1,0,1}$	122	96	24	28
$N_{0,5,0,5}$	95	78	26	36
$N_{1,1}$	102	48	29	93
$N_{10,10}$	129	12	29	149
$U_E$	45	695	1	26
E	27	91	31	32
$E_{0,001,0}$	27	91	31	32
...	...	...	...	...
$E_{10,0}$	27	91	31	32
$E_{0,0,001}$	14	93	32	28
$E_{0,0,01}$	14	93	32	28
$E_{0,0,1}$	14	93	32	28
$E_{0,0,5}$	10	88	32	28
$E_{0,1}$	10	101	33	32
$E_{0,10}$	12	101	33	36
$E_{0,001,0,001}$	14	92	32	28
$E_{0,01,0,01}$	14	92	32	28
$E_{0,1,0,1}$	14	92	32	28
$E_{0,5,0,5}$	12	78	33	34
$E_{1,1}$	13	52	39	46
$E_{10,10}$	32	20	39	92

## 5. Model Order in Layered Layout



**Figure 5.53.** Changes of the 54 evaluated models

sorting improves the ordering of over half of the evaluated SCCharts models. If node and edge model order are consistent, all models at least have a better backward edge routing. Hence, these models are more stable and more consistent. Moreover, model order improves the secondary notation while enabling developers to control the layout by model order. Therefore, modeling languages should employ model order pre-sorting and should also consider adding model order violations as a secondary criterion to edge crossings to handle cases for which model order pre-sorting results in a local minimum.

### 5.9.6 Study 5: Model Order for Lingua Franca by Experts

Since the studies in Section 5.9.3 and Section 5.9.4 only measured the quantitative effect of different layout strategies on backward edges, area, aspect ratio, and edge crossings, I want to focus on qualitative evaluation of Lingua Franca in the following. These results were partly published in [DH24b; DH24c].

This study should capture the desired placement and ordering of different semantic elements for Lingua Franca. It not really a formal study but rather a collection of feedback, formal, and informal questionnaires of Lingua Franca developers, users, and students working with Lingua Franca as part of their coursework, which I want to present in a structured way. Moreover, I include the qualitative evaluation collected by Riepe as part of his master thesis [Rie24], feedback from Magnition, which employs

## 5.9. Evaluating the Layered Algorithm

Lingua Franca for one of their projects, and feedback during Lingua Franca research and developer meetings.

Riepe first collected the following feedback during Lingua Franca meetings. However, note that we encountered that Lingua Franca developers typically have trouble to articulate what they like or dislike, since the possibilities for layout options are vast and their dependencies are unclear. Moreover, only two Lingua Franca developers and three developers from Magnition gave feedback on that occasion, which possibly threatens the validity of these preliminary results.

At first, Magnition developers stated that reducing control over the model is desirable if it would result in fewer edge crossings or otherwise benefit the layout. After experimenting and showing different results, they argued that stability might be more important especially for larger models and that edge crossings can be reduced by omitting edges or made less critical by edge highlighting, as seen in Figure 5.35 on page 149.

Since feedback collected by Riepe was not very concrete, I further investigated what might be important for Lingua Franca developers during various meetings, resulting in the following list of constraints:

1. Edge-crossings should be reduced.
2. Stability is important for large hierarchical models.
3. Drawings are generally too wide.
4. The ports of reactors define their interface. Hence, a layout algorithm should respect their order.
5. The reactor-instantiation-order should be used to control their ordering.
6. Reactions and actions should be drawn below reactors since reactors are more important.
7. Actions should be placed such that the flow indicates how they may be triggered.

Given these preliminary results, the group model order barycenter heuristic and strongly connected component model order cycle breaker

## 5. Model Order in Layered Layout

based on a total group model order promised to yield desired ordering of semantically different Lingua Franca elements.

Since I already encountered that Lingua Franca developers had trouble answering questions regarding diagrams by the work of Riepe, I created the following stimuli. Using 62 “interesting” Lingua Franca models—meaning they use different semantic elements or show a possibility to use model order to control the layout and have at least three nodes—with a layout size that allows three of them to be shown on screen, I generated layouts using three different strategies.

*Strategy 1, the reaction only strategy (R)* This strategy is presented in Section 5.9.4 and constrains the crossing minimization by the reaction order and uses the greedy model order cycle breaker (see Section 5.3.2).

*Strategy 2, the reactions and reactors (RR) strategy* This strategy assigns all reactors and reactions a model order. Hence, this strategy might control the flow of reactions and reactors by model order and their order during the crossing minimization using the model order barycenter heuristic (see Section 5.5.3). Moreover, RR also uses the greedy model order cycle breaker. However, since this strategy assigns reactors and reactions a model order and not only reactions, the cycle breaking results differ from R.

*Strategy 3, the group model order strategy (GO)* This strategy utilizes the total group model order such that the startup node should be before reactors, timers, reactions, modes, actions, and shutdown in this order in the strongly connected component model order cycle breaker (see Section 5.7.1) and in the group model order barycenter configuration (see Section 5.7.2).

Moreover, note that the goal of the different stimuli created by the three strategies was not to find the best solution for all of these 62 Lingua Franca models. Rather these stimuli should serve as a way to highlight possible ONO layouts and should help developers unfamiliar with automatic layout to express what they like or what they dislike.

I interviewed six Lingua Franca developers as long as they wanted by showing all three layout variants of a model on the same screen. I collected

## 5.9. Evaluating the Layered Algorithm

feedback by asking about their preferences and engaging in discussion about likes and dislikes and their reasons for them. Moreover, I asked follow-up questions to sketch potential implications for their likes and dislikes to show potential alternatives and discuss these too. This questionnaire revealed the following.

In a left-to-right layout, the vertical order, i. e., the order inside a layer, of different node types should be startup<sup>8</sup>, modal model, reactor, timer, reaction, action, dummy nodes, and lastly shutdown. This might serve as a suitable tie-breaking order. This leaves the option to enforce the order of reactions if that is desired. However, experts were inconclusive whether small elements, e. g., reactions or actions, should be below or above big elements, e. g., reactors.

The horizontal order, the flow of the model, is not that simple. Experts stated that actions should be after reactions if this does not disturb the flow of the model. However, in the loadbalancer example in Figure 5.56, Figure 5.56c produces one crossing based on the reaction order constraints while Figure 5.56a has no edge crossings as the result of a different cycle breaking decision, which seems more desirable.

In reactor-only networks, the reactor model order should determine the flow. The instantiation order of reactors should be used to identify the first reactor to draw at the leftmost position. Based on this the reactor order should be used as a tie-breaker with the strongly connected component model order approach if the models are as small. Otherwise, one may use the depth-first node group model order cycle breaker if the models get large. The order of other elements, e. g., reactions, should only determine the flow as a secondary tie-breaker. Additionally, there should be a more controlling mode that fully controls the flow based on the reactor order. Generally, however, the standard layout strategy should not introduce edge crossings or unnecessary backward edges by constraining the order.

Additionally, Lingua Franca developers think that the vertical order of reactions should be respected since their naming and numbering suggests an ordering. However, I suspect that if one would label reactions with a real name instead of a number, this might be perceived differently. Lingua

---

<sup>8</sup>Potentially one should also consider startup triggers that are inside reactors to be relevant. If this should be the case, a reactor with startup should be before a simple startup.

## 5. Model Order in Layered Layout

Franca developers further reported that the order of actions, reactors, ports, and other elements becomes important in models where the name suggests an ordering, e. g., input1 and input2. Moreover, reactors named, e. g., increase and decrease also suggest an ordering. Hence, considering the naming of graph elements to control their constraints is an open question, as described in Section 10.4.9.

If separate connected components should be visualized using component packing (see Chapter 6), they should be ordered as follows. Components with a startup trigger or components with a reactor that has a startup trigger should be at the top. They should be followed by large elements, e. g., reactors, if the startup element is a reactor. Otherwise, all small elements, e. g., actions and reactions, should follow with the large elements after them. Here, Lingua Franca developers want small elements to be next to other small elements and large elements to be next to other large elements such that similar elements are grouped.

For completeness, I also want to mention that developers also want the layout to visualize the order of enclaves as flow or vertical order<sup>9</sup>.

Developers also want actions to be placed after their enabler, e. g., a reaction, reactor, or modal model, in terms of flow. Experts argued that this is the case since one wants to know what initially schedules an action to see when it might become “active.”

I presented the results of this questionnaire again in a Lingua Franca meeting that included Magnition developers and the results did not spark controversy. Hence, these layout constraints presented here should be considered by a model order configuration and be configurable. After the final presentation of potential layout constraints, one developer that was not questioned before argued that physical actions should always be on the left since physical actions are conceptually inputs.

### 5.9.7 Study 6: Model Order for Lingua Franca by Students

In addition to the questionnaire of Lingua Franca experts regarding ordering in the previous section, I interviewed seven students that utilized Lingua

---

<sup>9</sup>I refer readers interested in enclaves for Lingua Franca to Robledo et al. [RMJ+24].

## 5.9. Evaluating the Layered Algorithm

Franca as part of a course on embedded systems for further input regarding the layout of Lingua Franca with model order strategies.

Even though I previously argued against questioning non-experts, asking students might still yield usable results and they are easier to come by than Lingua Franca experts. However, students are often far less experienced and often do not use secondary notation in the same way as experts do [Pet95]. Hence, I expected that students perceive Lingua Franca diagrams differently than experts and that they may sometimes not understand the models shown to them. Nevertheless, I think that this evaluation confirms some insights about the use of model order for Lingua Franca.

As part of this course, these students did already create Lingua Franca programs as part of weekly exercises. Hence, they were familiar with Lingua Franca but no experts. During self-assessment, two of the students classified themselves as novice Lingua Franca developers, two between novice and intermediate, and three as intermediate Lingua Franca developers.

I presented the students with six different models using different algorithm configurations. For each model, I showed two to three randomly ordered layouts on the same screen such that the bottom left of the screen had space to show the description and abbreviated textual model. The order of the layouts on the screen and the order of the graphs was not permuted between participants since I expected the number of participants to be no more than ten.

Before asking questions, the students that already participated in a lecture about automatic layout and model order for Lingua Franca were reminded about different aesthetic criteria and what stability, secondary notation, and control mean for layout of models. After that, a selection of layouts for the same graph was presented to the participants with the option to ask questions about the models throughout the questionnaire. For each graph, I asked the participants the following questions.

*Which diagram do you like best based on readability?* This question aims to find aesthetic criteria for Lingua Franca diagrams. Moreover, this helps to differentiate between aesthetic criteria and secondary notation of a diagram. For this, the participants were only presented with the different drawings of the same model.

## 5. Model Order in Layered Layout

*Which diagram has the best secondary notation?* This question allows the participants to express, which layout has better ordering or grouping. In particular, I wanted to find out whether the preference for secondary notation was the same as the secondary notation preferred by Lingua Franca experts. In addition to the different drawings, participants saw a short description of the model, i. e., the comment in Figure 5.54a.

*Which diagrams match with the textual order?* Here, I tested which part of the model order corresponded to the desired ordering in a model and whether the students could recognize that. Hence, I chose some models with conflicting node and edge model order. To make the decision, students additionally received an abbreviated textual model, as seen in Figure 5.54a.

*Would you change the textual model to better understand it?* Here, the participants had time to consider changing the model to clarify its meaning. I asked this questions, since many experts in Section 5.9.6 saw flaws in the textual model, which should be corrected such that model order strategies produce a more desirable result. Hence, I wanted to test whether novice users could also recognize these flaws. Additionally, I aimed to received comments on the practical use of model order.

All the tested models were unknown to the students and were selected based on three criteria. First, there should be at least two different layouts created by five selected algorithm configurations.

**SCC + GMOF** Strongly Connected Component model order cycle breaker and **Group Model Order enForced** for crossing minimization using the group model order barycenter heuristic (see Section 5.5.3). This corresponds to the GO approach in Section 5.9.6.

**DF + GMOF** Depth-First cycle breaker and **Group Model Order enForced** for crossing minimization.

**DF + GMOT** Depth-First node model order cycle breaker and **Group Model Order Tie-breaking** crossing minimization using nodes-and-edges pre-sorting (see Section 5.5.2) and weighted model order crossing minimization (see Section 5.5.4).



## 5.9. Evaluating the Layered Algorithm

*GMO + MORO* **Greedy Model Order** cycle breaker with **Model Order** enforced for crossing minimization while only considering the model order of **Reactions Only**. This corresponds to the R approach in Section 5.9.4 and Section 5.9.6.

*GMO + MORR* **Greedy Model Order** cycle breaker with **Model Order** enforced for crossing minimization while only considering the model order of **Reactions and Reactors**. This corresponds to the RR approach in Section 5.9.4 and Section 5.9.6.

Of these five strategies, I selected two to three different layouts that were not clearly *ONO* to show to the students.

Second, experts must have already given feedback for the models such that I know the desired layout, or the models should be well established such that the desired order is public knowledge in the community.

Third, I used three different classes of models. I used two models that only consisted of reactions and actions such that the *Lingua Franca* semantics and coding conventions almost entirely constrain the textual order. I used two models consisting of two reactors and internal behavior to ask what should be considered in a model with a conflicting node and edge model order that is additionally inconsistent with the description of the model. The last two models were more complex and very connected to get feedback for model order in non-trivial cases.

**Model 1: TrainDoor** The TrainDoor model is a simple model only consisting of two reactors, as seen in Figure 5.54. The model consists of a MotionDetector, which checks whether the train has recently moved, and a DoorController, which opens the door on button press if the train has not recently moved.

The layout in Figure 5.54b is created by all strategies that use the reactor model order to determine the flow. Figure 5.54c is created by the reaction only (*GMO + MORO*) strategy. Both layouts differ in the alignment and order of the reactors. Figure 5.54b aligns the reactors and orders MotionDetector before DoorController, while Figure 5.54c aligns the ports creating a straight edge between them and by chance orders DoorController before MotionDetector since both are considered to have no model order. Experts

## 5. Model Order in Layered Layout

```

/**
 * This program emulates a train door controller.
 * It has two components: one that controls the
 * door and one that senses motion. When the
 * door controller receives a request to open
 * the door (a button press), it has to first
 * check whether the vehicle was recently in
 * motion. The request will be denied if motion
 * has been detected less than two seconds ago.
 */

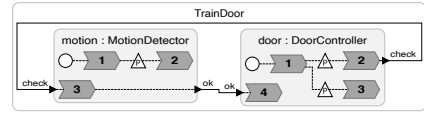
```

```

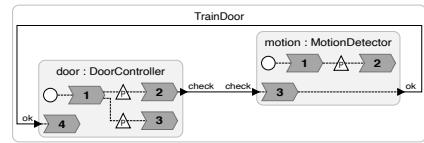
main reactor TrainDoor {
  motion = new MotionDetector()
  door = new DoorController()
  door.check -> motion.check
  motion.ok -> door.ok
}

```

(a) Documentation and abbreviated textual model of the TrainDoor main reactor



(b) SCC + GMOF, DF + GMOF, DF + GMOT, GMO + MORR



(c) GMO + MORO

**Figure 5.54.** The TrainDoor Lingua Franca model [Loh24] with textual main reactor and description.

argued that the reactor order should be used to control the desired layout of this model.

Of the seven participants, four preferred Figure 5.54b and three preferred Figure 5.54c in terms of readability. As a reason for this, three students argued that the reactor alignment (or non-alignment) drove their decision. Four students did not like the edge bend or liked the port alignment. One student argued that the hierarchical edge in Figure 5.54c drove their decision. For one student the compactness of Figure 5.54b was the main motivation for this decision.

In terms of secondary notation based on the diagram and the comment in Figure 5.54a, three changed their opinion from Figure 5.54b to Figure 5.54c. Moreover, two students changed their opinion from Figure 5.54c to Figure 5.54b. The remaining two students did not change their opinion and still preferred their original choice. In total, four participants preferred Figure 5.54b and three preferred Figure 5.54c. As reasons, four students answered that DoorController should be before MotionDetector based on the

## 5.9. Evaluating the Layered Algorithm

description. Two students argued that the MotionDetector should be before the DoorController since it enables the door. Here, the check label is on the left of the diagram, which is important since check happens before sending ok to the door.

Four students think that Figure 5.54b matches the textual model in Figure 5.54a while three students argued for Figure 5.54c. One student looked only at the reactor order and two students looked only at the edge order to determine the matching model. Since the node and the edge model order in the TrainDoor model are conflicting, they got different results. Four students stated that they considered both the edge and the node order. Three of them preferred Figure 5.54b and one Figure 5.54c. All three students that preferred the ordering based on edge order wanted to reorder the reactors in the textual model. One student that preferred the node order did not consider reordering the edges. Two other students that preferred the node order wanted to reorder the edges to match the reactor order. The last student that preferred the node order argued that he did not want to change anything but that the comment was more important than the order in the main reactor.

Overall students argued to use the model order to control the layout. However, some used the reactor order and some the edge order to exert control, which was possible since they were inconsistent. For this model, experts instead argued that the edge order was inconsistent and should be changed to conform with the reactor order, which should be primarily used to control the layout. Hence, I argue that it is better to fix the textual model rather than choosing a more elaborate layout algorithm, which could nevertheless create the desired layout.

**Model 2: SimpleChat** The SimpleChat model is again a simple model only consisting of two ChatHandler reactors that communicate with each other, as seen in Figure 5.55.

Figure 5.55b is created by the SCC + GMOF strategy, which, based on one expert comment in Section 5.9.6, orders the physical action to the left. Figure 5.55c is created by DF + GMOF, DF + GMOT, and GMO + MORR, which are all other strategies that determine the flow by the reactor model order. If the reactor model order does not determine the flow between

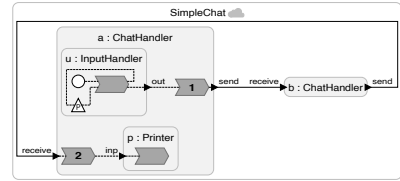
## 5. Model Order in Layered Layout

```

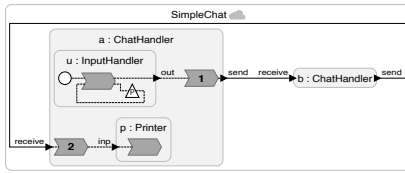
/**
 * This program is a simple chat
 * application for two users.
 */
...
federated reactor SimpleChat {
  a = new ChatHandler()
  b = new ChatHandler()
  b.send -> a.receive
  a.send -> b.receive
}

```

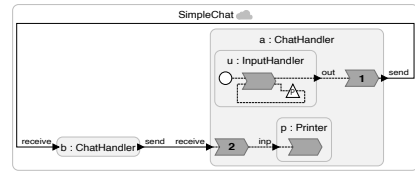
(a) Abbreviated textual model



(b) SCC + GMOF



(c) DF + GMOF, DF + GMOT, GMO + MORR



(d) GMO + MORO

**Figure 5.55.** The SimpleChat Lingua Franca model [JK24] with ChatHandler a expanded and ChatHandler b collapsed.

reactors, as it is the case for GMO + MORO, the algorithm may create Figure 5.55d. The layouts differ in the order of ChatHandler a and b as well in the placement of the physical action and the feedback edge in the InputHandler. I expected that students prefer reactor a before reactor b, which was also preferred by experts, but was unsure whether the students might also prefer the physical action at the start.

In terms of readability, no student preferred Figure 5.55b, one preferred Figure 5.55d, and six preferred Figure 5.55c while two of them were unsure about their decision. One participant could not decide at all. As a reason for their decision, three students mentioned that the whitespace above b and b being not at the top was important. Four student argued that the feedback edge as a result of the physical action placement in Figure 5.55b is ONO. One student argued that b before a is ONO, which is rather secondary notation

## 5.9. Evaluating the Layered Algorithm

than a readability criterion. Two students mentioned that the space above *b* in Figure 5.55d looks good and might help to focus on the communication in the model. One of them argued against Figure 5.55d based on the position of the receive port label.

In terms of secondary notation based on the diagram and a descriptive comment (see Figure 5.55a), no one changed their opinion. However, one participant argued that Figure 5.55b and Figure 5.55c both have good secondary notation. The reasons were their decision for readability for one participant, the lexicographic order of *a* and *b* for four participants, and *b* showing fewer details for one participant. Moreover, two participants that preferred Figure 5.55d, did so based on the positions of the send and receive labels.

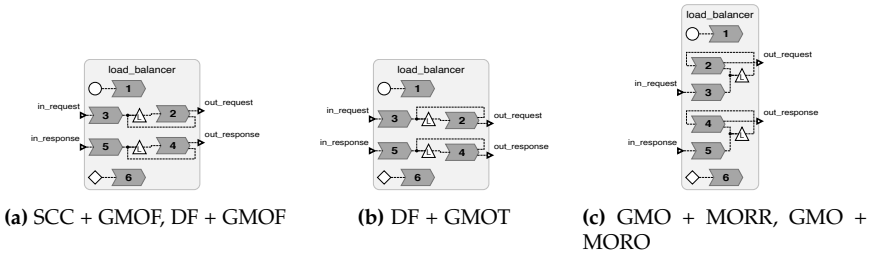
This shows that most participants prefer bigger elements on at the start of the layout and no whitespace in the top-left corner as seen in Section 5.9.6, and that the lexicographic order matters. It is, however, not clear whether the lexicographic order or the model order, which might also be lexicographic, is more important here, which I describe as one open question in Section 10.4.9.

One student argued that none of the layouts matched the textual order, since the reactor order and the edge order did again conflict, as it was also the case in Figure 5.54. One could not decide between Figure 5.55b and Figure 5.55c and another one could not decide between Figure 5.55c and Figure 5.55d. Two students preferred Figure 5.55c, one preferred Figure 5.55c, and one Figure 5.55d. As a reason, six quoted the model order of *a* and *b*, two the interface of the `InputHandler` reactor, two the edge order which created Figure 5.55d, and one that a physical action should be placed left as an input. Six students hence wanted to change the edge order. Additionally, two students wanted to collapse *a* and expand *b* such that the first element is smaller.

Here, students show a similar behavior as the *Lingua Franca* experts in that they prefer reactors to be ordered by their model order, but also value them being ordered lexicographically.

**Model 3: `load_balancer`** The `load_balancer` model consists only of reactions and actions, as seen in Figure 5.56. It models cache access between an `ll`

## 5. Model Order in Layered Layout



**Figure 5.56.** The load\_balancer Lingua Franca model [Ume24]. The abbreviated textual model can be seen in Section B.1.

and l2 cache<sup>10</sup>. Moreover, the depicted action-reaction network is a common pattern in Lingua Franca programs. Here, experts preferred Figure 5.56a for its clear flow that shows when actions are first enabled with feedback edges below, as detailed in Section 5.9.6.

Figure 5.56a is created by the SCC + GMOF and DF + GMOF strategy, which are the enforcing group model order crossing minimization strategies. Figure 5.56b is created by DF + GMOT with a setting to sort backward edges above normal edges. Without this setting, DF + GMOT should create a layout identical to Figure 5.56a. The strategies GMO + MORO and GMO + MORR that utilize the greedy model order cycle breaker create Figure 5.56c. This is the case since reaction 2 and 4 have two outgoing edges while the actions have two incoming edges such that their degree results in this cycle breaking. Overall, the layouts only differ in the backward edge selection and whether backward edges are below or above the rest of the drawing. I expected that students prefer a clear flow but also wanted to test one comment from an expert that argued that all enablers of an action and not only the first one should be before the action, as it is the case for Figure 5.56c.

In terms of readability, five students preferred Figure 5.56a, one preferred Figure 5.56b, and no one preferred Figure 5.56c. One participant could not decide between Figure 5.56a and Figure 5.56b. As a reason for this, two

<sup>10</sup>The load\_balancer is part of the larger cdn\_cache\_demo model by Magnition [https://github.com/MagnitionIO/LF\\_Collaboration/blob/main/complex-view-model/src/cdn\\_cache\\_demo.lf](https://github.com/MagnitionIO/LF_Collaboration/blob/main/complex-view-model/src/cdn_cache_demo.lf).

## 5.9. Evaluating the Layered Algorithm

students quoted compactness of the layout, and five mentioned the edge crossing in Figure 5.56c as a negative factor. Two students liked backward edges below forward edges and the below the rest of the layout and one liked them above forward edges. For two students a clear flow from left to right was essential, for one student the position of the physical edge was important, and one quoted that models in the lecture always had backward edges below.

In terms of secondary notation, three students did not understand the description and what the model should do. Two students could not decide between Figure 5.56a and Figure 5.56b, and two liked Figure 5.56a best. As a reason for their decision (if any), two students quoted the clear flow from left-to-right, and four mentioned the correct selection of the backward edge. Here, students have a similar preference as experts, which preferred Figure 5.56a.

Since the execution order of reactions constrains the possible reaction model order, there is no layout that really matches the textual description, which five participants figured out. One student argued that Figure 5.56c best matches the textual model and one argued that Figure 5.56b does. Their reasons were the vertical ordering of all reactions in Figure 5.56c, and the order of outgoing edges in Figure 5.56b, which matches the textual port model order. One student found the textual model more helpful than the layouts to understand the model, while one student argued the inverse. Four students did not want to change the textual model in Section B.1. However, three students would like to reorder the input and output ports, which were grouped together by the cache they belong to and not into inputs and outputs, as it is typically the case.

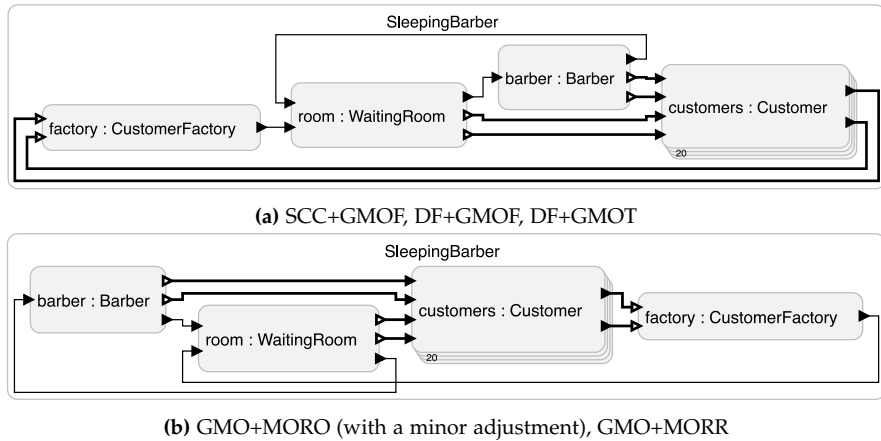
**Model 4: SleepingBarber** The SleepingBarber model is one of the more complex models used in this study, as seen in Figure 5.57. The modeled sleeping barber problem<sup>11</sup> is a classical synchronization problem by Dijkstra, which illustrates how processes might produce race conditions, which are partly prevented by Lingua Franca.

The layout of the Lingua Franca SleepingBarber model in Figure 5.57a is

---

<sup>11</sup>See [https://en.wikipedia.org/wiki/Sleeping\\_barber\\_problem](https://en.wikipedia.org/wiki/Sleeping_barber_problem)

## 5. Model Order in Layered Layout



**Figure 5.57.** The SleepingBarber Lingua Franca model [MCL+24]. See Section B.2 for the abbreviated textual model.

created by the SCC+GMOF, DF+GMOF, and DF+GMOT strategies, which utilize the reactor model order to determine the flow. Figure 5.57b is created by GMO + MORO and GMO + MORR. The strategies GMO + MORO and GMO + MORR focus more on the minimization of backward edges and hence create a flow based on that rather than on the textual description. The layouts mainly differ in the selection of the leftmost reactor. Figure 5.57a begins the model with the CustomerFactory reactor as specified in the textual model. Figure 5.57b begins by chance with the barber allowing for a barber-centric view on the model, which matches the description of the model in common literature. Experts deem both layouts to be sensible views on the model and argued that the reactor model order should be used to control the desired layout.

Hence, I expected that students would either prefer the barber-centric view or the more implementation driven view of the model based on what they focus on and that the different number of edge crossings and the selection of backward edges might make a difference for the participants.

In terms of readability, five students preferred Figure 5.57a and two preferred Figure 5.57b. One student argued that the “house-shape” of



## 5.9. Evaluating the Layered Algorithm

Figure 5.57a, i. e., it is slimmer at the top and wider at the bottom, was more pleasant. One student mentioned the length of backward edges, and four quoted the crossing backward edges as the criteria used for their decision. One student mentioned the placement of the barber at the start, and one mentioned the better flow from left-to-right in Figure 5.57a.

When presented with a barber-centric description, again five students argued that Figure 5.57a, which is not barber-centric, was better in terms of secondary notation compared to one student who argued for Figure 5.57b. One participant mentioned the number of edge crossings, which is not a criterion for secondary notation but rather readability, as his reason. Four participants mentioned the customer factory placed at the start as their reason. Two students thought that the barber-centric layout made sense but still preferred the one focused on the CustomerFactory. Moreover, the two students that preferred the barber-centric layout in Figure 5.57b did so because of the barber-centric description provided to them.

All students identified the CustomerFactory-centric solution depicted in Figure 5.57a as the one that matches the model order since it matches reactor model order, which starts with the CustomerFactory. The participants that previously liked the barber-centric solution did not want to change the textual reactor order to better fit the description. Three students argued that the textual model was not enough to understand the Lingua Franca model, since the number of edges was too large to comprehend.

Hence, most students confirmed the expert opinion that the reactor model order should control the flow of a model.

**Model 5: ACASXu2** The ACASXu2 model is the second more complex model, which is depicted in Figure 5.59. It describes two controllers that make sure that two aircraft do not collide in a shared airspace by modeling the own aircraft as well as an intruder aircraft.

Figure 5.59a is created by the SCC + GMOF creating a bad flow based on the reactor model order since the model order does not match the intention, as seen in Figure 5.58. Here, the Diff reactors should be instantiated after the controllers, as reported by experts. Hence, it is a carelessly developed textual model which should be changed to convey its actual meaning and I expected the students to come to the same conclusion. Figure 5.59b is

## 5. Model Order in Layered Layout

```
main reactor {  
  own = new Aircraft(...)  
  
  intruder = new Aircraft(...)  
  controller1 = new ACASController(period = 1 s)  
  diff1 = new Diff()  
  
  controller2 = new ACASController(period = 1 s)  
  diff2 = new Diff()  
  
  plot = new XYPlotter()  
}
```

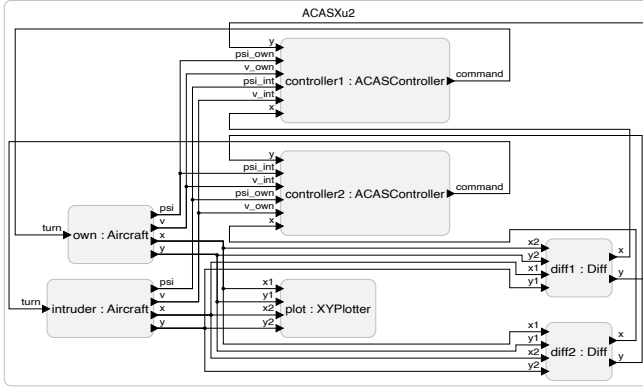
**Figure 5.58.** Abbreviated textual model of the ACASXu2 main reactor [CLP24] depicted in Figure 5.59.

created by GMO + MORR and creates a more desirable flow using the greedy model order cycle breaker since the model order has less influence using this strategy. However, this effect only occurs since the model itself has to be improved. Hence, this strategy also hides the flaws of the textual model. Figure 5.59c is created by GMO + MORO and only differs from the previous layout by the order of the reactors own and intruder. Hence, I expected that students will not like it if they cared about the vertical order of reactors. DF + GMOF and DF + GMOT were not shown to the students. Both depth-first strategies produced very bad results, as it is often the case in very connected models. Since nearly every node is connected with the others all nodes might be in different layers, which completely hides the symmetry of the model and is a common problem in depth-first cycle breakers, as detailed in Section 5.3.3.

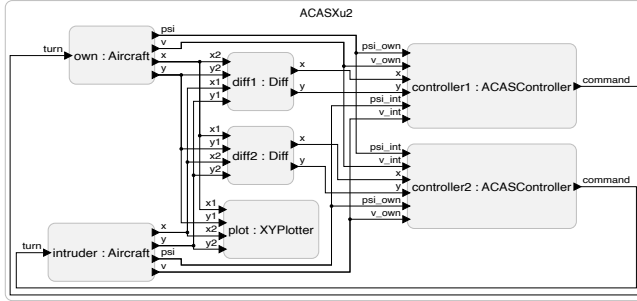
Experts rated Figure 5.59b as the best solution but argued that the model order in Figure 5.58 is bad and should be corrected. I expected that students would also see this problem once they see the textual model and that they would otherwise prefer Figure 5.59b.

Based on readability, one student rated Figure 5.59c best since the intruder reactor is bigger than the own reactor and should hence be at the top. Two students could not decide whether Figure 5.59b or Figure 5.59c were better. Moreover, two students preferred Figure 5.59b. The last two students

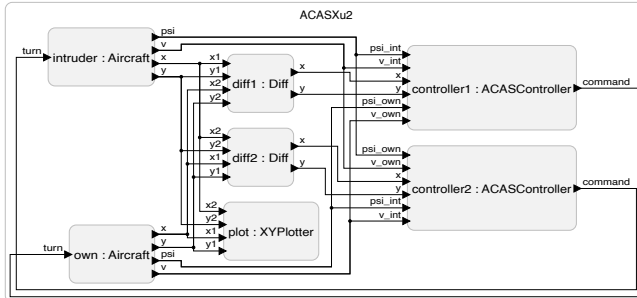
## 5.9. Evaluating the Layered Algorithm



(a) SCC + GMOF



(b) GMO + MORR



(c) GMO + MORO

Figure 5.59. The ACASXu2 Lingua Franca model [CLP24].

## 5. Model Order in Layered Layout

surprisingly preferred Figure 5.59a since the other drawings had confusing edge crossings between the Diff reactors and the ACASController reactors. Other reasons were the confusing feedback edges in Figure 5.59a for five students, the clutter between the aircraft and the controllers in Figure 5.59a together with the long edge to the Diff reactors for three students. Moreover, two students disliked the crossing feedback edges in Figure 5.59c. The same two students also preferred the position of the own reactor at the top.

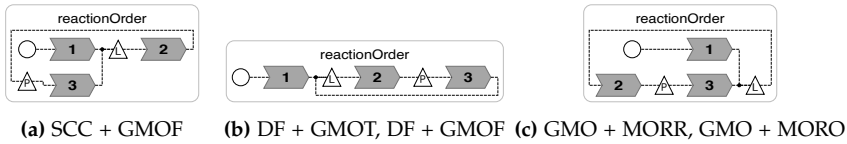
When presented with the information that the model showed two aircraft that avoid each other in the same airspace, all but one student preferred the layout in Figure 5.59b based on the placement of the own reactor and the flow. Only one participant argued that Figure 5.59a looked better than Figure 5.59b.

The two students that rated Figure 5.59a best based on readability argued that none of the layout matches the abbreviated model in Figure 5.58, which is wrong. The flow in Figure 5.59a matches the textual order. Three of the other participants argued that Figure 5.59b fits the textual model and quoted the fitting node order as their reason, which is also wrong. The order of own and intruder aircraft fit the node order in the layout but the order of the Diff reactors and the ACASController reactors does not. The remaining two participants correctly identified Figure 5.59a as the model fitting the textual model based on the abbreviated main reactor in Figure 5.58. Five participants argued that they would like to change the textual model to help comprehension by reordering the reactors such that the controllers and diffs would be interchanged. One, however, argued that the flow in Figure 5.59a is the correct one and the reactors should be grouped such that own, controller1, and diff1 should be next to each other. The last one argued that the edges, which were omitted in the abbreviated model shown to the participants, might need reordering.

Overall, the students again mostly want to control the flow by the reactor order. However, the model also shows that students are often still novices in a language and may give wrong answers or do not use secondary notation correctly compared to the experts interviewed in Section 5.9.6.

**Model 6: reactionOrder** The reactionOrder model depicted in Figure 5.60 is a model only consisting of reactions and actions, and again shows a

## 5.9. Evaluating the Layered Algorithm



**Figure 5.60.** The reactionOrder Lingua Franca model. See Section B.3 for the textual model.

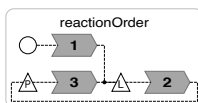
common pattern occurring in Lingua Franca models.

Figure 5.60a is created by the SCC + GMOF strategy. Here, the algorithm places the physical action in the first layer using the total group model order, as suggested by one expert in Section 5.9.6. Figure 5.60b is created by the depth-first cycle breaking strategies DF + GMOT and DF + GMOF. The greedy model order cycle breaking strategies GMO + MORO and GMO + MORR instead reverse the edge from the logical action to reaction 2, as seen Figure 5.60c. The layouts only differ in the selection of the backward edge and the route of the backward edge. Figure 5.60a and Figure 5.60c route the backward edge above and around the rest of the graph, while Figure 5.60b routes it below. I expected that students prefer a clear flow, as shown in Figure 5.60b, which is also preferred by experts. Moreover, I wanted to see whether the students also perceive the physical action as an input and prefer Figure 5.60a.

Based on readability, all participants rated Figure 5.60b highest as expected. As their reasons, five students quoted the flow, five a clear loop such that the startup node is not enclosed by the backward edge, and one student the backward edge routed below the drawing. Here, no one liked the physical action being at the start, which may, however, be the case since most disliked the routing of the feedback edge around the startup node. Hence, their evaluation might be different if they would have been presented the layout in Figure 5.61 instead.

Based on perceived secondary notation without a description, six participants still preferred Figure 5.60b because of the clear flow. One participant preferred the backward edge in Figure 5.60c instead. I, however, expected that most could not decide since this model has no given description and no fixed meaning.

## 5. Model Order in Layered Layout



**Figure 5.61.** Variant of SCC + GMOF with backward edge below.

All participants rated Figure 5.60b as the model that matches the textual description most since the ordering of reactions matches the flow. Two participants would have reordered the logical and the physical action such that the logical action would be declared first in the textual model.

### Summary of Study 6

To summarize, this study showed that students can recognize if the reactor order and reactor connection order does not match the layout if models are small. Moreover, the students mainly focus on flow and edge crossings in their decision-making based on readability and also are prone to make mistakes.

Even with the small selection of models, no clearly optimal layout configuration could be found. SCC + GMOF, however, worked for most models and student mostly recognized how they could change the reactor order to control the layout if they were unhappy with it. For reaction-action networks, such as Figure 5.56 or Figure 5.60, SCC + GMOF does not provide full control. Here, the fixed reaction order and separate model order groups for actions and reactions make control by model order impossible. Hence, reaction-action networks might benefit from a depth-first cycle breaker. Moreover, it was important to the students to route feedback edges below the drawing if possible, as seen in Figure 5.61 compared to Figure 5.60a.

Lastly, students did not share the perception that physical actions might be an input, as suggested by one Lingua Franca expert. Whether one should hence sort the physical actions before the reactions and reactors using a total group model order therefore requires additional research and potentially more studies involving Lingua Franca experts.

### 5.9.8 Study 7: Model Order for SCCharts in Practice

The previous studies aimed to qualitatively evaluate real world Lingua Franca models. Study 7 aims to evaluate how SCCharts models developed for real applications are used in industry by Scheidt & Bachmann System Technik GmbH (S&B). This study was partly published in [DH24b; DH24c] and consists of an interview with five SCCharts developers from S&B, who kindly provided me with 36 SCChart models and their time to ask questions about them.

I evaluated these 36 SCCharts using two different model order configurations for SCCharts. Strategy 1 is currently employed by S&B and has been the default for over a year. This strategy uses the strict model order cycle breaker (see Section 5.3.1), minimizes edge crossings using prefer-edges pre-sorting (see Section 5.5.2), and weighted model order crossing minimization (see Section 5.5.4) with node and port model order as a tie-breaker. Strategy 2 is the “full control” strategy (see Section 5.5.5). This strategy employs the same cycle breaker, but omits crossing minimization entirely by only doing pre-sorting with the prefer-edges strategy.

Before presenting the evaluation results, I want to present the structure of these SCCharts. The biggest regions in all models range from three to eleven states with a mean of 6.22 states per region and a median of six states, with a mean degree of 1.65 without self-transitions. Hence, most of the models have few nodes and are not very connected. The biggest model is an outlier and has in its biggest region 10 states and 43 edges without self-transitions and is regarded as large and very complex by S&B developers. Of these 36 models, 32 have a planar embedding. For 19 models, both strategies created identical layouts. This means that the model order matches the intention and that the optimal layout regarding readability for these 19 models matches the model order. Eleven additional models could be laid out the same way if one or two edges would be reordered. Hence, most models intuitively employ model order providing stability and control over secondary notation.

The reasons why Strategy 2 could not directly create the same drawings as Strategy 1 falls in three patterns:

- i) Backward edges that are drawn below forward edges, as seen in the

## 5. Model Order in Layered Layout

edge from Active to Interrupted in Figure 9.7 on page 259,

- ii) dummy nodes, which are nodes that determine the routes of edges, have no model order, similar to the problem depicted in Figure 5.28 on page 138, and
- iii) the biggest highly connected model is too complex to let humans manually reorder edges, as it is, e. g., the case in Figure 9.4b on page 241.

Using Strategy 1, the S&B developers did not notice anything out of the ordinary using the strict model order cycle breaker, even though it created dangling nodes in two different models, which were not seen as a problem by S&B developers. E. g., Figure 9.7 on page 259 has a dangling connector state in the top left of the drawing. This is a strong indication that strict model order cycle breaking is a good strategy for SCCharts even though it completely constrains the flow by model order. If it were not suitable, developers would have noticed the potentially irritating layouts the strict model order cycle breaker could potentially create.

The interview also revealed that S&B developers mainly want to improve the drawing based on edge crossings for better readability and do not like Strategy 2 that much, which focuses entirely on the model order. This may be the case since they often only *browse* the models but seldom *edit*. Moreover, edits are typically small changes to guards and effects of transitions. Hence, developers are already happy with the stability provided by Strategy 1 since there are no edges crossing hierarchy levels, which might compromise stability when expanding or collapsing regions in SCCharts, which is a problem for Lingua Franca models.

Moreover, S&B developers reported that transition guards are typically mutually exclusive such that the priority of edges does not matter. If this is the case, edges can be freely reordered if desired, which increases the degree of control model order can have on the layout. However, since this could not be verified by me since most guards include host code functions, I conservatively assume that the order of transitions cannot be freely changed.

Hence, Strategy 1 seems to be a good default strategy for SCCharts and possibly for other state machine dialects. Completely constraining the layout by model order as done by Strategy 2 should only be used to control the model when desiring a specific layout that requires more control.



# Model Order in Component Packing

The rectpacking (see Chapter 4) and the layered algorithm (see Chapter 5) consider the model order to layout Lingua Franca and SCCharts models. In this chapter, I propose that model order should also be considered for other layout algorithm such as the layout of *separate connected components*.

Since a layout algorithm may split a hierarchical graph into several subgraphs, separate connected components may exist inside the same parent node, as seen in Figure 6.1a. Here, the top-level graph inside the SleepingBarber reactor consists of four reactors: Barber, WaitingRoom, Customer, and CustomerFactory. The subgraph inside Customer consists of four separate connected components with connections to the outside. I.e., reactions 1, 2, 3, and 4 with their respective inputs and outputs. If the model order is not considered, an algorithm may not order the reaction-components inside Customer by their model order, which is always identical to the reaction numbering, and create the ONO layout in Figure 6.1a instead of the desired solution in Figure 6.1b. Moreover, not using model order as the component order means that an algorithm probably uses a more volatile ordering such as the component size, which might change when expanding or collapsing elements to show their inner behavior. Lastly, since these components are not connected, layering and crossing minimization techniques for layered layout presented in Chapter 5 create ONO layouts, as seen in Figure 6.2.

The first problem of using the layered approach for separate connected components can be seen in Figure 6.2a, which forces unrelated nodes on the same layer-grid and makes the spacing between nodes ONO. Without knowing about algorithm internals, it is not apparent why the spacing

## 6. Model Order in Component Packing

between the startup trigger, reaction 1, and the physical action is bigger than the spacing between the physical action and reaction 2. Additionally, Figure 6.2a is unnecessarily wide compared to desired solution Figure 6.2c. Although utilizing one-dimensional compaction strategies [Rüe18] may solve this issue by compacting the drawing after a layered layout, considering the model order for *model order component packing* might be the easier solution since this also reduces the size of the layered layout problems.

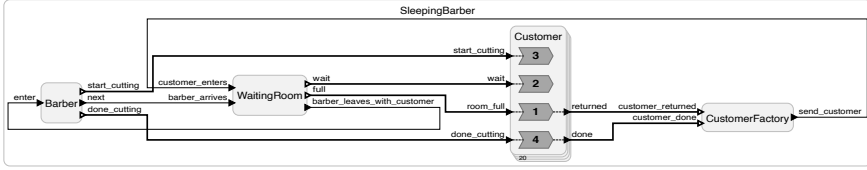
The second problem created by forcing separate connected components on the same layer grid is intertwined components, as seen in Figure 6.2b. Here, constraints between nodes or a local minimum during crossing minimization might create the displayed ordering. Parts of the model that are not connected become intertwined, which might let users believe that they are directly related and not separate components. This is also the case for the GameController in Figure 9.4a on page 241. Layouting these components separately solves both problems shown but requires considering model order to order and pack the components. Hence, model order should also apply to the layout of separate connected components and should be used in a *component packing* algorithm.

### 6.1 The Component Packing Problem

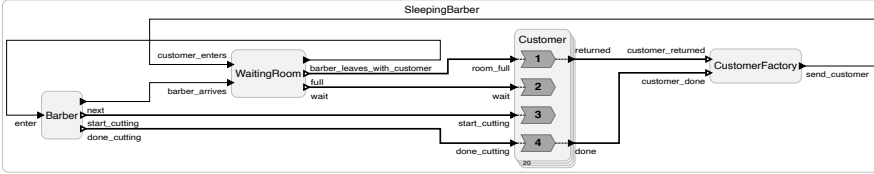
For Lingua Franca, the component packing problem is fairly simple. Lingua Franca has only separate connected components with EAST, WEST, or no connections. Hence, ordering them vertically yields the desired packing in Figure 6.1b or 6.2c. The more general *component packing problem* is defined as follows.

Given  $n$  components  $\{c_0, \dots, c_{n-1}\}$  defined by their bounding box with a width  $w_i$  and height  $h_i$ , one tries to optimize the scale measure (see Section 4.1.1) such that each component fits a desired aspect ratio. Additionally, node-node and edge-node overlaps should be prevented by reserving edge-routing space for each component based on the edge connections to the outside. The component packing problem resembles the region packing problem with additional constraints: Each rectangle may specify any of the sides NORTH, EAST, SOUTH, WEST to which it may have connections.

## 6.2. Component Packing by Schulze



(a) Wrong component order in Customer



(b) Right component order in Customer

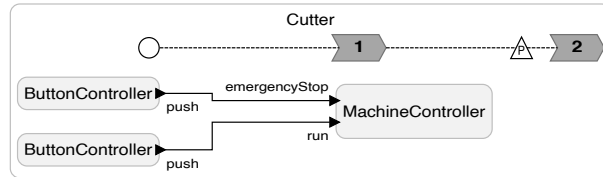
**Figure 6.1.** The Sleeping Barber Lingua Franca model [MCL+24] layouted without separate component model order.

## 6.2 Component Packing by Schulze

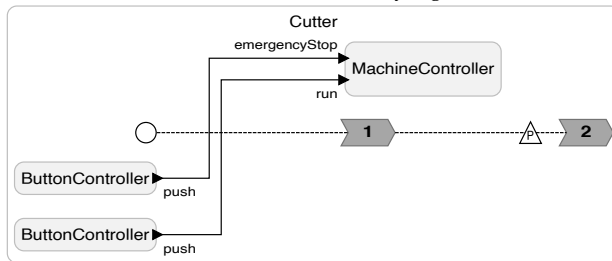
Schulze [Sch19b] describes an algorithm for packing separate connected components, which packs them based on their outgoing edges to external ports, as seen in Figure 6.3a.

Schulze considers the placement dependencies between different components and their outgoing connections to assign them to different *component groups* that have to be layouted diagonally under each other. To achieve this, Schulze suggests using the depicted  $3 \times 3$  grid, which is here filled with components with NORTH, EAST, SOUTH, WEST, NORTH-WEST, NORTH-EAST, SOUTH-EAST, SOUTH-EAST and no connections. The component group hence may consist of components that can be grouped together such that the outgoing connections do not cross each other. E. g., two NORTH-SOUTH-WEST-EAST components may not be in the same group. Instead, one wants to place the second NORTH-SOUTH-WEST-EAST diagonally below the first component in a new component group. However, one could add, e. g., a NORTH-WEST component to fill the upper-left space of a NORTH-SOUTH-WEST-EAST component.

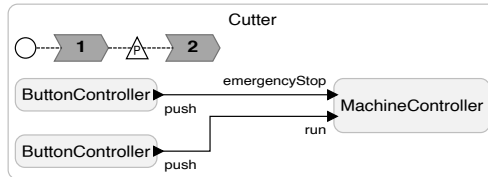
## 6. Model Order in Component Packing



(a) Forced on the same layer grid



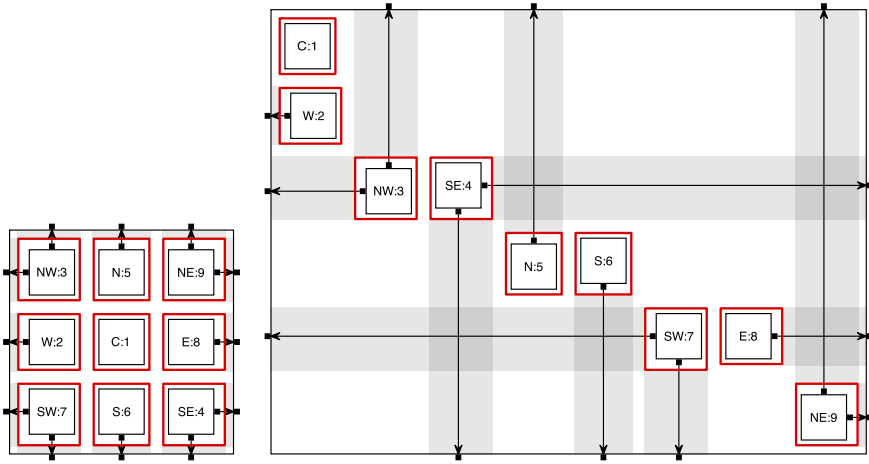
(b) Forced on the same layer grid creates a conflict



(c) Desired solution

**Figure 6.2.** The Cutter Lingua Franca model [Lee24b] layouted with and without considering separate connected components.

### 6.3. Model Order Component Packing



(a) The component packing algorithm [Sch19b] packs components in a component group on a  $3 \times 3$  grid.

(b) The component order C, W, NW, SE, N, S, SW, E, NE results in a higher scale measure and additional edge crossings using the model order component packing algorithm.

**Figure 6.3.** Component packing and model order component packing compared. Each component is marked in red with the edge routing space marked in gray.

## 6.3 Model Order Component Packing

Figure 6.3a shows that the component packing by Schulze does not respect the model order, as a trade-off to prevent edge crossings and increase the scale measure. However, as a result Schulze's component packing algorithm may cause the ordering problem depicted in Figure 6.1a, since the model order is disregarded.

Schulze's algorithm optimizes for readability such that the gray routing areas for edges and edge crossings are minimized. However, users have no direct control where a specific component might be placed. Additionally, adding a new connection to the outside might change the component packing and with it their order, possibly threatening stability and the mental map.

## 6. Model Order in Component Packing

---

### Algorithm 12: Model order component packing

---

**Input:** Ordered list of components  $cs$ , target width  $w$   
**Output:** Placed components  $cs$

```

1  currentRowLevel := 0
2  currentRowHeight :=  $c_0.height$ 
3  currentRowWidth :=  $c_0.width$ 
4  rowStartX := SOUTH  $\in c_0.side$  ?  $c_0.width$  : 0
5   $i := 1$ 
6  for (  $c \in cs \setminus c_0$  )
7      inNewRow := (EAST  $\in c_{i-1}.side$ )
8       $\vee$  (WEST  $\in c_i.side$ )
9       $\vee$  ( $currentRowWidth + c_i.width > w$ 
10      $\wedge \neg(rowStartX + c_i.width > w) \wedge \neg(NORTH \in c_i.side)$ )
11      $c_i.x = NORTH \in c_i.side$  ?  $currentRowWidth$  :  $rowStartX$ 
12     if ( inNewRow )
13         currentRowLevel += currentRowHeight
14         currentRowHeight :=  $c_i.height$ 
15          $c_i.y := currentRowLevel$ 
16     else
17          $c_i.y := currentRowLevel$ 
18         currentRowHeight :=  $\max(currentRowHeight, c_i.y + c_i.height)$ 
19     currentRowWidth :=  $c_i.x + c_i.width$ 
20     rowStartX := SOUTH  $\in c_i.side$  ?  $c_i.width$  :  $rowStartX$ 
21      $i := i + 1$ 

```

---

### 6.3. Model Order Component Packing

The *model order component packing* algorithm in Algorithm 12 solves the ordering issue at the cost of readability, as shown in Figure 6.3b. Similar to rectpacking, component packing has to determine a target width such that the packing yields a better aspect ratio. In contrast to rectpacking, model order component packing considers the external connections of a component. With a given target width, the model order component packing algorithm handles all components after each other and decides whether they are placed in the current row or in the next one.

A component without connections can just be placed at the next available position, as it is the case for the C component in Figure 6.4b.

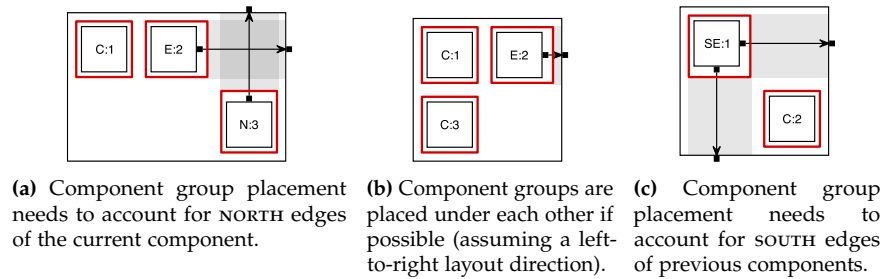
A component with a NORTH connection can only be placed such that it does not overlap with already placed components, as seen in Figure 6.4a. Here, N moves to the right such that its edge routing area does not overlap with already placed components. In Figure 6.3b, N, NW, and NE are also moved to the right to not cause edge-node crossings. Hence, such a NORTH-component needs to be able to ignore the target width to be placeable in all situations, as seen in line 10 in Algorithm 12.

A component with an EAST connection constrains the next component, as seen in line 7. The component following a component with an EAST connection must be placed in the next row, as seen in Figure 6.4b. Here, C is placed in the next row since E's edge routing space extends to the right and would otherwise overlap. This can also be seen in Figure 6.4a for E and N and in Figure 6.4c for SE and C. Note that for component packing by Schulze, an EAST-component does not necessary start a new component group. In component packing, it entirely depends on whether a component group is "full."

A component with a SOUTH connection increases rowStartX the first available x-coordinate inside a row. E. g., in Figure 6.4c C cannot be placed directly on the left edge of the parent node. This is also the case in Figure 6.3b for S, which constrains the row inset for all following rows. Here, a row inset rowStartX might again violate the target width of the drawing, as seen in line 10. Hence, in this case, one needs to disregard the target width to still allow valid placement of components, e. g., if only SOUTH-EAST components exist.

Finally, a component with a WEST connection always has to be placed

## 6. Model Order in Component Packing



**Figure 6.4.** Component placement needs to account for the connection edges.

in a new row. E. g., component W in Figure 6.3b is in a new row, as are NW and SW. This must be possible even if the target width is already reached.

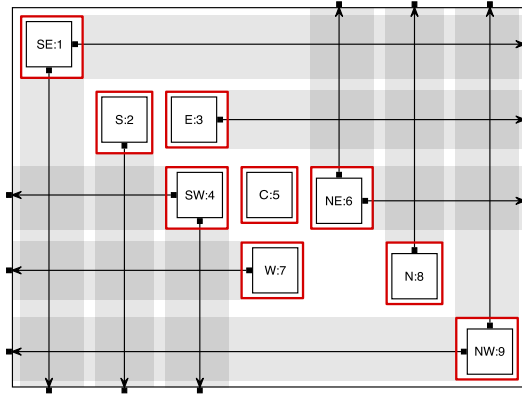
The resulting packing as a clear left-to-right reading direction given by the model order and will remain somewhat stable when introducing new connections to the outside by keeping the reading direction stable. However, a lot of space may be wasted since model order constrains the placement of components quite a lot. In this worst-case example, the packing by Schulze in Figure 6.3a needs nine space units (not considering spacing and padding and assuming components have the same size), while the model order component packing in Figure 6.3b needs 56 space units. Moreover, additional edge crossings may further decrease readability. Figure 6.5 illustrates that the model order component packing algorithm may create eighteen additional edge crossings per component group.

## 6.4 Summarizing Component Packing

To summarize, model order component packing might take more than six times the of the space required for the solution by Schulze to consider stability and the model order reading direction constraints, as seen in Figure 6.3. As result of all these model order constraints, the target width approximation for component packing is not very accurate. Additionally, model order component packing does not care about edge crossings such that additional edge crossings might be created by the model order reading



## 6.4. Summarizing Component Packing



**Figure 6.5.** Maximum number of edge crossings using model order component packing. This assumes the component order SE, S, E, SW, C, NE, W, N, NW given by the numbering.

direction constraints.

Future work on component packing should focus on a better width-approximation algorithm, as detailed in Section 10.4.5. Moreover, to improve possible ONO packings, one could also add a model order aware compaction algorithm, as presented in Section 10.4.6, that works with concrete edge routes rather than component bounds and edge directions. This might help to create space efficient component packings using model order.

However, for Lingua Franca the model order component packing is good enough and does not threaten readability, as seen in Figure 6.1. Since Lingua Franca has only `WEST` and `EAST` connections, the packing remains space efficient despite model order reading direction constraints. Moreover, it can only create additional edge crossings if the order of the connections to the outside is constrained, which is typically not the case for Lingua Franca.



## **Part II**

# **Interactivity**



# An Interactive Constraints Framework

Since the model order is only one-dimensional but a drawing typically has two-dimensions, model order cannot always express the necessary degree of control to create desired packings or layouts. Layout constraints, however, can be two- or more-dimensional and are typically utilized to increase the interactive control over a drawing [Dwy09].

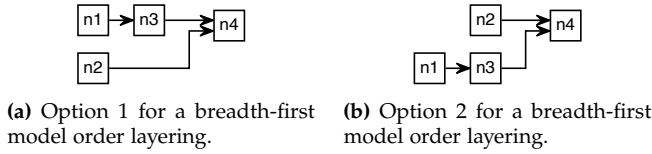
E.g., Figure 7.1 shows two possible layerings of a layered graph, which do not violate the model order layering constraints introduced in Section 5.4.2. I.e., there exists no node with a lower model order in a succeeding layer and no node with a higher model order in a preceding layer. However, a deterministic implementation of a breadth-first model order layerer, as proposed in Section 5.4.2, can either create Figure 7.1a or Figure 7.1b. At the same time, both layouts could be the desired layout. Therefore, model order cannot sufficiently control all graphs for all possible use-cases (*EVAL*), which could be solved by using layout constraints.

At the same time, model order and constraints are not alternatives but two tools that can be used together to create desirable drawings and to capture user intention to express good secondary notation, as detailed in Section 5.2.1.

On tool that let users interactively set layout constraints by *diagram interaction* is the Dunnart tool [DMW09]. Instead of utilizing the model order, diagram interaction directly captures the intention, e.g., based on grabbing edges and moving nodes.

Another tool that implements layout constraints is the interactive constraint framework proposed by Petzold et al. [PDS+23]. This work is based

## 7. An Interactive Constraints Framework



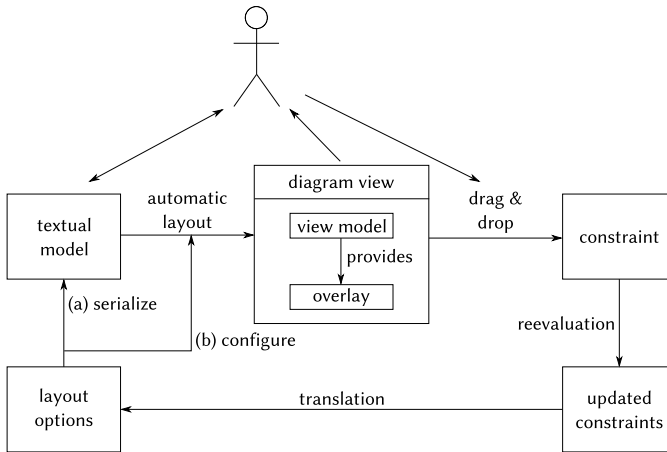
**Figure 7.1.** Even with the full control model order configuration, the user cannot enforce which option to use, both are equally compatible with breadth-first model order layering.

on two bachelor theses I advised [Pet19; Sch19a] (see Section A.2), which I generalized into a framework together with Petzold.

The underlying communication of the framework is depicted in Figure 7.2. Here, automatic layout creates a view model in a diagram view. A user might edit the textual model in an editor, which results in the corresponding diagram to be updated, as presented in Chapter 2 as the required editing paradigm for using model order. Additionally, a user might interact with the diagram view. Using drag-and-drop supported by a diagram overlay that visualizes constraints, user interaction creates abstract constraints, which requires a reevaluation of already existing constraints to prevent conflicts. Translating the abstract constraints into layout options that an algorithm understands enforces these constraints. Layout options are typically serialized (a) as part of the textual model to update the diagram. Alternatively, the layout constraints can configure the view model (b) to only update the diagram temporarily.

The proposed diagram interaction via drag-and-drop follows four principles to ensure that it actually matches the intention and preserves stability.

- (1) There should be no conflicts between constraints.
- (2) No unintended changes in node ordering and edge routing should occur.
- (3) Only elements that were interacted with get a new constraint.
- (4) Constraints of elements that were not interacted with should only be changed if these constraints violate Principles (1) or (2).

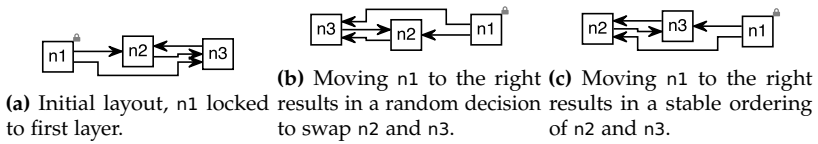


**Figure 7.2.** Interaction during interactive layout from user interaction to updating the diagram, as proposed by Petzold et al. [PDS+23].

Principle (1) is the most basic one. If two constraints are conflicting, Petzold proposed that one disregards one of them to resolve the conflict. The idea behind that is that the most recent action taken by the user has priority since it might be perceived more important by a user. E. g., moving a node to the coordinates another node is already constrained to should move the other node away. This might be handled differently by other interactive constraints frameworks. E. g., the Dunnart tool instead shows potential conflicts while trying to set constraints [DMW09].

Principle (2) ensures that the layout remains stable. If too many nodes or edges move simultaneously in unpredictable directions, the mental map is difficult to maintain. Therefore, layout creation strategies, i. e., layout strategies that do not record the different editing steps but only use a (textual) model as its input, that employ random decisions may threaten stability while using interactive layout. E. g., this may be the case for the greedy cycle breaker discussed in Section 5.3.2. This is the case since a constraint may change the underlying model such that a random decision might randomly decide differently, as seen in Figure 7.3. Here, a lock symbol

## 7. An Interactive Constraints Framework



**Figure 7.3.** Interactively setting constraints requires a stable layout. Layout constraints are visualized with a lock symbol.

visualizes that a node has a layout constraint. If the layout in Figure 7.3a, which constrains n1 to the first layer, is the initial layout, it is undesired that the order of n2 and n3 randomly changes if n1 is constrained to the last layer, as it is the case in Figure 7.3b. Instead, the order of n2 and n3 should remain stable, as seen in Figure 7.3c. The Dunnart tool [DMW09] solves this by considering layout adjustment [MEL+95] steps rather than laying out only based on the (textual) model file.

Principle (3) ensures that an interaction with the graph does not introduce unexpected constraints to the graph. Constraining a node to a specific position should not require constraints for all other nodes of a graph since their order might not be intentional. E.g., in Figure 7.3 one should not constrain n2 and n3 together with n1 to create the stable result in Figure 7.3c and prevent Figure 7.3b.

Lastly, Principle (4) ensures that an interaction only deletes or updates constraints if the current interaction requires this to prevent conflicting or invalid constraints. Additionally, Principle (4) makes sure that previously set constraints can be overridden by one interaction, which requires reevaluation of already existing constraints when introducing a constraint. E.g., instead of reevaluation one could require that the old constraint must be changed or deleted before introducing a potentially conflicting constraint, which adds unnecessary interaction steps and therefore complicates a constraint framework.

Using these principles, I illustrate the interactive constraint framework by Petzold on the example of the layered algorithm (see Chapter 5) and the rectpacking heuristic (see Section 4.4). Section 7.1 illustrates how setting absolute and relative constraints for layered layouts benefits from using model order. Section 7.2 discusses what constraints are necessary to increase



the control the rectpacking heuristic has over a packing in addition to using model order. Finally, Section 7.3 shortly discusses setting interactive constraints for different algorithms and generalizes how model order can be used together with constraints. Specifically, I generalize how constraints can improve the control model order might have and how model order adds stability to layout constraint frameworks (*EVAL*).

## 7.1 Interactive Layered Layout

In a layered layout one may want to specify a concrete layer of a node or a concrete position of a node in a layer, which are absolute constraints. Alternatively, users may want to specify layer or position relative to other nodes. E. g., “n1 is below n2” is a relative constraint, while “n2 is the first node in the second layer” is an absolute constraint. Additionally, a user may want to specify alignment, which I will omit here, since the respective phases that handle alignment, i. e., node placement and edge routing, have no suitable model order strategy, as detailed in Section 5.6 and Section 5.1.6.

There are already a lot of implementations for such relative or absolute layout constraints for the layered algorithm [BP90; Wad01; MSW19]. Hence, I only present the constraint reevaluation proposed by Petzold and its relationship with model order by example.

Figure 7.4 illustrates the reevaluation steps for setting an absolute constraint<sup>1</sup>. Figure 7.4a shows the initial layout. Additionally, I visualized the layer and position grid, this example works with. Here, n1 has no constraints, while we constrain n2 to the second position in its layer, which is visualized by a lock symbol with a vertical double-arrow. Additionally, I added “PC 1” to illustrate that the position the node should have in its layer<sup>2</sup>. Node n3 is constrained to the first position in the second layer. When picking up n1 and moving it to the first position in the second layer, as seen in Figure 7.4b, the diagram overlay displays the layers and potential positions to drop the node. Additionally, the overlay shows the original

---

<sup>1</sup>I omitted edges for clarity since they do not change the concept behind the constraint reevaluation and model order.

<sup>2</sup>Note that I begin counting positions and layers with 0 rather than 1.

## 7. An Interactive Constraints Framework

position of the node as a shadow left behind to ensure stability during the interaction. Since  $n_1$  is moved to the first position in the second layer,  $n_1$  gets the desired constraint, while the constraints of  $n_2$  and  $n_3$  update based on the interaction, as seen in Figure 7.4c. Node  $n_2$  updates its constraint “being the second node in its layer” to “being the first node in its layer”. This is the case since the position constraint of  $n_2$  would otherwise become invalid. Moreover, this prevents that adding a new node to the first layer may “activate” a potentially dormant constraint of  $n_2$ . Finally, node  $n_3$  is now constrained to be the second node in the second layer since the most recent interaction overrules its “first position in the second layer” constraint it previously had.

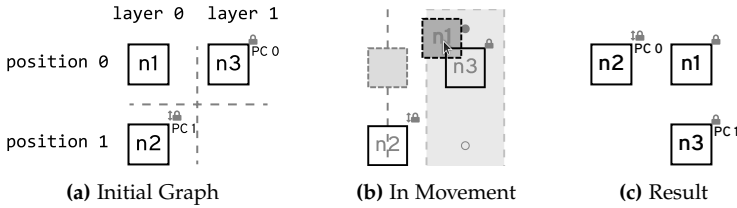
Figure 7.5 shows how relative constraints might be set using three nodes in the same layer. Here, relative layout constraints are visualized using an arrow pointing to the node the constrained node depends on. Figure 7.5a shows the initial constraints with node  $n_2$  constrained below  $n_1$ , Figure 7.5b shows the interaction with the relative constraint overlay<sup>3</sup>, and Figure 7.5c shows the final layout after setting the new constraint for  $n_3$  and updating the constraint of  $n_2$ . The initial graph only has a relative constraint ensuring that  $n_2$  is directly below  $n_1$ . When  $n_3$  is moved below  $n_1$ , as seen in Figure 7.5b, the relative constraint of  $n_2$  is updated to point to  $n_3$  instead, as seen in Figure 7.5c.

These constraints are mightier than model order can be. Some layouts cannot be created by the one-dimensional model order, as shown in Figure 7.1. Moreover, convention or semantics of a language might constrain the model order as it is the case for *Lingua Franca* and *SCCharts*. For *Lingua Franca*, the order of reactions cannot be freely changed. Additionally, different semantic elements are grouped and ordered by convention. For *SCCharts*, the edge model order represents their priority. Hence, changing them without adjusting the transition guards might change the semantic of a model. Here, diagram interaction might be the solution for the complicated layout of semantically different elements and an alternative to group model order and a way to constrain the *SCCharts* transition order without

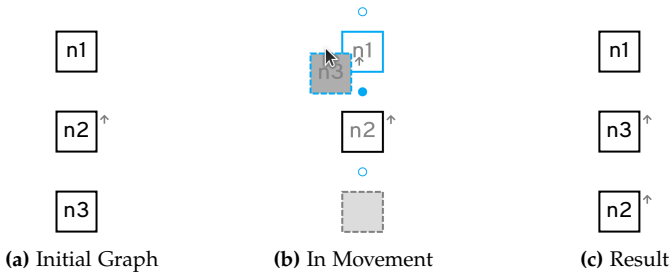
---

<sup>3</sup>Note that it is important that the overlay in Figure 7.5b is distinguishable from the absolute constraint overlay in Figure 7.4b if both absolute and relative constraints should be supported at the same time.

## 7.1. Interactive Layered Layout



**Figure 7.4.** Releasing n1 in layer 1 on position 0 updates the position constraints of n2 and n3. Layout constraints are visualized with a lock symbol.



**Figure 7.5.** Releasing n3 inside an existing chain makes n3 part of the relative constraint chain now consisting of n1, n3, and n2. Relative layout constraints on n2 and n3 are visualized with an arrow pointing to the node the constrained node depends on.

changing the semantics. However, I encountered that this is rarely used in practice since model order typically solves all SCCharts layout problems. Hence, the control layout constraints add to a diagram tool rarely matters in practice since model order can already express the most common cases that require control, e. g., the direction of edges.

At the same time model order improves constraints. Model order could be used to create a stable reference layout such that moving a node to a different layer does not result in changes because an algorithm randomly makes different decisions, as shown in Figure 7.3. Hence, model order might add stability to interactive diagram frameworks such that Principle (2) is not violated.

## 7. An Interactive Constraints Framework

### 7.2 Interactive Rectpacking

The framework described by Petzold et al. [PDS+23] does not only support the layered algorithm, but I also added support for the rectpacking heuristic (see Section 4.4) which solves the region packing problem (see Section 4.1). The region packing problem describes the problem of packing an ordered set of rectangles into a given aspect ratio while optimizing the scale measure (see Section 4.1.1).

Here, the one-dimensional model order lacks control over the layout and therefore control over the secondary notation since one dimension is not enough to describe a two-dimensional packing (*EVAL*). Numerical values such as a desired aspect ratio and a target width or target row height cannot be expressed by model order. Additionally, regions cannot be marked by model order. Therefore, the model order does not allow marking certain regions as row-dominant. Neither can model order express when a new row, stack, block, or subrow should be created. Instead, model order can only determine and control the order of regions (*EVAL*).

Hence, I propose to implement additional “line-break” constraints as part of future work on interactive rectangle packing, as detailed in Section 10.4.2. These would make it possible to specify when a new row, stack, or subrow should be created.

### 7.3 Generalizing Interactivity and Model Order

To summarize, interactive constraint frameworks benefit from the stable layouts model order creates by removing potential random decisions that might affect unrelated nodes or edges. Typically, this degree of control is enough for languages such as SCCharts to create all desired layouts. Constraints, however, increase the control a user might have on a model, which might be necessary for rectangle packing problems.

Other algorithms might also benefit from the diagram interaction and its interaction with model order. Tree layout is just a special case of layered layout and can utilize its constraints, as shown in the advised thesis of Carstensen [Car20]. These constraints, however, could mostly also be

### 7.3. Generalizing Interactivity and Model Order

covered by model order if the desired tree has no edges spanning multiple levels. Force or stress layout typically use interactivity to fix the position of certain nodes by dragging them to a desired position, which model order cannot do since it cannot express numerical values. Model order could, however, define an initial layout for force or stress layout, e. g., by utilizing the efficient layered algorithm described in Section 5.8.



# Structure-Based Editing

*Structure-based editing* [Pro08] is a type of WYSIWYG editing in which every change of the model results in a structurally correct model. E. g., by using a context menu one can create new states or edges between them.

The key difference between structure-based editing and a traditional WYSIWYG editor is, however, that the model is continuously layouted by automatic layout and that it typically uses a context menu instead of a palette. Hence, structure-based editing is a special kind of interactive framework that could also use model order for automatic layout. Instead of adding layout constraints by diagram interaction, diagram interaction creates nodes and edges via predefined editing commands. Moreover, compared to adding constraints, structure-based editing creates nodes and edges in a controlled manner allowing us to capture intended orderings at creation rather than by diagram interaction. Additionally, users might still introduce nodes and edges by textual editing when using the layout constraint framework proposed in Chapter 7, which is typically not the case during structure-based editing.

The necessary structure-based editing commands for state-machines have been sufficiently explained by Pronchow [Pro08] and reiterated by Jöhnk [Jöh22] for SCCharts. As a result, *entering text*, *clicking somewhere*, and *moving an element* were introduced as the atomic interactions each editing command consists of [Jöh22].

As it is the case for the interactive constraints framework proposed in Chapter 7 in Figure 7.2 on page 217, interactions do not directly change the diagram but rather the underlying textual model. Again, the textual model presents the ground truth, although it is typically hidden from the user during structure-based editing. Moreover, for structure-based editing

## 8. Structure-Based Editing

compared to the proposed constraints framework, diagram interaction does not translate to constraints that should result in the desired secondary notation. Diagram interaction rather creates desired model order to express the desired secondary notation for potentially new nodes and edges that are automatically layouted with model order in mind, which I sketch in the following on the example of SCCharts.

### 8.1 Model Order in Structure-Based Editing

Adding new states and transitions is very intentional in structure-based editing for SCCharts since a user may directly point to the place the new state should be created indicating desired secondary notation. To create a state, a user selects the appropriate context menu entry, clicks at the desired position (*clicking somewhere*), and enters the states name (*entering text*). Other commands work similarly such that the *clicking somewhere* and *moving an element* commands show the desired coordinates of a state or transition and with it the desired secondary notation and the user intention. Hence, I want to sketch how such an interaction could potentially create a model order that reflects the intention and that keeps the automatically layouted diagram stable during the interaction. To do this, I present the *adding a successor state* command for SCCharts.

As described by Jöhnk [Jöh22], *adding a successor state* has the following four steps.

1. A user selects the appropriate context menu entry on a state that will be the predecessor state to the new successor state.
2. A user clicks on the diagram to indicate where the state should be placed (*clicking somewhere*).
3. A user specifies the state name (*entering text*).
4. An optional transition guard and effect can be specified (*entering text*).

Here, the second step, the *clicking somewhere* action expresses the intention where the new state and with it where the new transition might be placed.



## 8.1. Model Order in Structure-Based Editing

If a user clicks behind the current state, the transition might be a backwards edge and the node should be placed in a layer before the source node. Therefore, the new state should have a lower model order than its predecessor. Based on the concrete coordinate and the already placed states, a concrete node model order for the state can be inferred. If a user clicks in front of the state, the new state should be in the next layer, similarly to the diagram interaction shown in Figure 7.4. Hence, diagram interaction specifies a layer and a position in a layer a new state should be added to, which translates to a node model order. Hence, using the strict model order cycle breaker (see Section 5.3.1), the breadth-first model order layerer (see Section 5.4.2), and the model order barycenter heuristic (see Section 5.5.3) with nodes-and-edges pre-sorting (see Section 5.5.2), enforces the relative position of all states by model order and keeps the layout stable.

The model order of the transition that is added together with the new successor state cannot be fully determined by diagram interaction since SCCharts transitions have priorities. Therefore, I assume that users add the transition by priority. E. g., a new transition is added below the existing ones using the lowest priority and therefore the highest model order. Since only the order of real nodes, i. e., states, is enforced with the suggested model order configuration, the model order barycenter heuristic may change the order of the transitions in the layout to prevent edge crossings. If the lowest priority is not desired for the new transition, a user must assign a different priority by invoking a new command via the transition context menu to change the edge model order to indicate a different priority. Note that model order for structure-based editing can be expanded by using the whitespace or grouping given by a textual model, as proposed in Section 10.4.9, to increase the amount of control a user might have on the drawing. This could, e. g., be used to solve the problem illustrated in Figure 7.1 on page 216.

To summarize, the very direct and controllable way in which new nodes and edges are introduced in structure-based editing together with a textual model allows using model order to capture intention even more directly. Moreover, it is possible to even translate this intention and with it the secondary notation into model order (*EVAL*). Hence, I propose to implement this model order aware structure based editing as part of future work, as detailed in Section 10.4.7.



## **Part III**

# **Application**



# Model Order in Practice

In this chapter, I finally want to answer the question that may concern readers now:

- ▷ What are the recommended model order configurations for your modeling language such that using model order does not threaten readability, increases stability, and allows users to control the secondary notation?

In the following, I will first discuss the recommended layout algorithms for SCCharts (see Section 9.1) and Lingua Franca (see Section 9.2). Section 9.3 generalizes the insights from SCCharts and Lingua Franca and recommends model order configurations for other modeling languages. The last section, Section 9.4, presents a guide on how to determine good model order configurations and what is necessary to find correlations between node and edge model order and the desired secondary notation in models.

## 9.1 Model Order for SCCharts

SCCharts can be used to model control-flow using SCCharts state machines mixed with parallel execution using SCCharts regions. Moreover, it is possible to model SCCharts using inherently parallel data-flow equations. While previous chapters illustrated how model order can be used for SCCharts control-flow and regions, model order is not used for SCCharts data-flow. This is the case since the data-flow equations do not employ order in a way such that the variable usage in these equations corresponds to a desired ordering. Moreover, many data-flow actors have a fixed port order in SCCharts, which further limits the leverage model order can have on the order of a data-flow visualization for SCCharts. Hence, I focus on

## 9. Model Order in Practice

recommendations for SCCharts control-flow and SCCharts regions at the basis of personal professional experience with SCCharts and the studies presented in Sections 5.9.1, 5.9.2, 5.9.5, and 5.9.8.

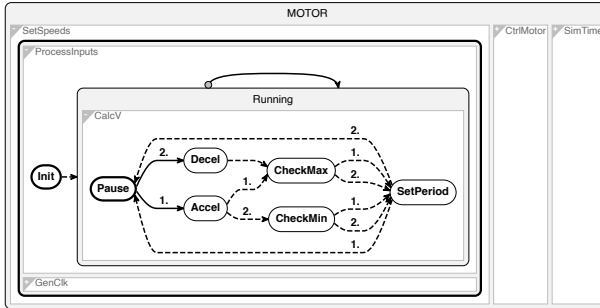
First, I discuss model order configurations for the layered layout problem posed by SCCharts state machines in Section 9.1.1. After that, I discuss recommendations for the packing of SCCharts regions in Section 9.1.2.

### 9.1.1 The SCCharts Layered Problem

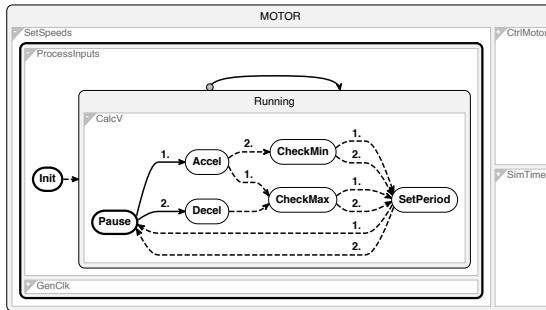
The state machine part of an SCChart is layouted with the layered algorithm (see Section 2.2). Compared to the declaration of regions, these state machines are typically the biggest part of an SCChart and carry the most meaning. Model order can here be configured for the three topological phases cycle breaking, layer assignment and crossing minimization for a *default* strategy and a *full control* strategy to be used on demand. I.e., one may choose a strategy that should be used as the default strategy and one strategy that should be used if developers want to constrain the drawing by model order.

The motor SCChart layouted with the legacy strategies, which were used before employing model order, the default strategies, and the full control strategies can be seen in Figure 9.1. In Figure 9.1a, the region placement, specifically the position of CtrlMotor and SimTime catches the eye since it would be clearly better to stack both regions as it is the case in Figure 9.1b and Figure 9.1c, which I will discuss in Section 9.1.2. Moreover, Figure 9.1a changed vertically the order of Decel and Accel and does not bundle the backward edges leading to the initial Pause state. Figure 9.1b instead shows a crossing free alternative that is mostly ordered and has a better region placement. The full control strategies in Figure 9.1c instead enforces all orderings and hence creates an edge crossing. However, this may just be the layout a user wanted to create. Moreover, it shows that CheckMin and CheckMax are not correctly connected to Accel and Decel, as seen at the correct solution in Figure 2.2. This shows that model order can help users to see potential flaws in a model.

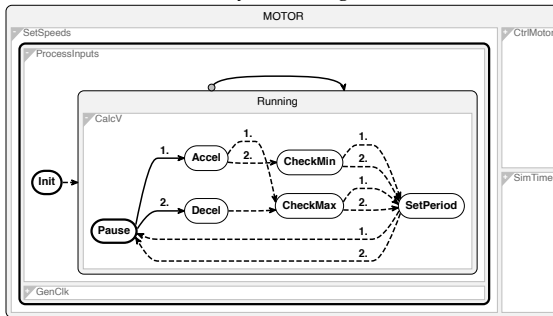
## 9.1. Model Order for SCCharts



(a) Legacy layout strategies for SCCharts



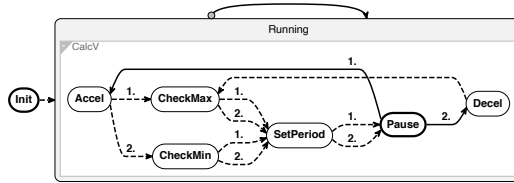
(b) New default layout strategies for SCCharts



(c) Full control strategies for SCCharts

**Figure 9.1.** A variant motor SCChart using the legacy layout, the new default strategy, and the full control strategy.

## 9. Model Order in Practice



**Figure 9.2.** Using the greedy cycle breaker for the erroneous motor SCChart without additional constraints

### SCCharts Cycle Breaking

Before I introduced model order for cycle breaking in 2021, SCCharts used the greedy cycle breaker (see Section 5.3.2), which has a hard time capturing user intention and expressing the desired secondary notation, as shown in Section 5.9.1. Even though Figure 9.1a has the same edge reversals as its alternatives, this is only the case since the initial Pause state was constrained to the first layer. Without this constraint, the inner state machine would result in the layout depicted in Figure 9.2, which is clearly *ONO*. Hence, I propose a model order cycle breaking strategy as the default.

In the study in Section 5.9.1, the strict model order cycle breaker performed best. The other evaluated strategies—the depth-first, breadth-first, greedy cycle breakers—failed to properly group states and create desired alignment. This is particularly interesting since the textual models of the evaluated SCCharts were created while model order did not influence the layout. This shows that many developers intuitively used a sensible node model order to indicate the flow in a model. This is the case since developers typically start to write down the initial state and order the new states below depending on the order of transitions they specify next. Hence, the model order of an edge’s source and target indicates the direction of an edge and should hence be used to control the direction of edges.

One potential result of the strict model order cycle breaker are dangling nodes. Figure 9.3 illustrates undesirable dangling nodes at the example of the obfuscated Backhoe SCChart. If a state is ordered to go against the model order, as seen in Figure 9.3a, a dangling node is created, as seen



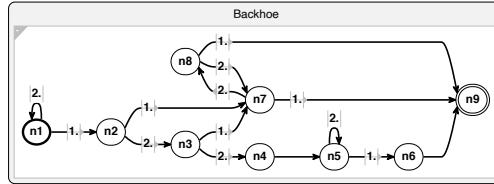
## 9.1. Model Order for SCCharts

```

...
state n6
...
state n8
...
state n7
...
final state n9

```

(a) ONO textual model



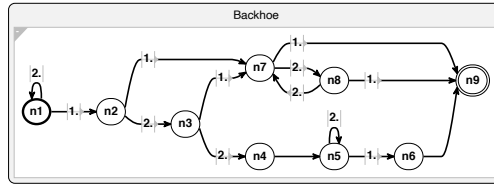
(b) ONO layout

```

...
state n6
...
state n7
...
state n8
...
final state n9

```

(c) OYES textual model



(d) OYES layout

**Figure 9.3.** The obfuscated Backhoe SCChart depicts a state machine that operates a backhoe. Figure 9.3b has a dangling node n8, while Figure 9.3d shows the desired layout without a dangling node.

in Figure 9.3b, although the node is not a source. E. g., state n8 is left of state n7 even though it has one incoming and one outgoing edge and could hence also be to the right of it.

I argue that this is not a downside but rather an indicator for a potential user error and an ONO textual model rather than an ONO layout algorithm. If such a dangling node occurs while using the strict model order cycle breaker, this points to a bad textual model with bad textual secondary notation, which might mislead users. The ONO layout in Figure 9.3b either happens because the developer was careless and did not use appropriate coding styles, or since the developer thought that the model actually executes n8 before n7 meaning that the flow was perceived differently. The appropriate solution in both of these cases is to fix the textual model by moving n8 after n7, as seen in Figure 9.3c to produce the OYES layout in Figure 9.3d.

## 9. Model Order in Practice

Hence, I propose to use the strict model order cycle breaker for SCCharts since it provides stability, control, and improves secondary notation by bringing the desired flow expressed by state ordering in the textual model into the diagram. Here, the default strategy is also the full control strategy.

### SCCharts Layer Assignment

As explained in Section 5.4.2, model order layer assignment can be used to group nodes in the same layer and uses a breadth-first node order to do this effectively. However, its use for SCCharts is limited because SCCharts developers tend to not order their nodes breadth-first, as seen in the analysis of the SCCharts models in Section 5.9.5 and Section 5.9.8.

SCCharts developers typically use control-flow branches, as depicted in Figure 5.20a. Here, states are grouped together forming branches from a “source”<sup>1</sup> to the last state of the flow. Using the breadth-first model order layerer or any other enforcing model order layerer results in an undesired layout, such as the one depicted in Figure 5.20b, creating unnecessary long edges.

Hence, the breadth-first model order or any model order layerer should not be used per default. The breadth-first model order layerer should only be used if the layering should be controlled and if users are willing to order states breadth-first. If such a breadth-first order is used, cycle breaking and layer assignment can create most<sup>2</sup> layerings. Hence, this option may be part of a full control layout mode that could be used as an alternative to the default model order configuration if users adapt the atypical breadth-first node order. As a default strategy, I, however, recommend a network simplex layerer [GKN+93] or a simple longest path layerer, which only use model order implicitly as iteration order.

---

<sup>1</sup>Sometimes this is not a real source but just a state from which most of the control flow originates, which, however, might have an initial state leading to it with some kind of initialization.

<sup>2</sup>Sections 5.8 and 7.1 give some insights about the limitations.

## SCCharts Crossing Minimization

For the crossing minimization problem posed by an SCChart, I recommend separate default and full control modes for model order crossing minimization to solve the issues of the legacy strategy depicted in Figure 9.1.

The default mode should use model order as a tie-breaker without creating additional edge crossings, as seen in Figure 9.1b. The full control mode should enable users to completely or partially control the order of edges on a node and the order of nodes in a layer while potentially creating additional edge crossings, as seen in Figure 9.1c.

As the default model order configuration, the prefer-edges strategy (see Section 5.5.2) should be used to pre-sort the ports and nodes. As shown in the quantitative model order metric study for SCCharts (see Section 5.9.5), pre-sorting has overall no negative effect<sup>3</sup> on the number of edge crossings. I recommend the prefer-edges pre-sorting strategy over the nodes-and-edges or prefer-nodes approaches, since priorities of edges seem to play a bigger role for SCCharts developers than the vertical order of nodes. Additionally, the default model order configuration should use weighted model order crossing minimization (see Section 5.5.4) as a secondary metric to edge crossings. This can be achieved by using, e. g., 0.001 as the order violation weight for port and node order violations. Using these configurations, the default crossing minimization strategy still aims to reduce the edge crossings. Additionally, it creates a stable and partly controllable solution that visualizes the transition priorities showing desired secondary notation in the layout if doing so does not impact readability. Especially the pre-sorting step proved to be the solution for many layout problems in legacy models such as the model by Motika depicted in Figure 3.2.

For presentations, papers, or to control the secondary notation of the layout, a full control model order configuration may be necessary. Here, cycle breaking uses the strict model order cycle breaker, layering may optionally use the breadth-first model order layerer, and crossing minimization should use full control model order crossing minimization (see Section 5.5.5). In-

---

<sup>3</sup>The study sometimes found a worse number of edge crossings if model order pre-sorting was used. However, this was only the case since the pre-sorting and the fixed number of random tries for layer sweeps resulted in different local minima, since different random permutations were used.

## 9. Model Order in Practice

stead of actually doing crossing minimization, the configuration only uses pre-sorting with the prefer-edges approach and skips crossing minimization entirely.

For SCCharts, all potential layer orders can be created by model order. This is the case since SCCharts have only one source node, i. e., the initial state, SCCharts have only “fake”<sup>4</sup> dangling source nodes, SCCharts have no feedback edges, and SCCharts have no hyperedges. Additionally, one must adjust the transition guards to be able to freely reorder transitions such that a changed edge model order, which impacts the transition priority, does not change the semantics of a model.<sup>5</sup> With this configuration, developers can control the layout to visualize the secondary notation of a model or to fix problems in a layout without potentially complicated and brittle diagram interactions.

### 9.1.2 The SCCharts Region Packing Problem

Back in 2020, the SCCharts region packing problem discussed in Section 4.1 was solved using the simple heuristic presented in Section 4.3. The simple heuristic, while it creates stable packings with a clear reading direction as desired by SCCharts developers, creates packings with low readability that do not match the desired aspect ratio and have a low scale measure. The incentive to develop the rectpacking heuristic presented in Section 4.4 were the often clearly ONO packings produces by the simple heuristic, such as the one in Figure 4.1a. The simple heuristic does not stack regions and produces malformed regions during whitespace elimination since all regions will be expanded to the row height. Additionally, ordering rectangles by size, as done by many strip-packing algorithms [DD92] to improve the packing, was never an option. SCCharts developers require a stable region order to orient themselves in their sometimes large models, as reported by SCCharts experts and experienced by myself. Hence, if a developer has zoomed into a large model to read some details, the mental map is easily compromised

---

<sup>4</sup>I consider dangling nodes fake if they are not strictly necessary for a minimum edge length layout. E. g., if dangling nodes are not source nodes, they are fake dangling nodes.

<sup>5</sup>Using this approach, I created all intermediate layouts for different layered phases to show the different calculation steps and possible orderings in this work.

if a change results in the currently interesting region moving out of the view port only because the region size influences the region order. Since the simple heuristic solved the stability issue with the layered layout equivalent of putting each node into its own layer, an order preserving—or rather reading direction preserving—and compact solution to the region packing problem is required for the layout of SCCharts.

Hence, the concept of model order as a reading direction led to the rectpacking heuristic presented in Section 4.4 with improved readability while keeping the region reading direction stable and consistent. Here, the evaluation in Section 4.6.1 showed that potential local optimizations for this heuristic did not influence the global scale measure—the main readability criterion for the region packing problem—of the evaluated models. Even the locally optimal solution did not significantly improve the global scale measure of the layouts. Hence, I recommend using the rectpacking heuristic without the optimization presented in Section 4.4.5 as the default. Additionally, I recommend enabling users to interactively set row and subrow constraints, as detailed in Section 7.2, to improve the control users have over the packing. Controlling the desired aspect ratio or the target width the drawing might have is here too indirect to influence the packing. Hence, rectpacking and therefore the packing of SCCharts regions requires diagram interaction to set constraints to increase the control users have over the secondary notation and the packing.

## 9.2 Model Order for Lingua Franca

Lingua Franca [LMB+21] is a coordination language that introduces new challenges to model order aware layout compared to the layout of SCCharts.

Lingua Franca mainly models data-flow and only models control-flow via *modes*, which I omitted here, since they share the characteristics of state machines languages such as SCCharts. Lingua Franca has different model order groups, i.e., there is not only one type of node but several. These different semantic elements do not adhere to one total model order but are defined in separate model order groups, as detailed in Section 2.3. Additionally, some elements, such as reactions, should have their vertical ordering

## 9. Model Order in Practice

enforced, as desired by Lingua Franca developers. Moreover, mixing reactors with low level elements such as actions, timers, startup, or reactions on the same hierarchy level is deemed bad style, according to Lingua Franca experts. Hence, a model such as Figure 2.4 should introduce reactors to capsule the timer and the reactions, as it is done, e. g., in Figure 2.3. This would allow to use a reactor-only layout mode, which does not need model order groups.

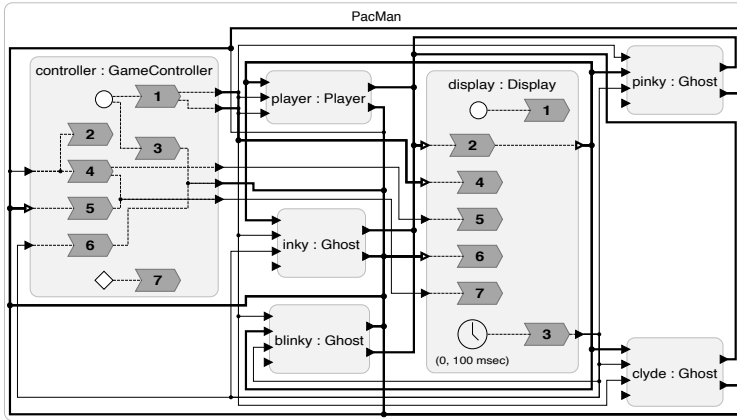
Since Lingua Franca does, as SCCharts, typically not adhere to a breadth-first node order, I discuss model order configurations only for the topological phases cycle breaking and crossing minimization. Model order layer assignment cannot be used effectively for Lingua Franca. As a comparison, Figure 9.4a shows the PacMan model layouted with the legacy layout strategies, which partly already employed model order, and Figure 9.4b shows the layout with the new default strategies. However, while this partly shows readability improvements for Lingua Franca, the static image cannot properly illustrate the increased stability of the layout, which I try to highlight in the following sections.

### 9.2.1 Lingua Franca Cycle Breaking

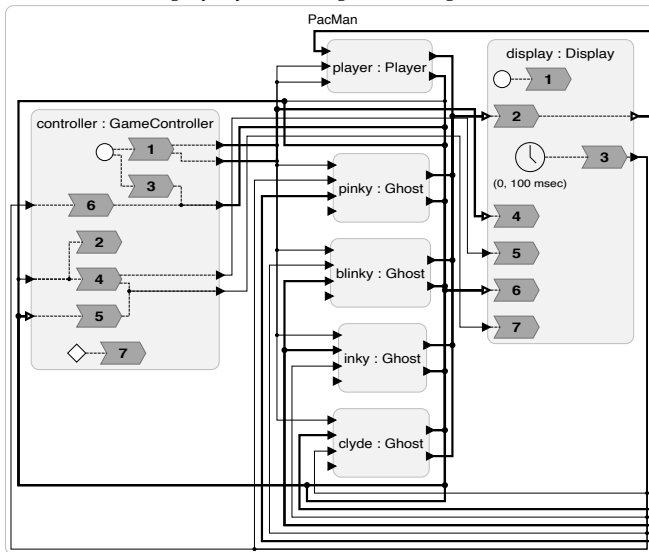
Like the layered approach for SCCharts, Lingua Franca should aim for two layout modes, one default mode that focuses on readability that uses model order as a tie-breaker, and one full control mode that enables users to control the layout by using the model order. Here, the studies in Sections 5.9.3, Section 5.9.6, and Section 5.9.7 provide us the necessary insights to recommend a cycle breaking strategy for Lingua Franca.

Since most layered subproblems for Lingua Franca models are small, one can spend more computation time on cycle breaking. Hence, I recommend the strongly connected component model order cycle breaker (see Section 5.7.1), which was often preferred by experts and students in interviews. This strategy still allows to partly control the flow and with it the secondary notation of the model by node model order, prevents ONO layouts by enforcing model order blindly, and produces stable layouts. The strongly connected component model order cycle breaker does not fall into the pitfalls of the strict model order cycle breaker or the depth-first

## 9.2. Model Order for Lingua Franca



(a) Legacy layout strategies for Lingua Franca



(b) New default layout strategies for Lingua Franca

**Figure 9.4.** The PacMan Lingua Franca model using the model depicted by von Hanxleden et al. [HLF+22]. The full control layout of this model is depicted in Figure 9.5.

## 9. Model Order in Practice

cycle breaker. The strongly connected component model order cycle breaker does not cause undesired edge reversals because model order groups are employed. Moreover, the strongly connected component model order cycle breaker sufficiently captures intention and is able to control the model. The required total group model order is created by sorting startup before modal model, reactor, timer, reaction, action, dummy nodes, and shutdown<sup>6</sup> based on the evaluation in Section 5.9.6 and 5.9.7. Whether physical actions should be sorted after startup triggers separately from logical actions may also be an alternative, which could not yet be fully evaluated. If timing should be an issue, I recommend the depth-first node model order cycle breaker or the greedy model order cycle breaker. These strategies should use the total group model order described above as the visiting order and secondary criterion to backward edges, which are typically used to measure readability in this phase.

As a full control mode, the strict model order cycle breaker can be used. This is possible since convention rather than semantics creates the model order groups for Lingua Franca. Even though users typically group elements, the different semantic elements can be freely mixed and there is no constraint dictating that one has to define actions separate from reactions. Hence, the strict model order cycle breaker may fully control the flow of a Lingua Franca layout. Hence, the full control mode also produces the desired cycle breaking for the PacMan model, as seen in Figure 9.5 compared to Figure 9.4b. However, note that the startup and shutdown triggers have no concrete order in a Lingua Franca model. The startup and shutdown triggers are rather ordered by the diagram synthesis, which translates the Lingua Franca model into a layout graph. Hence, one has to make sure that the layout nodes that correspond to startup and shutdown still get a sensible model order assigned such that they remain source nodes and startup is at the top and shutdown at the bottom.

If the model has reactor-only hierarchies, the reactor order along edges

---

<sup>6</sup>Note that the order total group model order of shutdown does not mean that it should be placed as a sink and at the right of a model. The strongly connected component cycle breaker and any cycle breaker other than the strict model order cycle breaker will always place it such that its outgoing edges point to the right since it is a real source and will always be placed on the left.



matches the developer intention, while the reactor edge order is often just a list of the connections beginning with the first reactor. Hence, the strict model order cycle breaker (see Section 5.3.1) can be employed for reactor-only networks and will improve the legacy cycle breaking, as seen in Figure 9.4b compared to Figure 9.4a. However, note that reactors and their edges are not grouped by the syntax, as it is the case for SCCharts. Hence, their link is not as strong and the order between reactors and reactor edges might often be conflicting.

If further research shows no strong correlation between the reactor model order and the desired flow, a depth-first node model order cycle breaker should be used instead. Here, model order selects the first element in a reactor network and still works as a tie-breaker via the visiting order.

### 9.2.2 Lingua Franca Crossing Minimization

Crossing minimization similarly to cycle breaking for Lingua Franca needs to deal with model order groups. Hence, a default model order configuration should mainly optimize readability criteria and should use the total group model order presented in Section 9.2.1.

Since crossing minimization might take the node or the edge model order into account, one has to decide which one should be used for model order pre-sorting (see Section 5.5.2). Since reactor orders are important to developers, e. g., Display dx should be above Display dy in Figure 2.4b, but port orders are more expressive during crossing minimization, the nodes-and-edges pre-sorting strategy should be used. Moreover, the reaction order should be enforced during crossing minimization by the model order barycenter heuristic, as this is desired by developers (see Section 5.9.6). Note that I recommend an enforced reaction order, even though this might just be a developer preference based on the reaction numbering, as detailed in Section 10.4.9.

Lingua Franca has cross-hierarchical edges that easily break the mental map of a model. By collapsing and expanding reactors, the order of the reactor edges might change, as shown in Figure 5.34. In such big and well-connected models, stability is more important than readability criteria, as detailed by experts in Section 5.9.4. Here, the model order barycenter

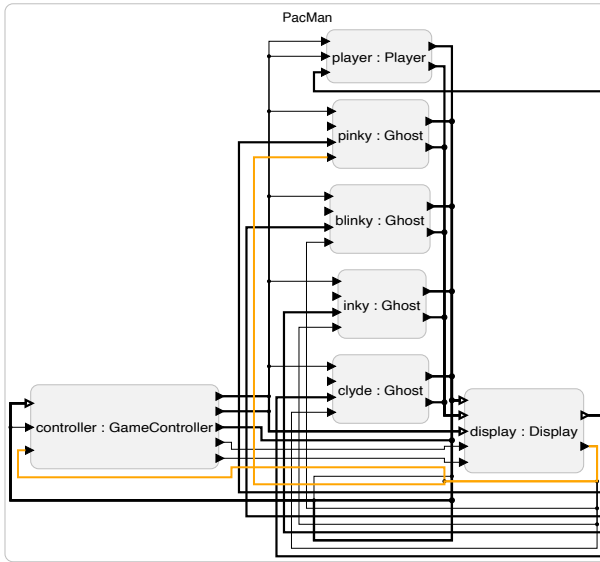
## 9. Model Order in Practice

heuristic (see Section 5.5.3) should be used to constrain the order of real nodes to achieve the desired stability while allowing edge crossings to be minimized. This can also be seen when comparing the legacy layout of the PacMan model with the new default. The legacy layout in Figure 9.4a shows that the different ghosts seem to have no fixed order. Figure 9.4b instead shows the desired order of ghost, which is pinky, blinky, inky, and clyde. Using this and the nodes-and-edges pre-sorting (see Section 5.5.2), the order of reactors remains stable. The additional edge crossings do not matter here for two reasons. First, a few more edge crossings are not that important if there are already many edge crossings [KPS14]. Second, highlighting of edges might mitigate the additional edge crossings by making edges recognizable and traceable, as seen in Figure 9.5. Here, the cluttered hyperedges of the PacMan model cannot be decluttered but highlighting will always show where an edge leads to.

Hence, I recommend two default strategies, one for small models and one for big models. The model order configuration for small models uses model order as a tie-breaker with nodes-and-edges pre-sorting (see Section 5.5.2) and weighted model order crossing minimization (see Section 5.5.4), which was reported as the best strategy in Section 5.9.4. Additionally, the model order barycenter heuristic constraints the order of reactions in a layer, since Lingua Franca developers perceived numbered elements that are out of order as irritating. The model order configuration for big models constrains the order of real nodes to create more stable layouts using the group model order barycenter heuristic. Here, the order of all real nodes is enforced while ports can move freely to eliminate edge crossings. Hence, model order keeps all important parts stable but still allows some crossing minimization for better readability.

The full control model order configuration for crossing minimization again uses the nodes-and-edges pre-sorting (see Section 5.5.2) and disables the crossing minimization entirely (see Section 5.5.5). However, since Lingua Franca might have multiple sources and therefore real dangling nodes in addition to hyperedges, model order groups, and feedback edges, I recommend using the greedy switch heuristic [EK86] as post-processing to solve ONO layouts that cannot be solved by model order alone, as detailed in Section 5.5.5. This may also be desired since the full control approach

## 9.2. Model Order for Lingua Franca



**Figure 9.5.** The PacMan Lingua Franca model with full control crossing minimization and a highlighted hyperedge.

typically creates bundled edges, as seen in Figure 9.5. Here, most edges are routed below the Ghost reactors.

Additionally, edge highlighting might here be necessary to ensure that the edges in Figure 9.5 can be followed. Here, visualization techniques, i. e., edge highlighting, rather than a better layout algorithm helps to make a layout more readable, which allows us to constrain the order of real nodes for stability and maintains the mental map. Hence, one should also consider that a diagram is in most cases not static but can be interacted with.

Note that the highlighting also illustrates that full control crossing minimization does not fully support hyperedges. The highlighted hyperedge in Figure 9.5 crosses itself. This could be solved by greedy post-processing of the hyperedge routes or by applying the greedy-switch heuristic [EK86], which can be tackled as one of the open problems, as detailed in Section 10.4.4.

## 9. Model Order in Practice

### 9.2.3 Lingua Franca Component Packing

Lingua Franca employs a second layout algorithm: the component packing algorithm presented in Chapter 6.

One case of component packing for Lingua Franca can be seen in Figure 6.1. Here, the reactions inside the Customer reactor are separate components with edges to the EAST or WEST side. To solve this, one should either constrain the order of ports on the Customer reactor to implicitly constrain the order of the unconnected reactions, or enforce the order of the components by model order.

Lingua Franca only uses a specialized version of the component packing problem, since no NORTH or SOUTH edges exist. Hence, Lingua Franca's separate connected components can be ordered by model order without running in the very space inefficient cases depicted in Figure 6.4a and Figure 6.4c. I argue that the model order component packing should hence be used for Lingua Franca instead of using the components' sizes or other criteria to order the components. Additionally, this improves the legacy layout configuration depicted in Figure 9.4a. In Figure 9.4a, the three separate components inside the GameController reactor were intertwined since separate connected components were not handled separately. This is solved using the model order component packing algorithm, as seen in Figure 9.4b. Hence, model order component packing makes Lingua Franca layouts much more stable, actually shows separate components separately, and allows developers to control the order of unconnected components in a Lingua Franca model to express secondary notation, as detailed in Section 6.3.

### 9.2.4 Lingua Franca and Hierarchy-Aware Layout

The hierarchical edges of Lingua Franca, and the problems they may cause for stability when collapsing or expanding reactors, raise the question of why one does not use hierarchy-aware layout. I.e., instead of separating the layout problem into separate subgraphs, the layout can be solved globally. My counterargument to this is that a global crossing minimization algorithm, which is more complicated, more time-consuming, and more likely to run into local minima, is often not necessary if users use good coding standards

## 9.2. Model Order for Lingua Franca

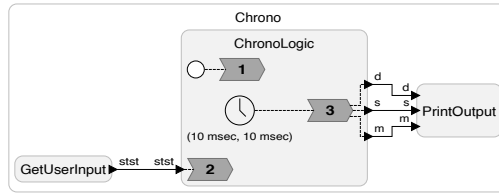
and employ model order.

In the following, I illustrate how model order can trivially solve the potentially hard problem of hierarchy-aware layered layout based on the Lingua Franca example in Figure 9.6, which I encountered in my research of Lingua Franca models.

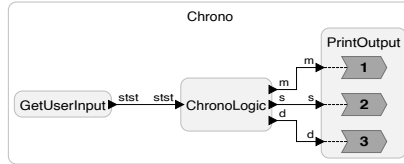
Figure 9.6 shows four different drawings of the Chrono Lingua Franca model, which depicts the Lingua Franca interpretation of the chronometer of Colaço et al. [CPP05]. Expanding either the ChronoLogic reactor or the PrintOutput reactor creates a crossing-free drawing, as seen in Figure 9.6a and Figure 9.6b. However, if both reactors are expanded at the same time, an edge crossing is unavoidable, as seen in Figure 9.6c. This happens since algorithms typically split the layout into separate subgraphs that are laid out independently, and because ChronoLogic and PrintOutput define a different order for *m*, *s*, and *d*, which represent a minute, a second, and a hundredth of a second. Hence, the top-level layout subproblem that deals with connecting the two reactors cannot reorder the already defined port orders of the two reactors and must create an edge crossing. The edge crossing can be prevented by using a hierarchy-aware layout algorithm to create the layout in Figure 9.6d. I, however, argue that this is often not necessary, since the edge crossing has a reason. In this example, carelessly defined interfaces create the edge crossing. If one inspects the textual model [SH24], it becomes apparent that the outputs of the reactor 3 in ChronoLogic are not *d*, *s*, and *m* but rather ordered *m*, *s*, and *d*, as seen in Section B.4, which causes the edge crossing.

Hence, the edge crossing points to an inconsistency in the textual model, which most likely would not have been found without the help of model order. Hence, hierarchy-aware layout is often not necessary if good coding standards are employed together with model order.

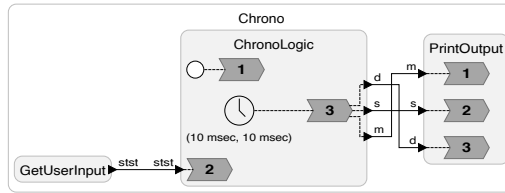
## 9. Model Order in Practice



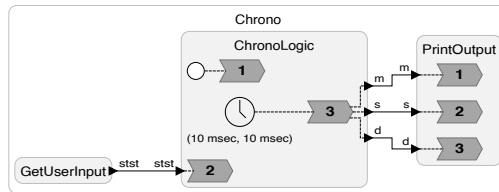
(a) Chrono model with ChronoLogic expanded. The outputs  $d$ ,  $s$ ,  $m$  are ordered as their model order dictates since no conflicts are found.



(b) Chrono model with PrintOutput expanded. Inputs  $m$ ,  $s$ ,  $d$  are ordered as their model order dictates since no conflicts are found.



(c) Chrono model with ChronoLogic and PrintOutput expanded. The edge crossings cannot be prevented since the outputs of ChronoLogic and the inputs of PrintOutput use a different model order for  $m$ ,  $s$ , and  $d$ .



(d) Chrono model with ChronoLogic and PrintOutput expanded. The edge crossing is prevented by either ensuring that  $d$ ,  $s$ , and  $m$  are ordered consistently or by doing hierarchy-aware layout.

**Figure 9.6.** The Chrono Lingua Franca model [SH24] inspired by the chronometer of Colaço et al. [CPP05].

## 9.3 Model Order for Other Modeling Languages

Model order configurations are language dependent. Hence, one cannot directly employ the model order configurations I recommended for SCCharts and Lingua Franca for other modeling languages. However, I argue that one can generalize the core concepts to other modeling languages that have a textual model that is automatically synthesized into a visual representation and follows the editing paradigm presented in Section 2.1.

Hence, this section serves as a guide to improve stability, secondary notation, and control by model order for modeling languages. Moreover, this section should help adopters to select model order strategies for rectangle packing, layered layout, and packing of separate connected components.

### 9.3.1 Rectangle Packing for Modeling Languages

Constraining a rectangle packing by model order may be desired if stability of the packing is more important than area efficiency or if the ordering of the rectangles is desired secondary notation. However, if users want more control over the layout, this requires to use layout constraints, as detailed in Section 7.2.

In Chapter 4, I presented two heuristics that create stable packings by considering the rectangles to be ordered by their model order. The simple heuristic in Section 4.3 is a sufficient algorithm if rectangles should be aligned in rows and the size of the rectangles is roughly the same. If the rectangle size varies between big and small rectangles, it is better to stack rectangles as it is done by the rectpacking heuristic in Section 4.4 since the simple heuristic is very area inefficient in this scenario. The rectpacking heuristic produces readable packings while maintaining a left-to-right reading direction such that rectangles can be read in model order using a row and subrow structure. Consequently, one should only use these heuristics if the packing should have a row or subrows structure and if this is desired secondary notation.

Both heuristics work with rectangles with a minimum size and can—if desired—eliminate whitespace (see Section 4.1.2) to fill potential gaps in the drawing to make it more aesthetically pleasing. Similarly, the proposed

## 9. Model Order in Practice

heuristics should not be used if an area efficient packing is desired or if stability and with it the reading direction of the rectangles is not important or less important than area efficiency.

### 9.3.2 Cycle Breaking for Modeling Languages

This section together with Section 9.3.3 and Section 9.3.4 suggest model order configurations for the layered algorithm. To create desired layouts, the cycle breaking step, which typically minimizes the number of backward edges, should create the desired flow by correct edge reversals to improve secondary notation. Hence, cycles need to be visualized such that a backward edge indeed shows when control or data flowing somehow “backward,” i. e., against their normal direction. To achieve this, the model order cycle breaking strategies presented in Section 5.3 work with the node model order as the intended flow. In an ideal textual model, the model order dictates the desired flow.

In the simple case, a modeling language only has one type of node, i. e., no model order groups (see Section 5.7) exist, and nodes can be freely reordered. If this is the case, the node model order should be used to control the flow and with it the secondary notation of a model by using model order at least as a tie-breaker, as detailed in the evaluation regarding SCCharts and Lingua Franca in Section 5.9. Especially for state machines languages, I argue that the flow can even be constrained by the model order using the strict model order cycle breaker presented in Section 5.3.1. This has the additional advantage that ONO layouts point to inconsistent textual orders, which might indicate an error in the model itself. Constraining the flow by model order might also be beneficial for other modeling languages that model control-flow since control-flow languages are typically very explicit in the declaration of elements that corresponds to nodes, as seen in SCCharts state machines and Lingua Franca modes. I argue that the correlation between desired flow and model order also holds true for other control-flow modeling languages such as UML flowcharts, sequence diagrams, and other state machine dialects. Hence, further research is required to confirm this, as detailed in Section 10.4.11.

Data-flow languages might also benefit from the strict model order



### 9.3. Model Order for Other Modeling Languages

cycle breaker, as it is the case for reactor-only networks in *Lingua Franca* or other textual data-flow languages with a single node type. However, if the different data-flow actors are only implicitly defined, e. g., by writing down an equation, the node model order derived from such an equation is typically not intended as a constraint. E. g., variable orders in *SCCharts* data-flow equations [GSS+24] or *Lustre* [HCR+91] equations do not carry enough intention to enforce their order in a layout. Hence, it may be better to use the greedy model order or the depth-first model order cycle breaker and to use model order as a tie-breaker.

The greedy model order cycle breaker focuses more on the minimization of backward edges and the depth first model order cycle breaker more on the structure of the graph. For the depth-first model order cycle breaker, the model order still defines the first element to start the depth-first search from. Hence, model order has still an effect as a tie-breaker. Moreover, one should use the edge model order as the visiting order for depth-first search if the edge order is more important in the textual model and carries more intention. Consequently, one should use the node model order as the visiting order if the order of nodes is more important. The depth-first cycle breaking algorithm should, however, not be used in very connected graphs that desire a specific layer grouping to prevent unintentional backward edges, as seen in Figure 5.59 on page 197. If this is the case, the greedy model order algorithm may find better edge reversals if it does not run into troubles finding the source node, as seen in Figure 9.2 on page 234.

If a language has multiple model order groups, i. e., multiple node types, and the models are still somehow small, the strongly connected component model order cycle breaker (see Section 5.7.1) can be used to detect the correct control- or data-flow cycles. If the runtime is an issue, a depth-first or greedy (group) model order cycle breaker can be used instead. Here, one has to potentially create a total group model order by prioritizing the different model order groups as a tie-breaker. This requires interviewing developers and analyzing models to find a good total group model order.

As a last note on cycle breaking, I think that one should never use the greedy cycle breaker that uses random decisions as a tie-breaker or any cycle breaking algorithm that employs random decisions. The results of Study 1 in Section 5.9.1 clearly show that the greedy cycle breaker has no

## 9. Model Order in Practice

effect on readability and a negative effect on stability, secondary notation, and control.

Moreover, one should always be aware of the visiting order of the commonly used depth-first algorithm such that the behavior in tie-breaking cases is deterministic, controllable, and as desired.

### 9.3.3 Layer Assignment for Modeling Languages

Layering or layer assignment aims to minimize the edge length but can also be used to partition nodes into groups to show desired secondary notation. Model order, however, cannot completely control this grouping because of its one-dimensional nature. Based on its performance with SCCharts, I argue that the breadth-first model order layerer should only be used if the nodes can adhere to a breadth-first order and it is desired to partition nodes into specific layers, as explained in Section 5.4.

Such a partition into layers might for example be desired if one wants to align actuators or sensors, as it is the case for the STPA control structure in Figure 5.5 on page 102 or for the STPA relationship diagram [PKH23] in Figure 5.21 on page 124. In both cases, the developer already knows the order of the different elements and utilizes a full control model order configuration such that the node type and the model order predetermine the topology of the layout. Note that enriching the breadth-first model order layerer by group model order only works in the trivial case. I. e., if different nodes should be partitioned by their model order group, as it is the case for STPA diagrams shown in Section 5.4.

Currently, I did not encounter a different use-case for model order layering, hence, most languages should utilize a network simplex layer assignment [GKN+93] or a simple long edge layer assignment algorithm [NTB05], which might use model order as the iteration order and hence as a tie-breaker. Hence, the suggested layer assignment algorithms mainly optimize readability. Moreover, the suggested layering algorithms limit the effect of model order on stability, secondary notation and control during layer assignment.

### 9.3.4 Crossing Minimization for Modeling Languages

Crossing minimization typically aims to reduce the edge crossings (hence the name), I instead propose that crossing minimization should rather visualize the desired vertical order and should be configured to create stable layouts using model order. However, finding a suitable model order configuration for crossing minimization is a little more complicated than finding one for cycle breaking and layer assignment since one can consider the node and the edge model order, as detailed in Section 5.5.

As a first principle, one can always use a tie-breaking configuration to create more stability without threatening readability, based on the evaluation in Section 5.9.5. Hence, all crossing minimization algorithms should use model order as a tie-breaker by letting model order determine the initial order of nodes in a layer and ports on a node. For this, either the prefer-edges or the nodes-and-edges pre-sorting (see Section 5.5.2) algorithms should be used.<sup>7</sup> One should select the nodes-and-edges pre-sorting if the node order matters. Otherwise, the prefer-edges strategy should be used. Using the prefer-edges strategy also has the advantage that the initial solution might often be crossing free even if the node and edge order are conflicting. Since it might still be the case that the initial solution is not crossing minimal, I advise using weighted model order crossing minimization (see Section 5.5.4), based on the feedback of the SCCharts experts and my personal experience with modeling languages. Here, the node and port order violations should be weighted with a violation weight of 0.001 or any small value that results in model order violations being a secondary criterion to edge crossings.

If layout stability is a bigger issue and model order should be used to keep the drawing more stable, the model order barycenter heuristic (see Section 5.5.3) can be used. Doing so keeps the order of real nodes, e. g., the order of states and reactors, stable if a pre-sorting strategy that preserves the node order is used. Hence, the model order barycenter heuristic should effectively only be used with the nodes-and-edges pre-sorting. As an alternative to keeping the order of real nodes stable is constraining the order of ports, as detailed by Schulze et al. [SSH14], which could also be achieved

---

<sup>7</sup>The prefer-nodes pre-sorting currently only serves as a demonstrator, but I have yet to find a language that could employ it. Hence, I currently cannot recommend it.

## 9. Model Order in Practice

by enforcing the port or edge model order. Both alternatives might make use of the greedy switch heuristic [EK86] if a few less edge crossings are more important than strictly keeping the order.

Finally, the full control mode (see Section 5.5.5), i. e., not doing crossing minimization, should only be used if stability is much more important than edge crossings or if crossing minimization compromises the underlying structure of a model, i. e., if one wants to fully control it. E. g., if a model is very connected and hierarchical, potential changes or expand and collapse of elements tends to result in many changes in the layout if stability is not considered, as explained in Section 5.5.3 based on Figure 5.34 and Figure 5.35 on page 149. Here, developers want to employ the full control strategy to have maximum stability and maximum control over the secondary notation of a model. Full control crossing minimization also has the advantage that it shows the structure of the model much clearer than a still cluttered solution with a few crossings less, as detailed by experts of SC-Charts and Lingua Franca. This control aspect of model order is often very important for human-made models. E. g., if one wants to create a specific layout for documentation or if one wants to create a structure-based editing approach (see Chapter 8), I advise to use a very controlling model order configuration such as this one. Again, a greedy switch heuristic [EK86] can be employed to solve ONO layouts caused by dangling nodes, feedback edges, hyperedges, as well as a too strict total group model order.

All presented model order crossing minimization strategies have a group model order variant, which can be used for model order groups, i. e., if different semantic elements create groups in the textual model. However, the different model order groups and their total group model order needs to be evaluated first to create the desired results, as detailed in Section 5.9.6. Additionally, different layout phases may use a different total group model order. E. g., in cycle breaking actions may be before reactions and in crossing minimization reactions may be before actions. Moreover, weighted model order crossing minimization may not count ordering violations between or in certain groups. E. g., if the action model order in Lingua Franca is deemed arbitrary, order violations between actions should not matter. Or, if actions and reactions are not really comparable by model order, one should configure that order violations between these groups do not count.

### 9.3. Model Order for Other Modeling Languages

Since evaluation and proper configuration of group model order is hence complicated, it is often easier to only use model order as a tie-breaker if different model order groups such that one does not enforce undesired orderings based on faulty group model order.

#### 9.3.5 Component Packing for Modeling Languages

Finally, I want to summarize when the packing of separate connected components with potential edges to the outside should use model order. This boils down to a guide when to constrain the order of separate connected components.

One should constrain the order of separate connected components, as shown in the Cutter example in Figure 6.2 on page 206, for stability and secondary notation. If only connections to opposite sides, e. g., only EAST and WEST connections exist, model order component packing does not threaten readability by increasing the drawing size, as detailed in Section 6.3. In the presented example, stability is most important to prevent that components change their order. As an alternative to model order, one could for example add layout constraints or use the component's "size," e. g., their actual size or the number of elements in them, for their ordering<sup>8</sup>. Layout constraints have the disadvantage that they typically need manual adjustment and are not implicitly given by the model, as it is the case for the model order. Using the size may threaten stability. This is the case since languages for MDE often show or hide the content of an element, making it smaller or bigger, e. g., reactors for Lingua Franca and regions for SCCharts may be expanded or collapsed. Additionally, adding new functionality may change the "size" of the components such that their ordering is affected. Using model order, i. e., ordering components by their minimal model order, e. g., the first occurrence of one of their elements, implicitly creates a stable component order. If the ordering is undesired, a user can always reorder elements textually such that the first component defines its elements first. This works best if the elements of the separate components are actually defined in separate groups in the textual model, which I would recommend as part of good coding conventions for textual modeling languages. Hence,

---

<sup>8</sup>But one should not use a random decision, as detailed in Section 5.1.7.

## 9. Model Order in Practice

model order component packing rewards good textual coding practices, visualizes bad habits, and shows flaws in the textual model.

If components might have arbitrary edges to the outside, I do not recommend model order component packing. As seen in Figure 6.3, the required area for a model order reading direction is much larger for general components. Moreover, model order component packing increases the number of edge crossings and may create cluttered packings. Hence, one should carefully consider if the resulting ordered packing is actually desired. However, if edges only point to opposite sides, i. e., to NORTH and SOUTH or to WEST and EAST, as it is the case for *Lingua Franca*, model order component packing does typically not increase the size of a packing. Hence, I advise to use the model order component packing for stability and the possibility to control the secondary notation in this case.

## 9.4 Lessons Learned

While Section 9.3 detailed what model order configurations might be used for modeling languages, this section presents the lessons learned on finding out what model order configurations might be useful. Hence, this section is largely based on personal experience, which stems from researching model order and interviewing developers.<sup>9</sup> As a guide for future model order engineers, I will focus on three different problems:

1. How to get good models to evaluate model order strategies for?
2. How to find developers for a given language to provide feedback?
3. How to get good feedback for model order configurations?

### 9.4.1 How To Get Good Models?

To evaluate model order, one needs models to evaluate. Without such models one can only guess useful model order configurations based on the recommendations in Section 9.3. To find good model order configurations for a given language, one has to actually try them out on models to see the effect on stability and the level of control they might give to a developer. Moreover, good models are essential to see whether the resulting layout is desired and to communicate with developers about model order or layout in general. Hence, the `cdn_cache_demo` model created by Magnition, the Lingua Franca playground<sup>10</sup>, the SCCharts models by S&B, and the various SCCharts models created by the real-time and embedded systems group in Kiel and their students made my research and this thesis possible in the first place.

Good models to evaluate model order are models developed by experts. Models of novices may use bad secondary notation in the textual model, as it was also encountered by Petre [Pet95] for graph drawing. Especially student models created during weekly exercises are often very poorly

---

<sup>9</sup>As a real guide to conducting studies I recommend “Experimental Human-Computer Interaction: A Practical Guide with Visual Examples” by Helen Purchase [Pur12].

<sup>10</sup><https://github.com/Lf-lang/playground-lingua-franca>

## 9. Model Order in Practice

written. Moreover, these models often have *copy-paste model order anomalies*, which are models that have a model order that is mainly influenced by how easy one can create the model with as much copy and pasting as possible. If this is the case, the model order is not intended and therefore flawed.

Additionally, one has to take into account that old models that were created before model order had an effect on layout might just not use a sensible model order. This might be the case even though I encountered a different effect for SCCharts. Moreover, models might have “weird” model orders because developers could not control the layout and randomly tried to reorder elements, which I encountered in several SCCharts create for research papers.

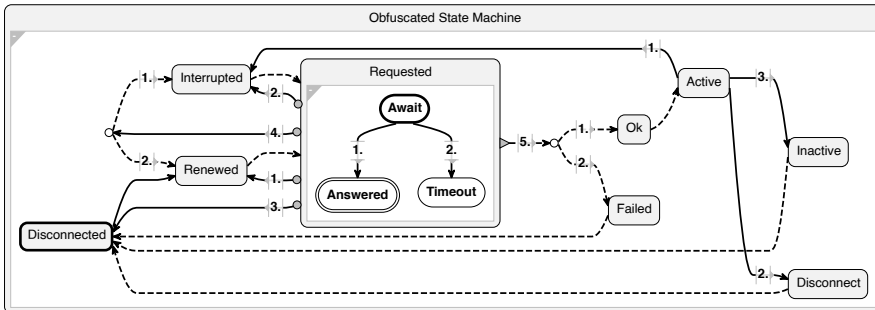
Realistic models, e. g., models for a real use-case used in industry, are much better than models from weekly exercises to evaluate model order. This has three reasons. First, professional developers are typically not novices in the given language. Second, developers care about a good textual model that is properly structured and documented since they will most likely work with it longer than one week. Third, such models have a representative size and complexity. Hence, if realistic models are small, one can decide to use more time-consuming layout algorithms and model order configurations. If the layouts tend to be cluttered, i. e., if the underlying graphs are very connected, stability is more important together with a proper edge highlighting.

However, realistic models are very hard to come by. Companies that use a modeling language might not give them away to researchers<sup>11</sup> or only provide obfuscated models. If models are obfuscated, as seen in Figure 9.3 on page 235, one can often no longer identify their meaning. Their meaning, however, is necessary to identify what ordering might be intended. Hence, model obfuscation should, if possible, only remove transition labels, remove variables, and simplify and generalize node labels, as seen in Figure 9.7. Here, anything domain specific was removed from the depicted state machine such that one can roughly understand the relationship between different states but cannot infer the concrete implementation and potentially expose trade secrets. Note that I manually obfuscated the node

---

<sup>11</sup>In my case, S&B provided unobfuscated models for research purposes with the option to depict obfuscated variants in publications, which I am very grateful for.





**Figure 9.7.** An obfuscated SCChart by S&B.

labels for Figure 9.7. Transitions and variables, however, can be easily and automatically removed from models. Hence, proper model obfuscation is one of the open problems described in Section 10.4.8.

Moreover, one should be aware of different writing styles if a modeling language is used by different developer groups. Different groups might use different coding standards and potentially also employ model order, textual secondary notation, and graphical secondary notation differently. This should be investigated for a modeling language by comparing different model sources from different groups or different times, e.g., before model order affected the layout.

### 9.4.2 How To Find Developers?

Not only does one need models to find model order configuration, one does also need developers that know the meaning behind the models or are familiar with the given modeling language to rate whether a layout is “better.” Moreover, developers can create more models, making finding developers even more important.

If one searches for model order configurations in a research context, one might want to ask students that know the given modeling language. I, however, argue that this is not ideal. While one might find plenty of stu-

## 9. Model Order in Practice

dents<sup>12</sup>, students are often only novices and do hence typically incorrectly employ or identify textual and graphical secondary notation, as seen in Section 5.9.7. Hence, one should try to find experts of a given language, which are, however, very hard to come by. Additionally, experts typically have a very full calendar and potentially no time for extensive feedback. Developers might, however, be more willing to participate if they see a clear benefit from better layout algorithms. Hence, a live presentation of stability and secondary notation might result in more and better feedback.

Ideally, one is oneself an expert developer of the given modeling language. E. g., since I had prior experience in SCCharts development I could more easily identify ONO layouts and their OYES counterpart than I could for Lingua Franca.

However, a well documented library of models, e. g., the Lingua Franca playground, might mitigate the lack of expert developers. If the meaning of a model can be identified from a description, one can use this description to evaluate whether a given model order configuration does the correct thing.

### 9.4.3 How To Get Good Feedback?

If one has good models and good developers, one has to ask the developers the right questions and show them the right models. But how can one find the right questions?

First, just letting people develop a model while recording might not always work. When doing so, the models might lack the complexity of a typical model or creating them will take too much time such that the required expert developer would most likely not be available.

Second, there is a difference between a domain expert for a given language and an expert for graphical notations and layout algorithms. In my experience, developers inexperienced in automatic layout have a hard time articulating what they like or dislike in a layout. However, such developers, as most people, identify things that they dislike far easier than things that they like. Hence, I often used non-perfect layouts to trigger responses different from “I don’t know” or counter-questions such as “Well,

---

<sup>12</sup>When researching a specialized modeling language, i. e., SCCharts or Lingua Franca, one may often only find very few students to interview.

which layout is typically regarded as better?“, which I encountered in informal interviews or meetings with experts.

Third, developers typically do not know what they want if they have no experience in automatic layout. More specifically, developers do not know whether certain layout goals conflict each other. E. g., stability and minimal edge crossings cannot be achieved at the same time. Moreover, developers inexperienced in automatic layout have a harder time envisioning what certain options, e. g., constraining the order of real nodes, might do to their diagrams without seeing it on an example model. Additionally, developers may have problems to mentally apply presented options to different models.

Hence, I often tried to pre-question experts before actually conducting a study to select different options that might capture different desirable or undesirable alternatives. E. g., to get feedback, an expert would be presented with two to four different pictures and could choose whatever option fits best. Additionally, I found that the reason for a preference or antipathy for specific layouts proved to be far more important than a resulting rating of layout options. Hence, I argue for interviews rather than questionnaires of developers with prepared layouts showing potential flaws to weight them against each other. Moreover, stability and control require a live demo to grasp the full extend of these goals and their effect on the layout.



# Conclusion and Future Work

This work investigated and illustrated the usage and capabilities of model order using SCCharts and Lingua Franca as example modeling languages. The following three key takeaways best summarize this work.

First, automatic layout for modeling language should not only aim for readable layouts measured by some aesthetic criteria but should also focus on stability and ways to control the secondary notation of a model, as detailed in Chapter 3. Second, model order, the order of elements in the textual model, provides the stability and control over the secondary notation that a modeling language requires. Third, automatic layout algorithm should therefore always consider model order, at least as a tie-breaker.

To further summarize the paper, I want to answer the two research questions of this work—*IMPL* and *EVAL*—based on the presented layout algorithms and investigated modeling languages. I.e., how can model order be integrated into layout algorithms, and how does model order affect readability, stability, secondary notation, and control for rectangle packing, layered layout, the packing of separate connected components in SCCharts and Lingua Franca.

## 10.1 Summarizing Model Order in Rectangle Packing

Chapter 4 presented two rectangle packing algorithms, the simple heuristic (see Section 4.3) and the improved rectpacking heuristic (see Section 4.4), which solve the region packing problem (see Section 4.1) posed by SCCharts (see Section 2.2).

## 10. Conclusion and Future Work

Here, model order can be integrated as the desired reading direction of rectangles (*IMPL*). For modeling languages such as SCCharts, a stable packing with a consistent reading direction is required for the desired secondary notation. Hence, I only focus on the effect of the presented rectangle packing heuristics on readability as well as control to answer *EVAL* for the presented rectangle packing heuristics.

While enforcing the model order reading direction, the simple heuristic produces many *ONO* packings with low readability according to the scale measure (*EVAL*). The rectpacking heuristic could improve this by stacking rectangles forming rows with subrows. Specifically, the rectpacking heuristic improved the scale measure while keeping the left-to-right reading direction consistent. Moreover, the rectpacking heuristic packs rows such that they have a row dominant element as a reference point (see Section 4.1.3) and still allows eliminating potential whitespace between rectangles (see Section 4.1.2) to further improve the visual perception of a row structure with a consistent reading direction. Additionally, I presented optimizations to increase the scale measure and with it the readability of a packing if the packing is not time critical, as seen in Section 4.4.5.

The evaluation of the optimal solution of the region packing problem, which was presented as a maximization problem in Section 4.5, compared to the heuristics showed that the rectpacking heuristic was able to create much more readable drawings (*EVAL*). Optimizations of the rectpacking heuristic only affected readability if rectangles vary in size such that they are stackable (*EVAL*). Additionally, for hierarchical models a locally optimal solution calculated using the maximization problem did not significantly improve the global scale measure and hence readability (*EVAL*), as shown in Section 4.6.1.

To summarize, for rectangle packing and the region packing problem, model order negatively affected readability (*EVAL*). However, since model order provides the desired stability with a consistent reading direction, it is desired by users of SCCharts and potentially other modeling languages and needs to constrain the reading direction in the packing (*EVAL*).

Even though model order controls the reading direction, model order cannot fully control a rectangle packing since model order has no way to express numerical values required for potential goal functions such

as a desired aspect ratio and a target width (*EVAL*). Moreover, the one-dimensional nature of the model order cannot express line-breaks in a row-structure, i. e., when to start a new row, stack, or subrow in a two-dimensional packing (*EVAL*). Here, an interactive constraint framework must provide additional control over the secondary notation, as presented in Section 7.2.

## 10.2 Summarizing Model Order in Layered Layout

For layered layouts, the effect of model order on readability, stability, secondary notation, and control is partly dependent on the modeling language and the structure of its underlying graphs, as shown in Chapter 5.

During the cycle breaking phase, I showed that the node model order can determine the direction of edges (*IMPL*). For SCCharts, the node model order typically determines the desired edge direction in the drawing since there is only one type of node. Here, the node model order controls the edge direction for desired secondary notation (*EVAL*). If the node model order should not constrain the direction of edges, the model order can be used as a tie-breaker and a secondary criterion, or be used as the iteration or visiting order (*IMPL*), e. g., during depth-first search. This has the advantage that using model order does not compromise the number of backward edges, which is the main readability criterion for cycle breaking (*EVAL*).

During layer assignment, the node model order can determine the partitions, i. e., the layers of the layered layout, if the node model order uses a desired breadth-first node order (*IMPL*). Even though this can be used for modeling languages such as STPA control-structure diagrams, this can typically not be used for SCCharts since developers do not order their states breadth-first. Hence, enforcing the model order during layer assignment for SCCharts does not result in a desired layering (*EVAL*).

For crossing minimization, the node model order and the edge model order of a graph can be leveraged (*IMPL*). As a first step, the nodes in a layer, and the edges on a node need to be pre-sorted based on the node and edge model order, which serves as a tie-breaker since it influences the

## 10. Conclusion and Future Work

first possibly crossing minimal solution that will be found. Using model order pre-sorting, model order can provide stability and partly control the secondary notation of a model without threatening readability (*EVAL*). Here, users can configure the importance such that the edge model order, the node model order, or both is used. Additionally, model order can here be used as a constraint. This is possible by constraining crossing minimization strategies such as the barycenter heuristic such that they cannot reorder nodes or edges based on the model order (*IMPL*). To fully control the position of nodes and edges, crossing minimization can be skipped entirely by only executing the pre-sorting step (*IMPL*). Hence, if the order of nodes and edges is desired, fully controlling the layout by model order keeps the layout stable and shows good secondary notation (*EVAL*). Moreover, if the layout is controlled by model order, an *ONO* layout often points to a user error or an inconsistency in the textual model.

To improve the runtime of the proposed algorithm, which is higher than necessary due to the assumption that the ordering may be changed by other phases and intermediate processors, I presented an algorithm to solve the first three phases of a layered algorithm efficiently using model order, as shown in Section 5.8.

The last two phases of the layered algorithm, node placement and edge routing, cannot effectively use model order. This is the case since both phases need to determine concrete coordinates and not only orderings of nodes and edges, as it is the case for cycle breaking, layer assignment, and crossing minimization.

If multiple different node types exist, as it is the case for *Lingua Franca*, it is often possible to use computation heavy solutions for each layout phase. This is possible since large models are typically divided into small layout subproblems based on the model hierarchies, as shown in Section 5.7.

However, if multiple node types exist and a graph has hyperedges, feedback edges, or dangling nodes, model order cannot completely control the drawing. Here, it might be necessary to use layout constraints, as detailed in Section 7.1. When using layer constraints, model order can again be used for the reference layout (*IMPL*) such that model order provides stability to sometimes brittle constraint frameworks (*EVAL*). This can even be used to interpret diagram interaction as model order when using structure-



## 10.3. Summarizing Model Order in Component Packing

based editing, as presented in Chapter 8.

For the concrete languages SCCharts and Lingua Franca, I recommend concrete layout strategies based on their desire for stability and control over secondary notation in Section 9.1 and Section 9.2. This can be generalized to other modeling languages, as shown in Section 9.3, if one considers the lessons learned on how to find good models and how to find and interview model developers for their preferences, as detailed in Section 9.4.

## 10.3 Summarizing Model Order in Component Packing

Chapter 6 illustrates how model order can be used to constrain the reading direction of separate connected components with connections to the outside (*IMPL*) to increase stability, control, and possibly secondary notation (*EVAL*).

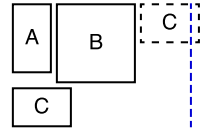
In the general case using separate connected components with arbitrary edges to the outside, I cannot recommend to use model order component packing. In the general case, the potential connections to the outside and the reading direction constraints create an unreasonable amount of whitespace and a significantly lower scale measure and much more edge crossings such that model order component packing threatens readability (*EVAL*). This is the case since model order component packing keeps the layout stable by enforcing a left-to-right reading direction by placing components using a row structure (*EVAL*). If, however, the connections to the outside are limited to neighboring sides, as it is the case for Lingua Franca, the model order component packing algorithm adds stability and control (*EVAL*), as detailed in Section 9.2.3. Moreover, this configuration prevents additional edge crossings and typically does not increase the layouts size.

## 10.4 Open Problems and Future Work

After answering *IMPL* and *EVAL* while summarizing the contents of this work, I want to present open problems, which should be solved to further

## 10. Conclusion and Future Work

**Figure 10.1.** Fuzzy target width for the rectpacking heuristic could result in a better scale measure if rectangle C could be placed in the first row (dashed rectangle) instead of the second row.



improve presented algorithms and preserve even more user intentions while using automatic layout.

### 10.4.1 Fuzzy Target Width for the Rectpacking Heuristic

Section 4.4 introduces the first step of the rectpacking heuristic as the width approximation step, which determines the target width of the packing. This target width serves as the maximum width of the packing throughout the algorithm.

Future work on this algorithm could introduce fuzziness to this target width such that it is no longer a fix value but rather a range the target width could be in. This could solve problems were the width approximation step slightly underestimates the required width and hence produces an ONO layout, as seen in Figure 10.1. Here, rectangle C does not fit in the first row by a very small margin and must hence be placed in the second row.

Even though this would solve some problems created by underestimating the target width, the effect will be limited since the required width is typically overestimated during the rectpacking heuristic.

### 10.4.2 Constraints for the Rectpacking Heuristic

As detailed in Section 7.2, I deem the “create a new row,” “create a new stack,” and “create a new subrow” constraints as one important part to add more control into the rectpacking algorithm. This control is essential since model order alone cannot control the line-breaks of a packing with a row and subrow structure. Hence, future work on rectangle packing could provide an implementation such that these line-break constraints can be set interactively to improve the control a user might have over a packing.

### 10.4.3 Deterministic Crossing Minimization

Even though model order makes most of the layered algorithm deterministic and controllable, as detailed in Chapter 5, crossing minimization in its implementation in ELK may still employ random decisions to prevent local minima.

If the model order crossing minimization pre-sorting step creates a layer permutation that results in a local minimum for edge crossings, the order of nodes and ports are randomly permuted to try a different starting configuration after the first two crossing minimization runs.

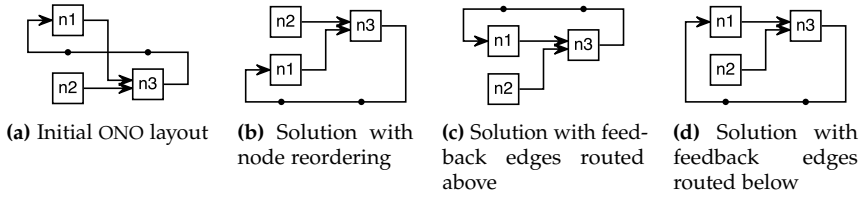
Hence, an open problem is whether one can deterministically find a different layer permutation that maintains most of the model order and avoids the local minimum. I.e., the next layer permutation to try should only minimally violate the model order and have a chance to avoid the local minimum. Additionally, it should be possible to enumerate all possible permutations. E.g., setting the number of crossing minimization runs for Figure 5.33b to 120 should enumerate all possible 120 port permutations on  $n_1$  in order of their model order violations and likeliness to result in a crossing minimal solution.

### 10.4.4 Dangling Nodes, Hyperedges, Feedback Edges, and Model Order

While model order can control most of the layout problem posed by SCCharts state machines, the position and routes of dangling nodes, hyperedges, and feedback edges cannot be fully controlled by model order for more general modeling languages such as Lingua Franca.

For hyperedges, a suitable edge segment ordering by model order has to be implemented. For feedback edges and dangling nodes one needs to find a better solution than a static decision, which cannot sufficiently capture secondary notation and does not allow to control all aspects of a layered layout. E.g., Figure 10.2a shows the current initial layout, created after model order pre-sorting. Using crossing minimization with unconstrained nodes and edges, this typically results in a reordering of  $n_1$  and  $n_2$ , as seen in Figure 10.2b. However, the potentially desired layouts in Figure 10.2c and

## 10. Conclusion and Future Work



**Figure 10.2.** Possible layouts with feedback edges and dangling nodes.

Figure 10.2d can only be created by constraining the order of nodes. This is the case since one can only compare dummy nodes to real nodes if both have connections to the previous layer, which is not the case for feedback edges and dangling nodes. E. g., in Figure 10.2 neither  $n1$ ,  $n2$ , nor  $n3$  can be compared to the dummy nodes marked with black dots.

One solution to the ONO layouts created by feedback edges and dangling nodes might be to do real crossing minimization or greedy post-processing and accept the resulting model order violation in Figure 10.2b. However, I suspect that the problem can be solved by determining the routes for edge the position of dangling nodes using neighboring edges to the currently unordered succeeding layer. E. g., by anticipating that routing the backward edge directly below  $n1$ , as seen in Figure 10.2a, has to create an edge crossing. This could potentially work by anticipating positions of the succeeding nodes and edges based on their model order, which needs further research.

### 10.4.5 Width Approximation for Component Packing

The packing of separate connected components is essentially a rectangle packing algorithm. Hence, it should implement the lessons learned from the simple heuristic and the rectpacking heuristic in terms of width approximation.

Similar to the region packing problem described in Chapter 4, the component packing problem requires a width approximation step that accounts for ordering and the existing edges to the outside. Currently, however, the target width is statically estimated. As an improvement, width approximation needs to take the concrete order of the components into

account. Moreover, placement constraints based on the edge direction need to be considered. Hence, I suggest solving this open problem if the actual aspect ratio of the packing is undesirable for future applications because of the static width approximation step.

### 10.4.6 Compaction for Component Packing

Another open problem for the packing of separate connected components is a model order aware post-compaction algorithm.

As seen in Figure 6.3b on page 207, considering model order for component packing may greatly increase the drawing size. Since the presented component packing algorithm only works based on the components bounds and the sides edges connect to, the drawing could be further compacted based on concrete node coordinates and edge routes using one-dimensional compaction [Rüe18]. Such a compacted drawing is depicted in Figure 10.3. Here, a two-dimensional<sup>1</sup> and model order aware compaction algorithm could compact the drawing by breaking the row structure without violating the model order reading direction. E. g., the rows containing S, W, NW, and NE were merged in Figure 10.3b compared to the original rows in Figure 10.3a. At the same time, the model order awareness prevents that NE could be moved to the top since it cannot be above NW, which blocks its vertical movement to preserve the reading direction.

While a compaction algorithm cannot remove the additional edge crossings created by the proposed model order compaction algorithm, it would still improve readability by increasing the scale measure of the packings. Therefore, this improvement together with an improved width approximation might be necessary for an insightful quantitative evaluation of the model order component packing algorithm, which is another open problem to tackle.

---

<sup>1</sup>Two-dimensional compaction could mean doing one-dimensional compaction twice in orthogonal directions.

## 10. Conclusion and Future Work

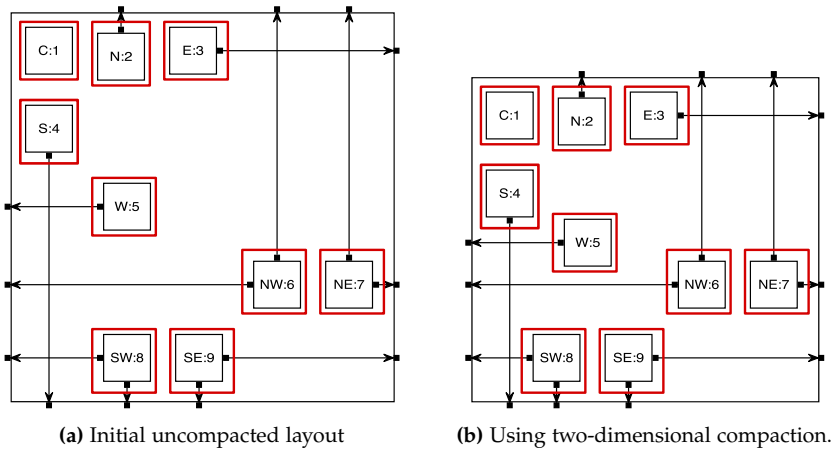


Figure 10.3. Compacting component packings

### 10.4.7 Implementing Model Order Aware Structure-Based Editing

Leveraging model order together with structure-based editing is currently only a conceptualized prototype that has neither proper implementation nor a proper evaluation.

As described in Chapter 8, structure-based editing may provide intention in form of diagram interaction such that users can directly express where a node or edge might be placed. I envision that one can use this interaction to insert a new element based on the model order and optional layout constraints or whitespace grouping (see Section 10.4.9) in a textual model.

Hence, model order aware structure-based editing could be implemented for further research on how to express model order in a more direct way.

### 10.4.8 Automatic Model Obfuscation

Section 9.4 describes finding realistic models of non-trivial size as one issue of finding and researching model order for modeling languages. This is due to the fact that good models are typically developed by industry and

## 10.4. Open Problems and Future Work

may contain trade secrets. Hence, such models must typically be obfuscated before giving them away to researchers. An automatic obfuscation system, e.g., for the languages SCCharts and Lingua Franca might solve this problem.

For Lingua Franca, obfuscation means that the reaction code must be deleted, which could be easily automated. For SCCharts, variables, transition guards, and transition effects must automatically be deleted or filled with strings of similar length if label placement algorithms should be researched.

Additionally, proper obfuscation for SCCharts and Lingua Franca requires automatic obfuscation of state and reactor names in a non-trivial way such that all potential dependencies between states or reactors remain recognizable. E.g., Figure 9.7 on page 259 does not give hints about the potential use of the state machine other than that “something might get active, inactive, interrupted, renewed, or disconnected.” However, it still gives hints on the desired ordering and symmetry in the layout.

Hence, model obfuscation is a non-trivial open question to work on to facilitate further research on layout algorithms for specific modeling languages.

### 10.4.9 Intention Detection via Text Processing

Intention cannot only be found in the ordering in the input model, i.e., the model order, but also in the naming of model elements as well as in line breaks or whitespace, which may create visual grouping in a textual model.

E.g., names such as “Source” or “Sink” imply that these should be placed accordingly, as seen in Figure 1.1 on page 3. Moreover, Lingua Franca developers reported that they would like to constrain ports numbered similar to “input1” and “input2” as their numbering suggests. Similarly, words such as “Increase” and “Decrease” suggest an ordering based on how people use these terms in natural language. Finally, whitespace between declarations, e.g., as in the textual model in Figure 5.58 on page 196 or in Figure 10.4, could be interpreted as grouping in the textual model, adding a second dimension to model order.

It is therefore an open question whether text processing neural networks

## 10. Conclusion and Future Work

**Figure 10.4.** The abbreviated main reactor of the Pac-Man Lingua Franca model. Here, whitespace and comments are used such that the Ghost reactors are grouped together. This grouping information could be used to improve the layout in addition to using model order.

```
main reactor {  
  ### Controller  
  controller = new GameController()  
  
  ### Model(s)  
  player = new Player(...)  
  
  # Ghosts  
  pinky = new Ghost(...)  
  blinky = new Ghost(...)  
  inky = new Ghost(...)  
  clyde = new Ghost(...)  
}
```

such as ChatGPT or other large language models can be used to detect ordering constraints based on naming in addition to using the model order. A second open question is whether whitespace and line-breaks in textual models can be used to create a more two-dimensional model order such that potential groupings can be identified automatically to increase control to illustrate desired secondary notation.

### 10.4.10 Configuring Layout via Machine Learning

The last open question I want to present tackles the potential complicated creation of a proper model order configuration and layout configuration for a modeling language.

One key problem that must be solved to use automatic layout in a real application is configuring the layout to the respective use-case. E. g., to find suitable model order configurations and layout configurations, as detailed in Section 9.4. Finding a suitable model order configuration requires a layout expert that is familiar with all available layout options and algorithms and a domain expert that knows how to use the language and what parts of a model should be visualized or highlighted. This is a tedious process, since the domain expert might want everything, everywhere, all at once, which is not always possible, while the layout expert needs to understand the needs of the domain. Multiple different options that are hard to grasp for humans, however, sound like a task designed for machine learning and may relate to



## 10.4. Open Problems and Future Work

Spönemann's evolutionary algorithm [SDH14a]. Such a layout configuration selection algorithm could work as follows in an IDE.

Whenever a user is presented with a diagram, the user might give feedback. This leaves two main problems: How should a user give feedback, and how can feedback be translated into different model order or layout configurations?

Let us assume the respective tool has a button to report an ONO layout. Just reporting that the diagram is undesirable is not enough since it remains unclear what exactly is wrong about it. The used feedback mechanism must be able to allow (preferably graphical) highlighting of the problem. I think that it would also be helpful to allow users to draw or show what they want instead and to translate that into layout options, layout constraints, or potentially into a different model order.

Moreover, I argue that just showing different alternative layouts might not be enough. Different alternatives might just introduce different ONO layouts. The selection of alternatives needs to include viable alternatives selected based on the problem the user had or the potentially sketched alternative, which seems to be a very hard problem.

Moreover, there might not only be one option<sup>2</sup> to create a specific layout. This leaves the open question how to select the correct options that would also work for other models.

Additionally, developers that employ a tool for real world applications might not like that their underlying layout algorithm might use their data. Hence, even an on-demand report button for ONO layouts or problems might not be usable by developers. Moreover, it might be necessary to obfuscate the models, as described in Section 10.4.8. However, even if a tool has an obfuscation mechanism, which renames all strings to different but sensible strings of a similar length, I do not think that the trust in such technology is high enough to employ it for confidential projects. However, such projects might hold the interesting models one needs to evaluate the used layout strategies, as detailed in Section 9.4.

Nevertheless, a learning layout configuration algorithm might be one option to truly understand the developer intention. Hence, this could be

---

<sup>2</sup>E. g., the ELK layered algorithm supports over 140 layout options, which could potentially all create the desired layout.

## 10. Conclusion and Future Work

an interesting further research focus to go beyond the capabilities of model order to capture intention and to create drawings that make WYSIWYG editors obsolete. Moreover, model order could still be used for automatic layout.

### **10.4.11 Evaluating more Modeling Languages**

As a final note on model order, I want to add that the evaluation of model order is incomplete.

Even though I thoroughly investigated SCCharts and Lingua Franca, the generalizations and statements about different modeling languages in Section 9.3 are mostly deduction and speculation rather than empirical evaluation. Hence, it could be fruitful to investigate and evaluate modeling languages that follow the text-first editing paradigm regarding their usage of model order.

# Acknowledgments

I thank everyone somehow involved in writing this thesis and I beg for forgiveness if I missed someone.

I want to thank my wife Janieke Bekasinski for her endless support throughout pandemic and other catastrophes. I want to thank her for listening to sometimes boring talks about automatic layout and for her feedback regarding layout of figures.

I want to thank my parents Silke and Torsten Domrös for her support, which allowed me to live a quite care-free student life and ultimately led to this thesis.

I also want to thank my first computer science teacher Sönke Schulmeister, who got me interested in computer science in the first place, Christian Motika, who introduced me into the real-time and embedded systems group, Reinhard von Hanxleden, who helped me refine the concept of model order during one particular coffee session while moving stones around, and all my colleagues in particular Alexander Schulz-Rosengarten that listened to model order and automatic layout more than once even though it is not interesting for them at all.

I also want to thank Edward A. Lee and Marten Lohstroh for encouraging and inviting me to visit Berkeley. Moreover, I want to thank them and Vincenzo Barbuto, Shaokei Lin, Francesco Paladino, and Efsane Soyer for their invaluable feedback regarding model order for Lingua Franca.

Getting real models and real developers to evaluate layout algorithms is hard and often not possible. Hence, I want to thank the Nis Wechselberg and the team at Scheidt & Bachmann for making time and supplying real world models to analyze as well as Khubaib Umer and Omer Majeed from Magnition.

Finally, I want to thank all friends and relatives for their support and the oncology team of the UKSH Kiel for fixing me up again.



# Publications

This thesis build and extends content already published in the following works and student theses.

## A.1 Relevant Publications

[DLH+21] Sören Domrös, Daniel Lucas, Reinhard von Hanxleden, and Klaus Jansen. “On order-preserving, gap-avoiding rectangle packing”. In: *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2021) - Volume 3: IVAPP*. INSTICC. SciTePress, 2021, pp. 38–49. ISBN: 978-989-758-488-6. DOI: 10.5220/0010186400380049

This is the first publication on model order to solve the region packing problem (Chapter 4) by introducing reading direction and packing constraints to strip packing and introducing a width approximation strategy for the rectangle packing problem.

[DLH+23] Sören Domrös, Daniel Lucas, Reinhard von Hanxleden, and Klaus Jansen. “Revisiting order-preserving, gap-avoiding rectangle packing”. In: *Computer Vision, Imaging and Computer Graphics Theory and Applications*. Cham: Springer International Publishing, 2023, pp. 183–205. ISBN: 978-3-031-25477-2. DOI: 10.1007/978-3-031-25477-2\_9

This publication expands on the work of Domrös et al. [DLH+21] by suggesting algorithm improvements and clearly stating the constraints required for optimal region packing.

[DH22] Sören Domrös and Reinhard von Hanxleden. “Preserving order during crossing minimization in Sugiyama layouts”. In: *Proceedings of the 17th Interna-*

## A. Publications

*tional Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2022) - Volume 3: IVAPP*. INSTICC. SciTePress, 2022, pp. 156–163. ISBN: 978-989-758-555-5. DOI: 10.5220/0010833800003124

This is the first publication on model order for layered layouts that focuses on tie-breaking strategies for crossing minimization, as presented in Chapter 5. A technical report [DH21] expands on the different model order crossing minimization strategies that might be employed and presents additional SCCharts examples.

- [DRv23] Sören Domrös, Max Riepe, and Reinhard von Hanxleden. “Model order in Sugiyama layouts”. In: *Proceedings of the 18th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2023) - Volume 3: IVAPP*. INSTICC. SciTePress, 2023, pp. 77–88. ISBN: 978-989-758-634-7. DOI: 10.5220/0011656700003417

This paper expands and summarizes model order for layered layouts by including strategies for cycle breaking, layer assignment, and crossing minimization that can be utilized as a tie-breaker or a constraint. Moreover, it introduces the problem of textual ordering conventions for different semantic elements on the example of Lingua Franca and sketches the concept of group model order.

- [PDS+23] Jette Petzold, Sören Domrös, Connor Schönberger, and Reinhard von Hanxleden. “An interactive graph layout constraint framework”. In: *Proceedings of the 18th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2023) - Volume 3: IVAPP*. With accompanying poster. INSTICC. SciTePress, 2023, pp. 240–247. ISBN: 978-989-758-634-7. DOI: 10.5220/0011803000003417

This paper introduces diagram interactive constraints framework presented in Chapter 7 and lays the groundwork for how model order and layout constraints interact.

- [DH24c] Sören Domrös and Reinhard von Hanxleden. “Diagram control and model order for Sugiyama layouts”. In: *Proceedings of the 9th International Conference on the Theory and Application of Diagrams, DIAGRAMS '24*. Vol. 14981. LNCS. Springer Nature Switzerland, 2024, pp. 76–83. DOI: 10.1007/978-3-031-71291-3\_6

## A.2. Relevant Advised Theses

This paper argues how stability, secondary notation, aesthetic criteria, the mental map, and the desire to control the layout work together and are influenced by model order, as described in Chapter 3. Domrös et al. [DH24b] presents a long version which supplies additional insights including lessons learned and a guidebook on how to extract model order configurations for additional modeling languages that employ modeling pragmatics [HLF+22].

[DH24a] Sören Domrös and Reinhard von Hanxleden. *Determining Sugiyama topology with model order*. Poster at the 32nd International Symposium on Graph Drawing and Network Visualization, TU Wien, Vienna, Austria. Sept. 2024. doi: 10.4230/LIPIcs.GD.2024.48

This extended abstract with an accompanying poster argues how one can solve the topological phases of the layered algorithm efficiently by using model order as a constraint, as detailed in Section 5.8.

## A.2 Relevant Advised Theses

[Pet19] Jette Petzold. “Intentional Layout in Sprotty Diagrams: Defining User Interaction”. Bachelor Thesis. Kiel University, Department of Computer Science, Sept. 2019. URL: <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jet-bt.pdf>

Petzold and Schönberger [Sch19a] implemented the first sketch for interactive layered diagrams. This led to the interactive layout constraint framework [PDS+23] presented in Chapter 7.

[Sch19a] Connor Schönberger. “Intentional Layout in Sprotty Diagrams: Reevaluating Introduced Constraints”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cos-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2019  
See above.

[Car20] Niklas Carstensen. “Interactive tree layout”. Bachelor thesis. Kiel University, Department of Computer Science, Oct. 2020. URL: <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jnc-bt.pdf>

## A. Publications

The work of Carstensen generalizes the work of Petzold [Pet19] and Schönberger [Sch19a] and adopts the interactive constraints framework for tree layouts.

- [Rie22] Max Riepe. “Model order and cycle breaking in SCCharts”. Bachelor Thesis. Kiel University, Department of Computer Science, Mar. 2022. URL: <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mwr-bt.pdf>

The advised thesis by Riepe first dived into model order cycle breaking and evaluated cycle breaking strategies for SCCharts as presented in Chapter 5. This led to the cycle breaking evaluation presented by Domrös et al. [DRv23] and was continued as a research project regarding cycle breaking for Lingua Franca.

- [Jöh22] Felix Jöhnk. “Structure-based editing for SCCharts”. Master thesis. Kiel University, Department of Computer Science, 2022. URL: <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fej-mt.pdf>

In this thesis, Jöhnk introduced structure-based editing for SCCharts, as presented in a generalized form in Chapter 8. Moreover, this led me to investigate how diagram interaction can be translated into model order as a way to serialize intention.

- [Rie24] Max Riepe. “Group model order for sugiyama layouts”. Master Thesis. Kiel University, Department of Computer Science, Mar. 2024. URL: <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mwr-mt.pdf>

This advised thesis by Riepe generalizes group model order for different semantic elements that are bound by textual conventions. This thesis extends on an earlier research project of Riepe that explored cycle breaking for Lingua Franca by evaluating cycle breaking as well as crossing minimization strategies for Lingua Franca. Moreover, this work explores how cycle breaking influences crossing minimization quantitatively based on potential model order strategies for Lingua Franca, as presented in Section 5.9.



## A.3 Additional Related Publications

Additionally, to the publications and theses mentioned above this work builds on the following publications I contributed to.

[DH21] Sören Domrös and Reinhard von Hanxleden. *Preserving order during crossing minimization in Sugiyama layouts*. Technical Report 2103. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Nov. 2021

This technical report expands on model order for crossing minimization [DH22].

[HLF+22] Reinhard von Hanxleden et al. “Pragmatics twelve years later: a report on Lingua Franca”. In: *11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Vol. 13702. Lecture Notes in Computer Science. Springer. Rhodes, Greece, Oct. 2022, pp. 60–89. DOI: 10.1007/978-3-031-19756-7\_5

This paper introduces how a visualization for Lingua Franca together with its textual model should be used in an IDE. This also includes a small section on automatic layout, which describes how model order influenced the layout of Lingua Franca at that time.

[DHS+23] Sören Domrös et al. *The Eclipse Layout Kernel*. 2023. DOI: 10.48550/arXiv.2311.00533

This arXiv paper presents the structure of the layout library ELK, which implements all layout strategies discussed in this work.

[DH24b] Sören Domrös and Reinhard von Hanxleden. *Diagram control and model order for sugiyama layouts*. 2024. DOI: 10.48550/arXiv.2406.11393

This arXiv paper expands on model order and control on a meta-level by giving more insights into how model order should be employed and how model order configurations for modeling languages can be found.

[KRD+24] Maximilian Kasperowski, Niklas Rentz, Sören Domrös, and Reinhard von Hanxleden. “KIELER: a text-first framework for automatic diagramming of complex systems”. In: *Proceedings of the 14th International Conference on the Theory*

## A. Publications

*and Application of Diagrams, DIAGRAMS '24*. Vol. 14981. LNCS. Springer Nature Switzerland", 2024, pp. 402–418. DOI: 10.1007/978-3-031-71291-3\_33

In this work, Kasperowski et al. present and evaluate the text-first editing paradigm for modeling languages, which I focus on in this work to use model order effectively. Additionally, this paper argues that model order brings control into the text-first diagram approach.

[KDH24] Maximilian Kasperowski, Sören Domrös, and Reinhard von Hanxleden. *The Eclipse Layout Kernel*. Poster at the 32nd International Symposium on Graph Drawing and Network Visualization, TU Wien, Vienna, Austria. Sept. 2024. DOI: 10.4230/LIPICs.GD.2024.56

This extended abstract with an accompanying poster gives a short overview over the Eclipse Layout Kernel including where model order strategies are used in the algorithm.

# Abbreviated Models

This appendix holds a selection of abbreviated Lingua Franca models to show their model order.

## B.1 load\_balancer

```
reactor load_balancer<T1, T2> (...) {  
  input[n_inputs] in_request:T1;  
  output[n_inputs] out_response:T2;  
  
  output[n_outputs] out_request:T1;  
  input[n_outputs] in_response:T2;  
  
  logical action send_out_request(0);  
  logical action send_out_response(0);  
  
  reaction (startup) {= =}  
  
  reaction (send_out_request) -> out_request, send_out_request {= =}  
  
  reaction (in_request) -> send_out_request {= =}  
  
  reaction (send_out_response) -> out_response, send_out_response {= =}  
  
  reaction (in_response) -> send_out_response {= =}  
  
  reaction (shutdown) {= =}  
}
```

## B. Abbreviated Models

### B.2 SleepingBarber

*/\*\**

- \* Upon startup, the barber goes to sleep and is woken up by the arrival of a customer. After serving each customer,*
  - \* the barber checks the waiting room for the next customer. If there is no customer waiting, the barber goes back to sleep.*
  - \* Customers arrive independently from each other at random times. If the barber is sleeping, the customer gets served right*
  - \* away. Otherwise, if there is room in the waiting room, the customer waits. If the waiting room is full, the customer*
  - \* goes away and returns a random amount of time later. Execution ends when all customers have been served.*
- \*/*

```
main reactor(...) {  
    factory = new CustomerFactory(...)  
    room = new WaitingRoom(...)  
    barber = new Barber(...)  
    customers = new[num_customers] Customer()  
  
    factory.send_customer -> room.customer_enters  
    room.full -> customers.room_full  
    room.wait -> customers.wait  
    room.barber_leaves_with_customer -> barber.enter  
    barber.next -> room.barber_arrives  
    barber.start_cutting -> customers.start_cutting  
    barber.done_cutting -> customers.done_cutting  
    customers.done -> factory.customer_done  
    customers.returned -> factory.customer_returned  
}
```

### B.3 reactionOrder

```
main reactor {  
    logical action a:char*;  
    physical action b:char*;  
  
    reaction (startup) -> a {= =}  
    reaction (a) -> b {= =}  
    reaction (b) -> a {= =}  
}
```

## B.4 Chrono

```

reactor ChronoLogic {
    ...
    // time output
    output m:int;
    output s:int;
    output d:int;

    // time state
    state dState:int(0);
    state sState:int(0);
    state mState:int(0);

    ...

    reaction(dTimer) -> d, s, m {=
        if (self->ststState) {
            self->dState = (self->dState + 1) % 100;
            if (self->dState == 0) {
                self->sState = (self->sState + 1) % 60;
                if (self->sState == 0) {
                    self->mState = (self->mState + 1) % 60;
                }
            }
            // Only create outputs for changes
            if_set(d, self->dState);
            if_set(s, self->sState);
            if_set(m, self->mState);
        }
    =}
}

...

reactor PrintOutput {
    input m:int;
    input s:int;
    input d:int;

    reaction(m) {=
        printf("m_=%d, ", m->value);
    =}
}

```

## B. Abbreviated Models

```
reaction(s) {=  
    printf("s_=%d\n", s->value);  
=}  
  
reaction(d) {=  
    printf("d_=%d\n", d->value);  
=}  
}  
  
main reactor Chrono {  
    ...  
    chrono.m -> out.m;  
    chrono.s -> out.s;  
    chrono.d -> out.d;  
}
```

# Acronyms

## B. Abbreviated Models

<i>DAR</i>	desired aspect ratio
<i>DSL</i>	Domain Specific Language
<i>ELK</i>	Eclipse Layout Kernel
<i>FPGA</i>	Field Programmable Gate Array
<i>IDE</i>	Integrated Development Environment
<i>KIELER</i>	Kiel Integrated Environment for Layout Eclipse Rich Client
<i>MDE</i>	Model-Driven Engineering
<i>ONO</i>	Obviously Non-Optimal
<i>OYES</i>	Obvious Yet Easily Superior
<i>S&amp;B</i>	Scheidt & Bachmann System Technik GmbH
<i>SM</i>	scale measure
<i>STPA</i>	System Theoretic Process Analysis
<i>UML</i>	Unified Modeling Language



*VS Code*

Visual Studio Code

*WYSIWYG*

What You See Is What You Get

*XML*

Extensible Markup Language



# List of Figures

1.1	The Source-Through Lingua Franca model . . . . .	3
1.2	The dependency structure of the dissertation chapters. . . . .	4
2.1	The motor SCChart in the KIELER VS Code. . . . .	8
2.2	The motor SCCharts with an abbreviated textual model. . . . .	11
2.3	The Sleeping Barber Lingua Franca model [MCL+24]. . . . .	14
2.4	The AccelerometerDisplay Lingua Franca model. . . . .	14
3.1	The motor SCChart visualizes stability. . . . .	25
3.2	SCChart by Motika. . . . .	28
3.3	An example for a modeling language. . . . .	30
4.1	Stacking regions increases the scale measure. . . . .	36
4.2	Whitespace elimination for a simple model. . . . .	37
4.3	An unconstrained region packing might hinder whitespace elimination. . . . .	37
4.4	The coordinate system has its origins in the top-left corner. . . . .	37
4.5	The row structure of a packing. . . . .	39
4.6	Rectangle packing and without row-dominant elements . . . . .	40
4.7	Rectangle packing with unloading constraints. . . . .	42
4.8	Strip packing with precedence constraints introduces vertical placement constraints for all rectangles. . . . .	43
4.9	Treemap visualizations allow rectangles with a more fluid shape. . . . .	44
4.10	The static width approximation of the box algorithm visualized. . . . .	45
4.11	The simple compaction algorithm only creates stacks of regions. . . . .	47
4.12	Disregarding the model order can produce a more compact drawing. . . . .	50
4.13	The four candidate positions for placing regions . . . . .	51

## List of Figures

4.14	The greedy nature of the width approximation step may overapproximate the target width. . . . .	51
4.15	Width approximation of the running example. . . . .	52
4.16	Greedy width approximation of the worst case example compared to the final packing. . . . .	53
4.17	Row placement and block grouping during placement. . . . .	55
4.18	The packing utilizes the four available region positions during the compaction step. . . . .	56
4.19	Preferably placing regions left-to-right may waste space. . . . .	58
4.20	The final packing after the compaction step emphasizes the left-to-right reading direction. . . . .	61
4.21	Worst case runtime for the compaction step. . . . .	61
4.22	Worst case space example. . . . .	64
4.23	The fixed row height after placement may result in a bad packing. . . . .	65
4.24	Target width revision visualized. . . . .	66
4.25	Row height revision. . . . .	68
4.26	The problem of row height revision visualized. . . . .	68
4.27	Visualization of the whitespace elimination step using the row structure. . . . .	70
4.28	Whitespace can also be eliminated to let a packing fit into a desired aspect ratio. . . . .	71
4.29	Whitespace elimination without a row structure visualized. . . . .	73
4.30	Considering row-dominant elements lowers the scale measure. . . . .	79
4.31	Rectpacking evaluation of scale measure and aspect ratio. . . . .	82
4.32	Scale measure and aspect ratio of 372 SCCharts models. . . . .	85
4.33	The kh_mutex SCChart from the Railway Project '14 [SMS+15]. . . . .	86
4.34	ONO region stealing visualized. . . . .	88
4.35	Greedy block splitting creates ONO packings. . . . .	88
4.36	A different block creation improves the packing. . . . .	89
4.37	ONO region alignment with and without whitespace elimination. . . . .	89
5.1	ONO layouts for the cycle breaking, layer assignment, and crossing minimization step. . . . .	94

5.2	The five phases of ELK Layered. . . . .	96
5.3	The updated ELK development timeline. . . . .	99
5.4	The ELK Layered processors. . . . .	100
5.5	STPA control structure with downward layout direction of an autonomous vehicle. . . . .	102
5.6	The SleepingBarber Lingua Franca model with ONO segment routing. . . . .	104
5.7	The problem of random layout decisions visualized. . . . .	105
5.8	Dot layered layout example with constraints. . . . .	106
5.9	Dot uses the node model order for vertical ordering as a tie-breaker. . . . .	109
5.10	Dot uses depth-first search to determine the flow of a graph with the node order as the visiting order. . . . .	109
5.11	A sequence diagram with life-lines Alice, Bob, and Charlie. . . . .	110
5.12	Strict model order cycle breaking visualized based on a total linear order. . . . .	113
5.13	The greedy cycle breaker and the model order cycle breaker. . . . .	114
5.14	The greedy cycle breaking algorithm randomly makes wrong decisions. . . . .	116
5.15	The greedy model order cycle breaking cannot determine the first node. . . . .	117
5.16	Depending on the visiting order, a depth-first or breadth-first cycle breaker may create Figure 5.16a or Figure 5.16b. . . . .	118
5.17	Layer assignment as a tie-breaker. . . . .	120
5.18	Model order layer assignment without a consistent order. . . . .	121
5.19	Enforced model order layer assignment. . . . .	121
5.20	Strict model order layering potentially affects the width of a layout. . . . .	123
5.21	The CAPTN STPA relationship diagram. . . . .	124
5.22	Depth-first model order layering on an example model. . . . .	129
5.23	Following the edge order or node order may lead to different orderings. . . . .	130
5.24	Barycenter heuristic visualized. . . . .	132
5.25	Find a local minimum during crossing minimization depends on the initial node and port order. . . . .	133

## List of Figures

5.26	How the initial permutation influences crossing minimization.	134
5.27	The three crossing minimization pre-sorting strategies: prefer-edges, nodes-and-edges, and prefer-nodes. . . . .	137
5.28	Illustration of “No connection”. . . . .	138
5.29	Illustration of “transitive order”. . . . .	139
5.30	An example for model order port pre-ordering to prevent ONO layouts. . . . .	142
5.31	Feedback edge port sorting illustrated. . . . .	144
5.32	Hyperedges need to consider all sources and all targets to make desired decisions based on model order. . . . .	144
5.33	Stability when enforcing the node order. . . . .	146
5.34	The PacMan Lingua Franca model with a marked blinky reactor.	148
5.35	The PacMan Lingua Franca model with enforced node order. .	149
5.36	Enforcing node and edge model order creates additional crossings. . . . .	150
5.37	Ports or nodes align themselves based on model order. . . .	153
5.38	Group model order in Lingua Franca. . . . .	154
5.39	In the ACASXu2 Lingua Franca model. . . . .	156
5.40	Example model for a strongly connected components cycle breaker. . . . .	159
5.41	The efficient model order layering algorithm visualized. . . .	162
5.42	An example for the efficient strict model order topology algorithm. . . . .	163
5.43	ONO models using the efficient model order topology algo- rithm. . . . .	163
5.44	The obfuscated piston SCChart used by Riepe [Rie22] as one of the stimuli. . . . .	165
5.45	Study 1: The average performance and the resulting ranking per graph. . . . .	167
5.46	Study 3: Number of backward edges normalized by the mean number of backward edges [Rie24]. . . . .	172
5.47	Study 3: Number of edge crossings normalized by the mean number of edge crossings [Rie24]. . . . .	172
5.48	Study 3: Time to layout in milliseconds [Rie24]. . . . .	172
5.49	Study 3: Aspect ratio of the resulting drawing [Rie24]. . . .	172

5.50	Lingua Franca editor and diagram configuration. . . . .	174
5.51	Study 3: Normalized edge crossings using the depth-first node order cycle breaker [Rie24]. . . . .	176
5.52	Study 3: Edge crossings normalized by edge crossings in the respective cycle breaking category [Rie24]. . . . .	176
5.53	Changes of the 54 evaluated models . . . . .	180
5.54	The TrainDoor Lingua Franca model. . . . .	188
5.55	The SimpleChat Lingua Franca model [JK24] with ChatHandler a expanded and ChatHandler b collapsed. . . . .	190
5.56	The load_balancer Lingua Franca model. . . . .	192
5.57	The SleepingBarber Lingua Franca model. . . . .	194
5.58	The abbreviated textual model of the ACASXu2 main reactor. .	196
5.59	The ACASXu2 Lingua Franca model. . . . .	197
5.60	The reactionOrder Lingua Franca model. . . . .	199
5.61	Variant of SCC + GMOF with backward edge below. . . . .	200
6.1	The Sleeping Barber Lingua Franca model [MCL+24] layouted without separate component model order. . . . .	205
6.2	The Cutter Lingua Franca model [Lee24b] layouted with and without considering separate connected components. . . . .	206
6.3	Component packing and model order component packing compared. . . . .	207
6.4	Component placement needs to account for the connection edges. . . . .	210
6.5	Maximum number of edge crossings using model order component packing. . . . .	211
7.1	The full control model order configuration cannot create all layerings. . . . .	216
7.2	Interaction during interactive layout. . . . .	217
7.3	Interactively setting constraints requires a stable layout. . .	218
7.4	Releasing n1 in layer 1 on position 0 updates the position constraints of n2 and n3. . . . .	221
7.5	Releasing n3 inside an existing chain makes n3 part of the relative constraint chain now consisting of n1, n3, and n2. . .	221

## List of Figures

9.1	A variant motor SCChart using the legacy layout, the new default strategy, and the full control strategy. . . . .	233
9.2	Using the greedy cycle breaker for the erroneous motor SC-Chart without additional constraints . . . . .	234
9.3	The obfuscated Backhoe SCChart. . . . .	235
9.4	The PacMan Lingua Franca model. . . . .	241
9.5	The PacMan Lingua Franca model with full control crossing minimization and a highlighted hyperedge. . . . .	245
9.6	The Chrono Lingua Franca model [SH24] inspired by the chronometer of Colaço et al. [CPP05]. . . . .	248
9.7	An obfuscated SCChart by S&B. . . . .	259
10.1	Fuzzy target width for the rectpacking heuristic. . . . .	268
10.2	Possible layouts with feedback edges and dangling nodes. .	270
10.3	Compacting component packings . . . . .	272
10.4	The abbreviated main reactor of the PacMan Lingua Franca model. . . . .	274



# List of Tables

2.1	Different elements of an SCChart . . . . .	11
2.2	Different elements of a Lingua Franca model . . . . .	13
4.1	Layout problems per depth $d$ in the used SCCharts models. .	84
5.1	Study 1: Confidence of the participants of study 1 on the topic of automatic graph drawing on a five-point Likert scale.	166
5.2	Study 1: Importance of aesthetic criteria . . . . .	168
5.3	Overview and encoding of the evaluated algorithmic alterna- tives. . . . .	178
5.4	Edge crossings and graph order violations for 54 non-trivial SCChart models. . . . .	179



# List of Algorithms

1	Greedy Width Approximation Heuristic . . . . .	49
2	Simple placement algorithm for the rectpacking heuristic [DLH+23]. . . . .	54
3	Compaction algorithm for the rectpacking heuristic [DLH+23].	57
4	The strict model order cycle breaker . . . . .	112
5	Greedy (Model Order) Cycle Breaker . . . . .	116
6	Breadth-first model order layering . . . . .	122
7	Breadth-first model order layerer by post-processing [DRv23] .	125
8	Depth-first model order layerer . . . . .	127
9	Crossing minimization without model order . . . . .	132
10	Crossing minimization with model order . . . . .	135
11	Strongly Connected Component Model Order Cycle Breaker .	158
12	Model order component packing . . . . .	208



# Bibliography

- [ABI06] John Augustine, Sudarshan Banerjee, and Sandy Irani. “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)*. Cambridge, Massachusetts, USA: ACM, 2006, pp. 180–189. ISBN: 1-59593-452-9. DOI: 10.1145/1148109.1148139.
- [ADD+20] Reyhan Ahmed, Felice De Luca, Sabin Devkota, Stephen Kobourov, and Mingwei Li. “Graph drawing via gradient descent,  $(GD)^2$ ”. In: *Graph Drawing and Network Visualization*. Cham: Springer International Publishing, 2020, pp. 3–17. DOI: 10.1007/978-3-030-68766-3\_1.
- [AH23] Patrizio Angelini and Reinhard von Hanxleden. *Graph drawing and network visualization: 30th international symposium, gd 2022, tokyo, japan, september 13–16, 2022, revised selected papers*. Vol. 13764. Springer Nature, 2023. DOI: 10.1007/978-3-031-22203-0.
- [BHV00] Mark Bruls, Kees Huizing, and Jarke J Van Wijk. “Squarified treemaps”. In: *Data visualization 2000*. Springer, 2000, pp. 33–42.
- [BK02] Ulrik Brandes and Boris Köpf. “Fast and simple horizontal coordinate assignment”. In: *Proceedings of the 9th International Symposium on Graph Drawing (GD ’01)*. Ed. by Petra Mutzel, Michael Jünger, and Sebastian Leipert. Vol. 2265. LNCS. Springer, 2002, pp. 33–36. ISBN: 978-3-540-43309-5. DOI: 10.1007/3-540-45848-4.
- [BP90] Karl-Friedrich Böhringer and Frances Newbery Paulisch. “Using constraints to achieve stability in automatic graph layout algorithms”. In: *Proceedings of the SIGCHI Conference on Human*

## Bibliography

- Factors in Computing Systems*. Seattle, Washington, USA: ACM, 1990, pp. 43–51. ISBN: 0-201-50932-6. DOI: 10.1145/97243.97250.
- [Car20] Niklas Carstensen. “Interactive tree layout”. Bachelor thesis. Kiel University, Department of Computer Science, Oct. 2020. URL: <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jnc-bt.pdf>.
- [CLP24] Arthur Clavière, Edward A. Lee, and Claire Pagetti. *ACASXu2 lingua franca model*. <https://github.com/lf-lang/playground-lingua-franca/blob/main/examples/Python/src/acas/ACASXu2.lf>. Accessed: 2024-12-06. 2024.
- [Coh13] Neil Cohn. *The visual language of comics: introduction to the structure and cognition of sequential images*. A&C Black, 2013.
- [CPP05] Jean-Louis Colaço, Bruno Pagano, and Marc Pouzet. “A conservative extension of synchronous data-flow with State Machines”. In: *ACM International Conference on Embedded Software (EMSOFT’05)* (Jersey City, NJ, USA). Jersey City, NJ, USA: ACM, Sept. 2005, pp. 173–182. DOI: 10.1145/1086228.1086261.
- [DD92] Kathryn A. Dowsland and William B. Dowsland. “Packing problems”. In: *European Journal of Operational Research* 56.1 (1992), pp. 2–14. ISSN: 0377-2217. DOI: [https://doi.org/10.1016/0377-2217\(92\)90288-K](https://doi.org/10.1016/0377-2217(92)90288-K).
- [DH21] Sören Domrös and Reinhard von Hanxleden. *Preserving order during crossing minimization in Sugiyama layouts*. Technical Report 2103. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Nov. 2021.
- [DH22] Sören Domrös and Reinhard von Hanxleden. “Preserving order during crossing minimization in Sugiyama layouts”. In: *Proceedings of the 17th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2022) - Volume 3: IVAPP*. INSTICC. SciTePress, 2022, pp. 156–163. ISBN: 978-989-758-555-5. DOI: 10.5220/00108338000003124.

- [DH24a] Sören Domrös and Reinhard von Hanxleden. *Determining Sugiyama topology with model order*. Poster at the 32nd International Symposium on Graph Drawing and Network Visualization, TU Wien, Vienna, Austria. Sept. 2024. DOI: 10.4230/LIPIcs.GD.2024.48.
- [DH24b] Sören Domrös and Reinhard von Hanxleden. *Diagram control and model order for sugiyama layouts*. 2024. DOI: 10.48550/arXiv.2406.11393.
- [DH24c] Sören Domrös and Reinhard von Hanxleden. “Diagram control and model order for Sugiyama layouts”. In: *Proceedings of the 9th International Conference on the Theory and Application of Diagrams, DIAGRAMS '24*. Vol. 14981. LNCS. Springer Nature Switzerland, 2024, pp. 76–83. DOI: 10.1007/978-3-031-71291-3\_6.
- [DHS+23] Sören Domrös, Reinhard von Hanxleden, Miro Spönemann, Ulf Rüegg, and Christoph Daniel Schulze. *The Eclipse Layout Kernel*. 2023. DOI: 10.48550/arXiv.2311.00533.
- [Dij68] Edsger W. Dijkstra. “Co-operating sequential processes”. English. In: *Programming languages : NATO Advanced Study Institute : lectures given at a three weeks Summer School held in Villard-le-Lans, 1966 / ed. by F. Genuys*. Academic Press Inc., 1968, pp. 43–112. DOI: 10.1007/978-1-4757-3472-0\_2.
- [DLB22] Giuliano De Carlo, Philip Langer, and Dominik Bork. “Advanced visualization and interaction in GLSP-based web modeling: realizing semantic zoom and off-screen elements”. In: *Proceedings of the 25th International Conference on Model Driven Engineering Languages and Systems. MODELS '22*. Montreal, Quebec, Canada: Association for Computing Machinery, 2022, pp. 221–231. ISBN: 9781450394666. DOI: 10.1145/3550355.3552412.
- [DLH+21] Sören Domrös, Daniel Lucas, Reinhard von Hanxleden, and Klaus Jansen. “On order-preserving, gap-avoiding rectangle packing”. In: *Proceedings of the 16th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2021) - Volume 3: IVAPP*. INSTICC.

## Bibliography

- SciTePress, 2021, pp. 38–49. ISBN: 978-989-758-488-6. DOI: 10.5220/0010186400380049.
- [DLH+23] Sören Domrös, Daniel Lucas, Reinhard von Hanxleden, and Klaus Jansen. “Revisiting order-preserving, gap-avoiding rectangle packing”. In: *Computer Vision, Imaging and Computer Graphics Theory and Applications*. Cham: Springer International Publishing, 2023, pp. 183–205. ISBN: 978-3-031-25477-2. DOI: 10.1007/978-3-031-25477-2\_9.
- [DMW09] Tim Dwyer, Kim Marriott, and Michael Wybrow. “Dunnart: a constraint-based network diagram authoring tool”. In: *Revised Papers of the 16th International Symposium on Graph Drawing (GD ’08)*. Vol. 5417. LNCS. Springer, 2009, pp. 420–431. ISBN: 978-3-642-00218-2. DOI: 10.1007/978-3-642-00219-9.
- [DMX13] Jefferson LM Da Silveira, Flávio K Miyazawa, and Eduardo C Xavier. “Heuristics for the strip packing problem with unloading constraints”. In: *Computers & operations research* 40.4 (2013), pp. 991–1003. DOI: 10.1016/j.cor.2012.11.003.
- [DRv23] Sören Domrös, Max Riepe, and Reinhard von Hanxleden. “Model order in Sugiyama layouts”. In: *Proceedings of the 18th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2023) - Volume 3: IVAPP*. INSTICC. SciTePress, 2023, pp. 77–88. ISBN: 978-989-758-634-7. DOI: 10.5220/0011656700003417.
- [Dwy09] Tim Dwyer. “Scalable, versatile and simple constrained graph layout”. In: *Computer graphics forum*. Vol. 28. 3. 2009, pp. 991–998. DOI: 10.1111/j.1467-8659.2009.01449.x.
- [DXM14] Jefferson LM Da Silveira, Eduardo C Xavier, and Flávio K Miyazawa. “Two-dimensional strip packing with unloading constraints”. In: *Discrete Applied Mathematics* 164 (2014), pp. 512–521. DOI: 10.1016/j.endm.2011.05.018.
- [EK86] Peter Eades and David Kelly. “Heuristics for reducing crossings in 2-layered networks”. In: *Ars Combinatoria* 21 (1986), pp. 89–98.



- [ELS93] Peter Eades, Xuemin Lin, and W. F. Smyth. “A fast and effective heuristic for the feedback arc set problem”. In: *Information Processing Letters* 47.6 (1993), pp. 319–323. ISSN: 0020-0190. DOI: 10.1016/0020-0190(93)90079-0.
- [ES90] Peter Eades and Kozo Sugiyama. “How to draw a directed graph”. In: *Journal of Information Processing* 13.4 (1990), pp. 424–437. ISSN: 0387-6101. DOI: 10.1109/WVL.1989.77035.
- [GHM+14] Carsten Gutwenger, Reinhard von Hanxleden, Petra Mutzel, Ulf Rüegg, and Miro Spönemann. “Examining the compactness of automatic layout algorithms for practical diagrams”. In: *Proceedings of the Workshop on Graph Visualization in Practice (GraphViP '14)*. 2014, pp. 42–52.
- [GJ83] Michael R. Garey and David S. Johnson. “Crossing number is NP-complete”. In: *SIAM Journal on Algebraic and Discrete Methods* 4.3 (1983), pp. 312–316. DOI: 10.1137/0604033.
- [GKN+93] Emden R. Gansner, Eleftherios Koutsofios, Stephen C. North, and Kiem-Phong Vo. “A technique for drawing directed graphs”. In: *Software Engineering* 19.3 (1993), pp. 214–230. DOI: 10.1109/32.221135.
- [GN00] Emden R. Gansner and Stephen C. North. “An open graph visualization system and its applications to software engineering”. In: *Software—Practice and Experience* 30.11 (2000), pp. 1203–1234. ISSN: 00380644. DOI: 10.1002/1097-024X(200009)30:11<1203::AID-SPE338>3.0.CO;2-N.
- [Gra92] Winfried Graf. “Constraint-based graphical layout of multi-model presentations”. In: *Advanced Visual Interfaces-Proceedings Of The International Workshop Avi'92*. Vol. 36. World Scientific. 1992, p. 365. DOI: 10.22028/D291-24845.
- [GSS+24] Lena Grimm, Steven Smyth, Alexander Schulz-Rosengarten, Reinhard von Hanxleden, and Marc Pouzet. “From lustre to graphical models and sccharts”. In: *ACM Trans. Embed. Comput. Syst.* 23.5 (2024), 66:1–66:28. DOI: 10.1145/3544973. URL: <https://doi.org/10.1145/3544973>.

## Bibliography

- [Har87] David Harel. “Statecharts: A visual formalism for complex systems”. In: *Science of Computer Programming* 8.3 (June 1987), pp. 231–274. DOI: 10.1016/0167-6423(87)90035-9.
- [HCR+91] Nicolas Halbwachs, Paul Caspi, Pascal Raymond, and Daniel Pilaud. “The synchronous data flow programming language LUSTRE”. In: *Proceedings of the IEEE* 79.9 (Sept. 1991), pp. 1305–1320.
- [HDM+14] Reinhard von Hanxleden, Björn Duderstadt, Christian Motika, Steven Smyth, Michael Mendler, Joaquín Aguado, Stephen Mercer, and Owen O’Brien. “SCCharts: Sequentially Constructive Statecharts for safety-critical applications”. In: *Proc. ACM SIGPLAN Conference on Programming Language Design and Implementation (PLDI ’14)*. Edinburgh, UK: ACM, June 2014, pp. 372–383. DOI: 10.1145/2594291.2594310.
- [HEH+13] Weidong Huang, Peter Eades, Seok-Hee Hong, and Chun-Cheng Lin. “Improving multiple aesthetics produces better graph drawings”. In: *Journal of Visual Languages & Computing* 24.4 (2013), pp. 262–272. ISSN: 1045-926X. DOI: <https://doi.org/10.1016/j.jvlc.2011.12.002>.
- [HHE07] Weidong Huang, Seok-Hee Hong, and Peter Eades. “Effects of Sociogram Drawing Conventions and Edge Crossings in Social Network Visualization.” In: *J. Graph Algorithms Appl.* 11.2 (2007), pp. 397–429. DOI: 10.7155/jgaa.00152.
- [HLF+22] Reinhard von Hanxleden, Edward A. Lee, Hauke Fuhrmann, Alexander Schulz-Rosengarten, Sören Domrös, Marten Lohstroh, Soroush Bateni, and Christian Menard. “Pragmatics twelve years later: a report on Lingua Franca”. In: *11th International Symposium on Leveraging Applications of Formal Methods, Verification and Validation (ISoLA)*. Vol. 13702. Lecture Notes in Computer Science. Springer. Rhodes, Greece, Oct. 2022, pp. 60–89. DOI: 10.1007/978-3-031-19756-7\_5.

- [JK24] Byeonggil Jun and Hokeun Kim. *SimpleChat lingua franca model*. <https://github.com/lf-lang/playground-lingua-franca/blob/main/examples/C/src/ChatApplication/SimpleChat.lf>. Accessed: 2024-12-06. 2024.
- [JM04] Michael Jünger and Petra Mutzel, eds. *Graph drawing software*. Springer, 2004. ISBN: 978-3-642-62214-4. DOI: 10.1007/978-3-642-18638-7.
- [JMM+16] Adalat Jabrayilov, Sven Mallach, Petra Mutzel, Ulf Rüegg, and Reinhard von Hanxleden. “Compact layered drawings of general directed graphs”. In: *Proceedings of the 24th International Symposium on Graph Drawing and Network Visualization (GD ’16)*. 2016, pp. 209–221. DOI: 10.1007/978-3-319-50106-2.
- [Jöh22] Felix Jöhnk. “Structure-based editing for SCCharts”. Master thesis. Kiel University, Department of Computer Science, 2022. URL: <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/fej-mt.pdf>.
- [Kar72] Richard M. Karp. “Reducibility among combinatorial problems”. In: *Complexity of Computer Computations (Proceedings of a Symposium on the Complexity of Computer Computations, March, 1972, Yorktown Heights, NY)*. Ed. by Raymond E. Miller and James W. Thatcher. New York: Plenum Press, 1972, pp. 85–103. DOI: 10.1007/978-3-540-68279-0\_8.
- [KDH24] Maximilian Kasperowski, Sören Domrös, and Reinhard von Hanxleden. *The Eclipse Layout Kernel*. Poster at the 32nd International Symposium on Graph Drawing and Network Visualization, TU Wien, Vienna, Austria. Sept. 2024. DOI: 10.4230/LIPIcs.GD.2024.56.
- [KPS14] Stephen G. Kobourov, Sergey Pupyrev, and Bahador Sabet. “Are crossings important for drawing large graphs?” In: *Proceedings of the 22nd International Symposium on Graph Drawing (GD ’14)*. Vol. 8871. Springer, 2014, pp. 234–245. ISBN: 978-3-662-45802-0. DOI: 10.1007/978-3-662-45803-7\_20.

## Bibliography

- [KRD+24] Maximilian Kasperowski, Niklas Rentz, Sören Domrös, and Reinhard von Hanxleden. “KIELER: a text-first framework for automatic diagramming of complex systems”. In: *Proceedings of the 14th International Conference on the Theory and Application of Diagrams, DIAGRAMS '24*. Vol. 14981. LNCS. Springer Nature Switzerland, 2024, pp. 402–418. DOI: 10.1007/978-3-031-71291-3\_33.
- [KSS+12] Lars Kristian Klauske, Christoph Daniel Schulze, Miro Spönermann, and Reinhard von Hanxleden. “Improved layout for data flow diagrams with port constraints”. In: *Proceedings of the 7th International Conference on the Theory and Application of Diagrams (DIAGRAMS '12)*. Vol. 7352. LNAI. Springer, 2012, pp. 65–79. DOI: 10.1007/978-3-642-31223-6.
- [Kyn09] Jan Kynčl. “Enumeration of simple complete topological graphs”. In: *European Journal of Combinatorics* 30.7 (2009). EuroComb'07: Combinatorics, Graph Theory and Applications, pp. 1676–1685. ISSN: 0195-6698. DOI: 10.1016/j.ejc.2009.03.005.
- [Lee24a] Edward A. Lee. *AccelerometerDisplay lingua franca model*. <https://github.com/icyphy/lf-buckler/blob/main/src/AccelerometerDisplay.lf>. Accessed: 2025-02-14. 2024.
- [Lee24b] Edward A. Lee. *Cutter lingua franca model*. <https://github.com/lf-lang/playground-lingua-franca/blob/main/experiments/C/src/Safety/Cutter.lf>. Accessed: 2024-11-20. 2024.
- [LL19] Marten Lohstroh and Edward A. Lee. “Deterministic Actors”. In: *Proc. Forum on Specification and Design Languages (FDL '19)*. Southampton, UK, Sept. 2019. DOI: 10.1109/FDL.2019.8876922.
- [LMB+21] Marten Lohstroh, Christian Menard, Soroush Bateni, and Edward A. Lee. “Toward a Lingua Franca for deterministic concurrent systems”. In: *ACM Transactions on Embedded Computing Systems (TECS)* 20.4 (May 2021), Article 36. DOI: 10.1145/3448128.
- [LMS+20] Marten Lohstroh, Christian Menard, Alexander Schulz-Rosengarten, Matthew Weber, Jeronimo Castrillon, and Edward A. Lee. “A language for deterministic coordination across multiple timelines”. In: *Proc. Forum on Specification and Design*

- Languages (FDL '20)*. Kiel, Germany, Sept. 2020. DOI: 10.1109/FDL50818.2020.9232939.
- [Loh24] Marten Lohstroh. *TrainDoor lingua franca model*. <https://github.com/lf-lang/playground-lingua-franca/blob/main/examples/C/src/train-door/TrainDoor.lf>. Accessed: 2024-12-06. 2024.
- [Luc18] Daniel Lucas. “Order- and drawing area-aware packing of rectangles”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/dalu-bt.pdf>. Bachelor Thesis. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Sept. 2018.
- [MCL+24] Christian Menard, Hannes K. M. Chorlian, Edward A. Lee, and Thee Ho. *SleepingBarber lingua franca model*. <https://github.com/lf-lang/playground-lingua-franca/blob/main/examples/C/src/SleepingBarber.lf>. Accessed: 2024-11-19. 2024.
- [MEL+95] Kazuo Misue, Peter Eades, Wei Lai, and Kozo Sugiyama. “Layout adjustment and the mental map”. In: *Journal of Visual Languages & Computing* 6.2 (June 1995), pp. 183–210. DOI: 10.1006/jvlc.1995.1010.
- [MHE20] Amyra Meidiana, Seok-Hee Hong, and Peter Eades. “New quality metrics for dynamic graph drawing”. In: *Graph Drawing and Network Visualization*. Cham: Springer International Publishing, 2020, pp. 450–465. DOI: 10.1007/978-3-030-68766-3\_35.
- [Mot17] Christian Motika. *Sccharts—language and interactive incremental implementation*. Kiel Computer Science Series 2017/2. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2017. DOI: 10.21941/kcss/2017/02.
- [MSW19] Robin JP Mennens, Roeland Scheepens, and Michel A Westenberg. “A stable graph layout algorithm for processes”. In: *Computer Graphics Forum*. Vol. 38. 3. Wiley Online Library, 2019, pp. 725–737.

## Bibliography

- [NTB05] Nikola S. Nikolov, Alexandre Tarassov, and Jürgen Branke. “In search for efficient heuristics for minimum-width graph layering with consideration of dummy nodes”. In: *Journal of Experimental Algorithmics* 10 (2005). ISSN: 1084-6654. DOI: 10.1145/1064546.1180618.
- [PDS+23] Jette Petzold, Sören Domrös, Connor Schönberner, and Reinhard von Hanxleden. “An interactive graph layout constraint framework”. In: *Proceedings of the 18th International Joint Conference on Computer Vision, Imaging and Computer Graphics Theory and Applications (VISIGRAPP 2023) - Volume 3: IVAPP*. With accompanying poster. INSTICC. SciTePress, 2023, pp. 240–247. ISBN: 978-989-758-634-7. DOI: 10.5220/0011803000003417.
- [Pet19] Jette Petzold. “Intentional Layout in Sprotty Diagrams: Defining User Interaction”. Bachelor Thesis. Kiel University, Department of Computer Science, Sept. 2019. URL: <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/jet-bt.pdf>.
- [Pet95] Marian Petre. “Why looking isn’t always seeing: Readership skills and graphical programming”. In: *Communications of the ACM* 38.6 (June 1995), pp. 33–44. DOI: 10.1145/203241.203251.
- [PH23] Jette Petzold and Reinhard von Hanxleden. “Tool Support for System-Theoretic Process Analysis”. In: *Electronic Communications of the EASST* 82 (2023).
- [PHG06] Helen C. Purchase, Eve E. Hoggan, and Carsten Görg. “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)*. Vol. 4372. LNCS. Springer, 2006, pp. 184–195. ISBN: 978-3-540-70903-9. DOI: 10.1007/978-3-540-70904-6.
- [PKH23] Jette Petzold, Jana Kreiß, and Reinhard von Hanxleden. “PASTA: Pragmatic Automated System-Theoretic Process Analysis”. In: *53rd Annual IEEE/IFIP International Conference on Dependable Systems and Network, DSN 2023, Porto, Portugal, June 27-30, 2023*. IEEE, 2023, pp. 559–567. DOI: 10.1109/DSN58367.2023.00058.

- [PPP12] Helen C. Purchase, Christopher Pilcher, and Beryl Plimmer. “Graph drawing aesthetics—created by users, not algorithms”. In: *IEEE Transactions on Visualization and Computer Graphics* 18.1 (2012), pp. 81–92. ISSN: 1077-2626. DOI: 10.1109/TVCG.2010.269.
- [Pro08] Steffen Prochnow. “Efficient Development of Complex Stat-echarts”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/spr-diss.pdf>. PhD thesis. Kiel, Germany: Christian-Albrechts-Universität zu Kiel, Faculty of Engineering, July 2008.
- [Pto14] Claudius Ptolemaeus, ed. *System design, modeling, and simulation using Ptolemy II*. Ptolemy.org, 2014. URL: <http://ptolemy.org/books/Systems>.
- [Pur02] Helen C. Purchase. “Metrics for graph drawing aesthetics”. In: *Journal of Visual Languages and Computing* 13.5 (2002), pp. 501–516.
- [Pur12] Helen C. Purchase. *Experimental human-computer interaction - A practical guide with visual examples*. Cambridge University Press, 2012. ISBN: 978-0-521-27954-3.
- [Pur14] Helen C. Purchase. “A healthy critical attitude: revisiting the results of a graph drawing study”. In: *Journal of Graph Algorithms and Applications* 18.2 (2014), pp. 281–311. DOI: 10.7155/jgaa.00323.
- [Pur97] Helen C. Purchase. “Which aesthetic has the greatest effect on human understanding?” In: *Proceedings of the 5th International Symposium on Graph Drawing (GD ’97)*. Vol. 1353. LNCS. Springer, 1997, pp. 248–261. DOI: 10.1007/3-540-63938-1\_67.
- [RES+16] Ulf Rüegg, Thorsten Ehlers, Miro Spönemann, and Reinhard von Hanxleden. “A generalization of the directed graph layering problem”. In: *Proceedings of the 24th International Symposium on Graph Drawing and Network Visualization (GD ’16)*. 2016, pp. 196–208. DOI: 10.1007/978-3-319-50106-2\_16.

## Bibliography

- [RH18] Ulf Rüegg and Reinhard von Hanxleden. “Wrapping layered graphs”. In: *Proceedings of the 10th International Conference on the Theory and Application of Diagrams (DIAGRAMS '18)*. Springer, 2018, pp. 743–747. DOI: 10.1007/978-3-319-91376-6\_72.
- [Rie10] Martin Rieß. “A graph editor for algorithm engineering”. Bachelor Thesis. Kiel University, Department of Computer Science, Sept. 2010. URL: <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mri-bt.pdf>.
- [Rie22] Max Riepe. “Model order and cycle breaking in SCCharts”. Bachelor Thesis. Kiel University, Department of Computer Science, Mar. 2022. URL: <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mwr-bt.pdf>.
- [Rie24] Max Riepe. “Group model order for sugiyama layouts”. Master Thesis. Kiel University, Department of Computer Science, Mar. 2024. URL: <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/mwr-mt.pdf>.
- [RLP+16] Ulf Rüegg, Rajneesh Lakkundi, Ashwin Prasad, Anand Kodaganur, Christoph Daniel Schulze, and Reinhard von Hanxleden. “Incremental diagram layout for automated model migration”. In: *Proceedings of the ACM/IEEE 19th International Conference on Model Driven Engineering Languages and Systems (MoDELS '16)*. 2016, pp. 185–195. DOI: 10.1145/2976767.2976805.
- [RMJ+24] Julian Robledo, Christian Menard, Erling Jellum, Edward A Lee, and Jeronimo Castrillon. “Timing enclaves for performance in lingua franca”. In: *2024 Forum on Specification & Design Languages (FDL)*. IEEE, 2024, pp. 1–9. DOI: 10.1109/FDL63219.2024.10673834.
- [RSC+15] Ulf Rüegg, Christoph Daniel Schulze, John Julian Carstens, and Reinhard von Hanxleden. “Size- and port-aware horizontal node coordinate assignment”. In: *Proceedings of the 23rd International Symposium on Graph Drawing and Network Visualization (GD '15)*. 2015, pp. 139–150. DOI: 10.1007/978-3-319-27261-0\_12.



- [RSG+16] Ulf Rüegg, Christoph Daniel Schulze, Daniel Grevismühl, and Reinhard von Hanxleden. “Using one-dimensional compaction for smaller graph drawings”. In: *Proceedings of the 9th International Conference on the Theory and Application of Diagrams (DIAGRAMS '16)*. Vol. 9781. LNCS. Springer, 2016, pp. 212–218. DOI: 10.1007/978-3-319-42333-3\_16.
- [RSS+16] Ulf Rüegg, Christoph Daniel Schulze, Carsten Sprung, Nis Wechselberg, and Reinhard von Hanxleden. *Edge bundling for dataflow diagrams*. Poster at the 24th International Symposium on Graph Drawing and Network Visualization (GD '16). 2016.
- [Rüe18] Ulf Rüegg. *Sugiyama layouts for prescribed drawing areas*. Kiel Computer Science Series 2018/1. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2018. DOI: 10.21941/kcss/2018/1.
- [Sch16] Christoph Daniel Schulze. “Two opportunities and challenges of automatic layout in visual languages”. In: *Proceedings of the ACM Student Research Competition at MODELS 2016 co-located with the 19th International Conference on Model Driven Engineering Languages and Systems (MODELS 2016)*. 2016.
- [Sch19a] Connor Schönberner. “Intentional Layout in Sprotty Diagrams: Reevaluating Introduced Constraints”. <https://rtsys.informatik.uni-kiel.de/~biblio/downloads/theses/cos-bt.pdf>. Bachelor thesis. Kiel University, Department of Computer Science, Sept. 2019.
- [Sch19b] Christoph Daniel Schulze. *Text in diagrams: challenges to and opportunities of automatic layout*. Kiel Computer Science Series 2019/4. Dissertation, Faculty of Engineering, Kiel University. Department of Computer Science, CAU Kiel, 2019. DOI: 10.21941/kcss/2019/4.
- [Sch24] Alexander Schulz-Rosengarten. *Language design for reactive systems — on modal models, time, and object orientation in lingua franca and sccharts*. Kiel Computer Science Series 2024/1. Dissertation, Faculty of Engineering, Kiel University. Department of Computer Science, 2024. DOI: 10.21941/kcss/2024/1.

## Bibliography

- [SDH14a] Miro Spönemann, Björn Duderstadt, and Reinhard von Hanxleden. *Evolutionary meta layout of graphs*. Technical Report 1401. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Jan. 2014.
- [SDH14b] Miro Spönemann, Björn Duderstadt, and Reinhard von Hanxleden. “Evolutionary meta layout of graphs”. In: *Proceedings of the 8th International Conference on the Theory and Application of Diagrams (DIAGRAMS '14)*. Vol. 8578. LNAI. Springer, 2014, pp. 16–30. DOI: 10.1007/978-3-662-44043-8\_3.
- [SFH+10] Miro Spönemann, Hauke Fuhrmann, Reinhard von Hanxleden, and Petra Mutzel. “Port constraints in hierarchical layout of data flow diagrams”. In: *Proceedings of the 17th International Symposium on Graph Drawing (GD '09)*. Vol. 5849. LNCS. Springer, 2010, pp. 135–146. DOI: 10.1007/978-3-642-11805-0.
- [SFH09] Miro Spönemann, Hauke Fuhrmann, and Reinhard von Hanxleden. *Automatic layout of data flow diagrams in KIELER and Ptolemy II*. Technical Report 0914. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, July 2009.
- [SH24] Alexander Schulz-Rosengarten and Reinhard von Hanxleden. *Chrono lingua franca model*. <https://github.com/lf-lang/playground-lingua-franca/blob/main/experiments/C/src/ModalModels/Motivation/Chrono/Chrono.lf>. Accessed: 2024-11-21. 2024.
- [SHH18] Christoph Daniel Schulze, Gregor Hoops, and Reinhard von Hanxleden. “Automatic layout and label management for UML sequence diagrams”. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '18)*. 2018. DOI: 10.1109/VLHCC.2018.8506571.
- [SHL+23] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Marten Lohstroh, Edward A. Lee, and Soroush Bateni. “Polyglot modal models through Lingua Franca”. In: *Proceedings of Cyber-Physical Systems and Internet of Things Week 2023. CPS-IoT Week*

- '23. San Antonio, TX, USA: ACM, 2023, pp. 337–342. DOI: 10.1145/3576914.3587498.
- [SHM+18] Alexander Schulz-Rosengarten, Reinhard von Hanxleden, Frédéric Mallet, Robert de Simone, and Julien Deantoni. “Time in SCCharts”. In: *Proc. Forum on Specification and Design Languages (FDL '18)*. Munich, Germany, Sept. 2018. DOI: 10.1109/FDL.2018.8524111.
- [SLH16] Christoph Daniel Schulze, Yella Lasch, and Reinhard von Hanxleden. “Label management: keeping complex diagrams usable”. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '16)*. Sept. 2016, pp. 3–11. DOI: 10.1109/VLHCC.2016.7739657.
- [SMS+15] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Nis Boerge Wechselberg, Carsten Sprung, and Reinhard von Hanxleden. *SCCharts: the railway project report*. Technical Report 1510. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Aug. 2015.
- [SMS+19] Steven Smyth, Christian Motika, Alexander Schulz-Rosengarten, Sören Domrös, Lena Grimm, Andreas Stange, and Reinhard von Hanxleden. *SCCharts: the mindstorms report*. Technical Report 1904. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, Dec. 2019.
- [Spö15] Miro Spönemann. *Graph layout support for model-driven engineering*. Kiel Computer Science Series 2015/2. Dissertation, Faculty of Engineering, Christian-Albrechts-Universität zu Kiel. Department of Computer Science, 2015. ISBN: 9783734772689.
- [SSH14] Christoph Daniel Schulze, Miro Spönemann, and Reinhard von Hanxleden. “Drawing layered graphs with port constraints”. In: *Journal of Visual Languages and Computing, Special Issue on Diagram Aesthetics and Layout* 25.2 (2014), pp. 89–106. ISSN: 1045-926X. DOI: 10.1016/j.jvlc.2013.11.005.

## Bibliography

- [SSM19] Alexander Schulz-Rosengarten, Steven Smyth, and Michael Mendler. “Towards object-oriented modeling in SCCharts”. In: *Proc. Forum on Specification and Design Languages (FDL '19)*. Southampton, UK, Sept. 2019. DOI: 10.1145/3453482.
- [SSR+14] Miro Spönemann, Christoph Daniel Schulze, Ulf Rüegg, and Reinhard von Hanxleden. “Counting crossings for layered hypergraphs”. In: *Proceedings of the 8th International Conference on the Theory and Application of Diagrams (DIAGRAMS '14)*. Vol. 8578. LNAI. Springer, 2014, pp. 9–15. DOI: 10.1007/978-3-662-44043-8\_2.
- [SSS+14] Christoph Daniel Schulze, Miro Spönemann, Christian Schneider, and Reinhard von Hanxleden. “Two applications for transient views in software development environments (show-piece)”. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC '14)*. Melbourne, Australia, July 2014.
- [STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiro Toda. “Methods for visual understanding of hierarchical system structures”. In: *IEEE Transactions on Systems, Man and Cybernetics* 11.2 (Feb. 1981), pp. 109–125. DOI: 10.1109/TSMC.1981.4308636.
- [Sug02] Kozo Sugiyama. *Graph drawing and applications for software and knowledge engineers*. Vol. 11. Software Engineering and Knowledge Engineering. World Scientific, 2002. ISBN: 978-981-02-4879-6. DOI: 10.1142/9789812777898\_0008.
- [SWH18] Christoph Daniel Schulze, Nis Wechselberg, and Reinhard von Hanxleden. “Edge label placement in layered graph drawing”. In: *Proceedings of the 10th International Conference on the Theory and Application of Diagrams (DIAGRAMS '18)*. LNCS. Springer, 2018, pp. 71–78. ISBN: 978-3-319-91376-6. DOI: 10.1007/978-3-319-91376-6\_10.
- [Tam13] Roberto Tamassia, ed. *Handbook of graph drawing and visualization*. CRC Press, 2013. ISBN: 978-1584884125.

- [Tar72] Robert E. Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM Journal of Computing* 1.2 (1972), pp. 146–160. DOI: 10.1137/0201010.
- [Tay96] Conrad Taylor. “What has WYSIWYG done to us?” In: *The Seybold Report on Publishing Systems* 26.2 (Sept. 1996).
- [Ume24] Khubaib Umer. *load\_balancer lingua franca model*. [https://github.com/MagnitionIO/LF\\_Collaboration/blob/main/complex-view-model/src/loadbalancer\\_linked\\_latest.lf](https://github.com/MagnitionIO/LF_Collaboration/blob/main/complex-view-model/src/loadbalancer_linked_latest.lf). Accessed: 2024-12-06. 2024.
- [Wad01] Vance Waddle. “Graph layout for displaying data structures”. In: *Proceedings of the 8th International Symposium on Graph Drawing (GD ’00)*. Vol. 1984. LNCS. Springer, 2001, pp. 98–103. DOI: 10.1007/3-540-44541-2\_23.
- [WCB+17] Yunhai Wang, Xiaowei Chu, Chen Bao, Lifeng Zhu, Oliver Deussen, Baoquan Chen, and Michael Sedlmair. “Edwordle: consistency-preserving word cloud editing”. In: *IEEE transactions on visualization and computer graphics* 24.1 (2017), pp. 647–656. DOI: 10.1109/TVCG.2017.2745859.
- [ZTH+22] Jan Zielasko, Sören Tempel, Vladimir Herdt, and Rolf Drechsler. “3d visualization of symbolic execution traces”. In: *2022 Forum on Specification and Design Languages, FDL 2022, Linz, Austria*. IEEE. 2022, pp. 1–8. DOI: 10.1109/FDL56239.2022.9925664.

