Group Model Order for Sugiyama Layouts

Max Philipp Wilhelm Riepe

Masters's Thesis
March 2024

Prof. Dr. Reinhard von Hanxleden
Real-Time and Embedded Systems
Department of Computer Science
Kiel University

Advised by
Sören Domrös

Selbstständigkeitserklärung

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

Kiel,

Abstract

Automatic visualization of graphs for modeling languages is very beneficial for the development time, as it eliminates the task of manually creating clear and readable layouts which can be tedious. Especially models in development that are evolving and changing requiring constant new layout creation.

The Sugiyama algorithm is a commonly used layout algorithm for synthesized graphs from a textual modeling language. Recent research on the layered algorithm shows that the order of declaration in the textual model has an *intention* that may be used to improve the layout creation. However, for modeling languages with multiple node types which are grouped by their type in the textual model, the use of the raw textual order has problems. Connections between different node types cannot be evaluated based on the textual order, as the strict order induced by the textual grouping creates artifacts that go against common graph drawing practices, e.g., reducing the number of edge crossings.

This thesis aims to find a way of incorporating the initial order for languages with restrictions on the order of component declarations, by grouping node types. For this multiple approaches for the first and third phase of the layered approach have been implemented and evaluated. This thesis proposes *Group Model Order* sensitive strategies, an approach to evaluate nodes from different groups of the textual order. Furthermore, one aspect of working with hierarchical graphs is the ability to work with certain sections or subgraphs by expanding and collapsing nodes. An approach is suggested that improves the stability of the layout for these actions.

Acknowledgements

First, I want to thank Prof. Dr. Reinhard von Hanxleden for the opportunity to write this thesis and his feedback. I want to express gratitude to Sören Domrös, who advised this thesis, and for the generous and swift feedback he provided, not only for this thesis but also for my Bachelor's thesis. The Real-Time and Embedded Systems group provided feedback to advance this thesis in an atmosphere I will miss after completing it. Finally, I want to thank my family and friends for supporting me.

Contents

List of Figures

List of Tables

Abbreviations

GUI	Graphical User Interface
FSM	Finite State Machine
D&D	Drag And Drop
WYSIWYG	What You See Is What You Get
ELK	Eclipse Layout Kernel
ONO	Obviously Non-Optimal
NONO	Nothing is Obviously Non-Optimal
OYES	Obvious Yet Easily Superior
SCChart	Sequentially Constructive Statechart
LF	Lingua Franca
KIELER	Kiel Integrated Environment for Layout Eclipse Richt Client
SCC	Strongly Connected Components
IDE	Integrated Development Environment
VS Code	Visual Studio Code
WCET	Worst-Case Execution Time
G-CB	Greedy Cycle Breaker
BF-CB	Breadth-First Cycle Breaker
BFS	Breadth-First Search
DF-CB	Depth-First Cycle Breaker
DFS	Depth-First Search
MO-CB	Model Order Cycle Breaker
ST-CB	Strict Type Cycle Breaker
MOLA-CB	Model Order Look Ahead Cycle Breaker
SCC-CB	Strongly Connected Component Cycle Breaker
CSV	Comma-Separated Value
SCC_CON-CB	Strongly Connected Components Connectivity Cycle Breaker
SCC_NODE-CB	Strongly Connected Components Node Type Cycle Breaker
CPU	Central Processing Unit
RAM	Random-Access Memory

Chapter 1

Introduction

Two formats exist for presenting information on paper, textually or with diagrams. While textual specifications have the benefit of precision, this precision with words can be very long, especially when describing graphical information like a GUI or complex system designs. Showing dependencies or connections is much easier in a graphical way. Graphical abstraction of information can reduce the required time to understand the information. Comparing large datasets of raw data in a table format is disadvantageous compared to data in a diagram.

One fundamental concept in computer science are *Finite State Machines* (FSMs). A FSM consists of nodes (or states) and edges (or transitions) that connect different nodes. These can be used to simulate behavior based on the current state, with conditional transitions to change the behavior of the system. A large field in computer science that utilizes these state machines heavily is game development, where FSMs can be used to modify the animation of a character based on the current scenario (i.e., falling, running, idle) or to change the behavior of a character in general (i.e., chasing, running away, fighting). One of the most significant problems when working with large state machines is the creation of a readable layout. Figure 1.1[1] shows a state machine for animating a character in the game development engine Unity[2]. The model uses the *Drag And Drop* (D&D) editor for animation states contained in Unity, called mecanim[3].

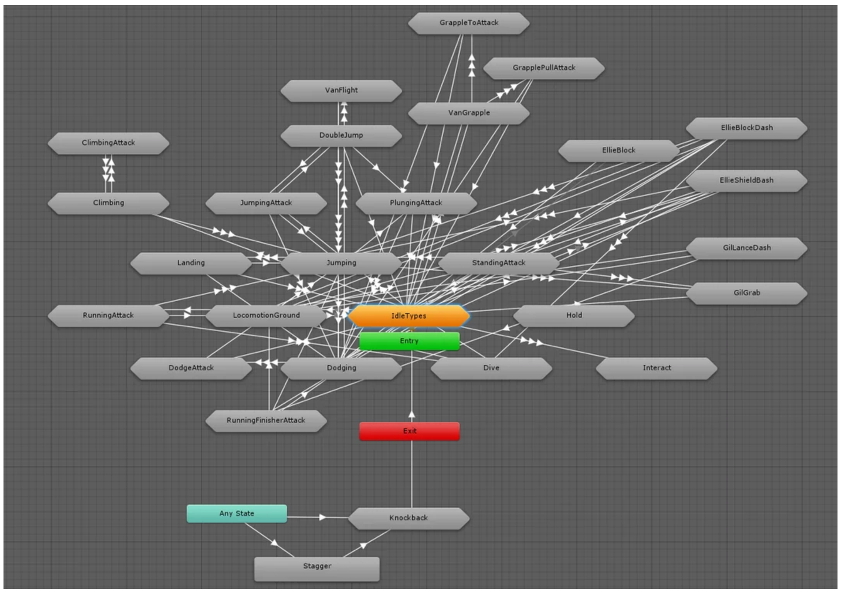


Figure 1.1. Large scale Mecanim model for animating the states of a character.

[1] https://www.reddit.com/r/Unity3D/comments/7bvy81/using_mecanim_as_a_general_purpose_state_machine/

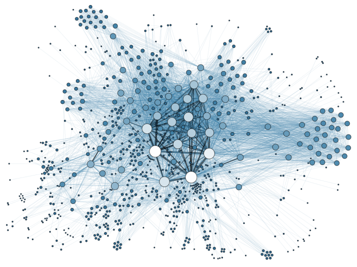
[2] https://unity.com/

[3] https://docs.unity3d.com/462/Documentation/Manual/MecanimAnimationSystem.html

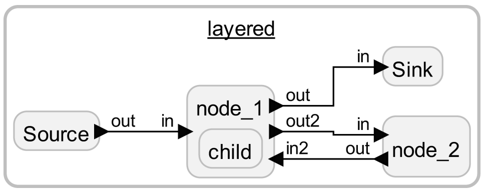
This editor lacks an automatic layout functionality. As a result, many similar models may be found online, with frequent questions in the Unity community on reddit[4] how to properly create layouts in this editor.

Creating a clear layout is tedious and time consuming [Pet95]. The task of creating a good layout is especially frustrating for FSMs that are still in development and, as a result, require changes and possibly a significant restructuring of the layout. The solution for this problem are automatic layout algorithms. Similar to the editor in Unity, the game engine Unreal Engine[5] has a low code option for developing games by connecting behavior nodes called Blueprints[6]. This editor had the same problem of missing an automatic layout functionality. However, a solution for this is available as a plugin[7].

Figure 1.2 shows the two different layout algorithms. The force-based layout algorithm [FR91] and the layered algorithm [STT81]. Figure 1.2b shows a layered graph with three layers L ={{`Source`},{`node_1`},{`Sink`,`node_2`}}, contained within a root node with the label `layered`.



(a) A network layout using the force-based layout approach [Gra14].



(b) A graph with the layered layout algorithm.

Figure 1.2. The force and layered layout algorithms.

While the force-based approach is perfect for showing connected groups, working with a FSM that uses this approach is insensible, as this approach does not focus on reducing edge crossings. The force-based approach generally does not differentiate between directed and undirected graphs, not utilizing additional information and intention in the graph. The version of the layered approach for this thesis follows a general layout direction, which can show the flow of a graph. Edges that go against this layout direction are called *backward edges*. Additionally, a fundamental principle for the layered approach is the reduction of visual artifacts like edge crossings [STT81]. These principles for the layered approach allow easier tracking of edges.

1.1 Input Variants

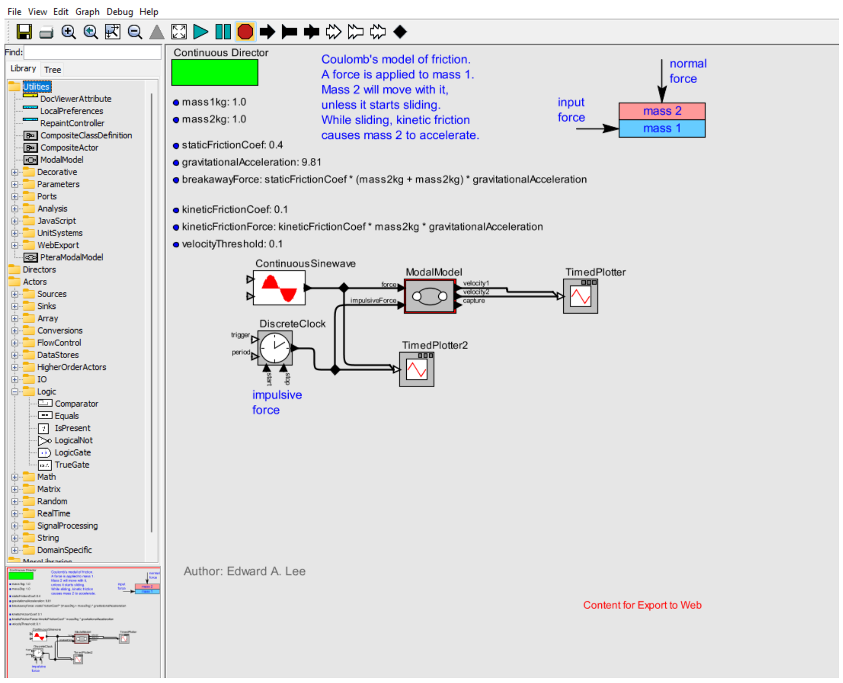
Generally, there are two approaches to the creation of diagrams:

▻ **Drag And Drop Editors**: This form of graph creation typically presents a palette of graphical elements the user can choose from. The chosen element is placed in the drawing area, where the user can freely move this element to the desired location. Connections between elements are

[4]`www.reddit.com/r/Unity3D/search/?q=Mecanim`
[5]`https://www.unrealengine.com/de`
[6]`https://docs.unrealengine.com/4.27/en-US/ProgrammingAndScripting/Blueprints/GettingStarted/`
[7]`https://github.com/howaajin/graphformatter`

manually created. Figure 1.3a shows such a D&D Editor for the *modeling language* Ptolemy II[8]. These editors are often called *What You See Is What You Get* (WYSIWYG) editors.

▻ **Graph synthesis**: In this approach, a graph is synthesized from a textual definition. For this input variant, a notation is specified that declares the rules and terminology to define different types of nodes and the connections between nodes. Listing 1.1 shows a simple textual definition in the *Eclipse Layout Kernel* (ELK)[9] [DHS+23] Text format[10] (file format: `.elkt`). Figure 1.3b shows the synthesized graph for this definition.



Coulomb's model of friction.
A force is applied to mass 1.
Mass 2 will move with it,
unless it starts sliding.
While sliding, kinetic friction
causes mass 2 to accelerate.

- mass1kg: 1.0
- mass2kg: 1.0
- staticFrictionCoef: 0.4
- gravitationalAcceleration: 9.81
- breakawayForce: staticFrictionCoef * (mass2kg + mass2kg) * gravitationalAcceleration
- kineticFrictionCoef: 0.1
- kineticFrictionForce: kineticFrictionCoef * mass2kg * gravitationalAcceleration
- velocityThreshold: 0.1

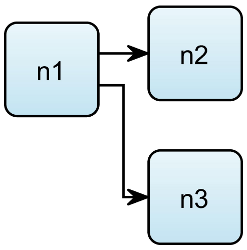
Author: Edward A. Lee

Content for Export to Web

(a) Ptolemy II GUI and sample model. The element picker is shown in the left of the image, with the drawing to the right of it.

```
algorithm: layered

node n1
node n2
node n3

edge n1 -> n2
edge n1 -> n3
```

Listing (1.1) Graph definition in the elkt format.



(b) The synthesized diagram for Listing 1.1

Figure 1.3. D&D editor for graphs vs synthesizing a textual mode.

A graph created from a textual model requires an automatic layout algorithm if the editor does not

[8]`https://ptolemy.berkeley.edu/index.htm`
[9]`https://eclipse.dev/elk/`
[10]`https://eclipse.dev/elk/documentation/tooldevelopers/graphdatastructure/elktextformat.html`

supply a drag-and-drop functionality. However, both versions of input graphs can utilize automatic layout algorithms.

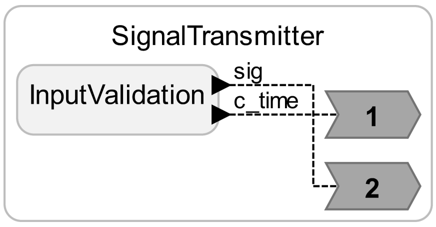
The term *pragmatics* [FH10] is used to describe the benefits of separating the textual and graphical views. The textual version can be used for code synthesis and simulation, while the diagram view helps with the development and documentation. Pragmatics has been a key principle in the development of *Lingua Franca* (LF) [HLF+22]. LF is the modeling language used for this thesis, further introduced in Section 2.6.

1.2 Related Work

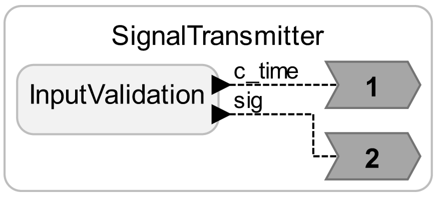
This thesis aims to improve the layout creation using the layered approach [STT81] for models with textual node type grouping, as explained in Section 1.4.

The term *secondary notation*, as defined by Marian Petre [Pet95], describes additional information in models that are not part of the syntax and how it can be used to create better layouts. This additional information can be used to increase the connection between the textual definition of a graph and the structure of the final layout. An example of achieving this connection is utilizing the declaration order of the textual file, as described in Section 1.3.1.

Additionally, this thesis aims to reduce layout decisions that lead to *Obviously Non-Optimal* (ONO) solutions. ONO layouts break the *Nothing is Obviously Non-Optimal* (NONO) principle for good layouts [KDM+16]. A layout can be considered ONO for a multitude of reasons. I.e., Figure 1.4a shows a layout which is ONO concerning edge crossings, however it retains the declaration order of the ports, making the layout optimal for the use of *model order*, as defined in Section 1.3.1. A slight difference in the layout may create an *Obvious Yet Easily Superior* (OYES) layout concerning edge crossings, as shown in Figure 1.4b. However, this variant ignores the inherent port order making it ONO concerning the utilization of model order.



(a) Model layout with edge crossing, retaining inherent port order.



(b) Model layout without edge crossing, ignoring inherent port order.

Figure 1.4. Comparing different layouts of the same graph with a minor adjustment.

1.3 Control

Control is defined as the impact the developer of a model can have on the final layout. For D&D editors, the developer has complete control, freely manipulating the layout in the drawing area. This control may come with the drawback of requiring more time from the developer, especially for very fine grain decisions. It may seem that this relation between control and workload is generally the case, as higher control means fewer autonomous decisions are possible. However, the inability to modify the layout easily may be frustrating if a layout is not clear. Additionally, autonomous decisions may be faulty and result in bad layouts. While some layout algorithms remove the control, which might be okay for some graphs, this is undesirable for layout algorithms used for models where the visualization is not only used as an abstraction but is used for development and debugging. Without any control, the developer is at the mercy of the layout algorithm to create a result that is clear and readable. The recent explorations of integrating model order into the layout process introduced a method for retaining control with the benefits of autonomous decisions. [Rie22]

1.3.1 Model Order

The *model order* is defined as the inherent order of the model. This can be the order of the original placement of the elements in the graph, or the order of declaration in the textual file creates the model order. This inherent ordering is a form of secondary notation that is often neglected, as layout algorithms usually do not consider the initial order. The intentional ordering of nodes in the textual file represents a *mental map* of the nodes, an abstract layout the model creator has. Using model order may help with finding nodes quicker in the diagram, as the textual structure is represented in the layout. Additionally, the use of model order increases the control of the model creator. Model order is a one-dimensional order that is mapped onto a two dimensional drawing plane. However, the decisions on this plane are also one-dimensional, i.e., the relative placement of hypothetical nodes `n1` and `n2` can be interpreted as two one-dimensional model order comparisons to determine the relative x or y position.

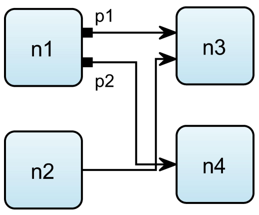
Different elements have a different model order, i.e., the *port order* is independent from the model order for nodes or the edge order. The port order of different nodes is incomparable, as the port order is only relevant for ports of the same node. The same applies to edges. Furthermore, some modeling languages may not declare an order for a specific element. E.g., in Sequentially Constructive Statecharts (SCCharts) ports are not declared explicitly but are created implicitly and inherit the order from the explicitly defined edge. The same applies to all node type in LF, except for reactors. Therefore, one challenge for model order is to create sensible solutions to either artificially extract a model order or to utilize different strategies if an element with model order is compared to another element without fitting model order.

Figure 1.5 shows an example of utilizing model order compared to random decisions. Listing 1.2 shows the textual file for a simple model with two nodes. Figure 1.5a shows the synthesized graph for the textual file in Listing 1.2, and utilizes the model order to order the nodes. Figure 1.5b shows the same graph with a layout that does not utilize model order and employs decisions to find the best layout regarding edge crossings. However, a problem arises if the textual file has conflicting declaration orders. Figure 1.6 shows an example of *crossing declaration order*, meaning the declaration order of different graphical elements and the elements they connect to are not identical or even inverse. The node and edge definitions, shown in Listing 1.3, show this: `n1`, `n2` and the edges in reverse order `(init,n2)`, `(init,n1)`. Figure 1.6a shows the layout if the node order is prioritized and Figure 1.6b shows the layout with prioritized edge order. Enforcing both orders would create an edge crossing.

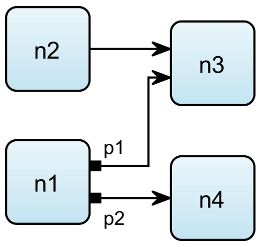
The first approaches to utilizing this order for the layered algorithms were presented by Domrös et al. [DH21]. These proposals have been pushed to more parts of the layered algorithm, and recent research has shown that using the textual ordering to aid the layout creation can lead to better layouts [DRv23; Rie22].

```
algorithm: layered

node n1{
  port p1
  port p2
}
node n2
node n3
node n4

edge n1.p1 -> n3
edge n1.p2 -> n4
edge n2 -> n3
```

Listing (1.2) Simple model in elkt format.



(a) Using model order.

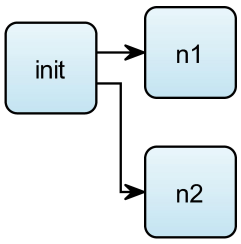


(b) Random decision to reduce crossings.

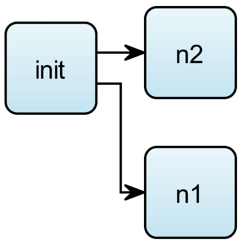
Figure 1.5. Example for secondary notation.

```
algorithm: layered

node init {...}
node n1 {...}
node n2 {...}

edge init -> n2
edge init -> n1
```

Listing (1.3) Crossing declaration order.



(a) Prioritize node order.



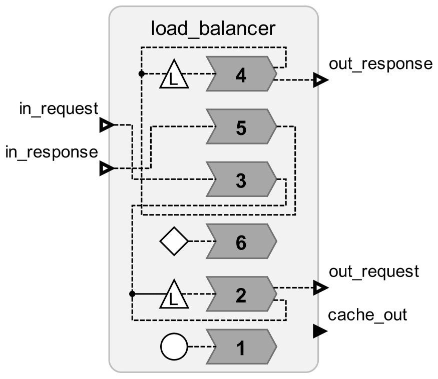
(b) Random decision to reduce crossings.

Figure 1.6. Prioritizing edge order.

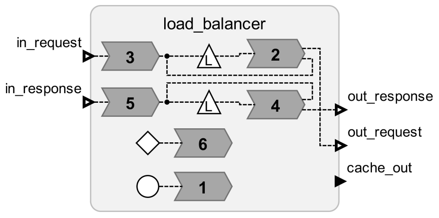
1.4 Group Model Order

Previous works on model order have mainly worked with *Sequentially Constructive Statecharts* SCCharts [HDM+14]. In SCCharts, the declaration order of nodes is freely adjustable. While edge declaration is also freely adjustable, the order of edges indicates the order of execution. Edges in SCCharts have conditions, and if the condition is fulfilled, the transition is executed, changing the state and executing any possible action the edge defined. As SCCharts are designed to be deterministic, only one transition condition should be fulfillable. Therefore, the edge order is used as a priority, taking the transition with the highest priority that is fulfilled.

LF has multiple *node types*, one of which has a constraint similar to that of edges in SCCharts. The different node types in LF are primarily defined in an order where the different types are grouped, with edges freely connecting between nodes of any type. This *group order* impairs the model order and creates a *restricted model order*. Figure 1.7 shows the limitations of utilizing the model order cycle-breaking approach used for SCCharts. In Figure 1.7a the first phase of the layered approach uses a strict model order approach, but since reactions are defined after the other node types, edges from reactions to other types are reversed. Figure 1.7b shows how the distinction of node groups can help with the layout creation, by reducing edge crossings. Listing 1.4 shows the textual model, where a blank line separates the different groups of node-types. This grouping of node types restricts the use of model order.



(a) Strict model order cycle-breaking.



(b) Cycle-breaking considering group order.

```
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;
   output cache_out: p_cdn_cache_entry_t;

   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)
}
```

Listing (1.4) LF declaration file.

Figure 1.7. Limitations of strictly using model order.

1.5 Outline

The following chapter introduces basic graph terminology and introduces the technologies that are utilized. Following that, Chapter 3 and Chapter 4 explore different phases of the layered algorithm and how group model order may be employed to improve the layout creation. These chapters show implementations for different strategies of resolving the restricted model order and keeping the benefits of utilizing model order. Following this, the different strategies are evaluated in Chapter 5. The analysis uses quality metrics for this, which are defined in 2.3. Finally, a conclusion to the results of this thesis is given in Chapter 6.

Chapter 2

Preliminaries

In this chapter, basic terminology, methodologies, principles, and frameworks for this thesis are introduced and explained.

We begin with some fundamental terminology of graph theory and additional terms for graph drawing. This is followed by layout quality metrics and the introduction of the relevant frameworks, such as the ELK and the Kiel Integrated Environment for Layout Eclipse Richt Client (KIELER)[1] environment [HFS11]. This chapter concludes with a brief introduction into the modeling language Lingua Franca[2], an overview of the different layout phases of the layered algorithm, an introduction of an algorithm to find *Strongly Connected Components* (SCC) and the definition of *stability*.

2.1 Terminology

Most of the following terms and their definitions are standard in graph theory. Some additional terms will be defined to enhance the flow and facilitate a better understanding of the following chapters.

Beginning with the most general terms:

2.1 Definition (Graph). A *graph* G is a pair (V, E), where V is an unordered finite set of nodes and an unordered set of edges.

2.2 Definition (Directed edges). Let $u, v \in V$ and $e \in E$ with $e = (u, v)$. For *directed* edges the pair (u, v) is ordered and indicates the direction of the edge, starting in u and ending in v. Conversely, an *undirected* edge does not have a direction. The pair has no ordering and $(u, v) = (v, u)$.

2.3 Definition (Out-degree). The *out-degree* of a node v is the number of edges that start in v, written v_{out}.

2.4 Definition (In-degree). The *in-degree* of a node v is the number of edges that end in v, written v_{in}.

2.5 Definition (Source). A node is called *source*, if its in-degree is 0. An example is the node labeled `Source` in Figure 1.2b.

2.6 Definition (Sink). A node is called *sink*, if its out-degree is 0. An example is the node labeled `Sink` in Figure 1.2b.

2.7 Definition (Outflow). For a node n the *outflow* $v_{outflow}$ is defined as $v_{outflow} = v_{out} - v_{in}$.

2.8 Definition (Directed graph). A graph $G = (V, E)$ is a called *directed* graph, if all edges $e \in E$ are directed edges. A directed graph is also called *digraph*.

2.9 Definition (Path). Let $G = (V, E)$ be a graph. A *path* of length n is defined as a tuple $(v_1, ..., v_n)$ with $v_1, ..., v_n \in V$ and it holds that $\forall i, 0 < i < n : (v_i, v_{i+1}) \in E$.

Let $\mathbb{P}$ be the set of all paths of G, therefore $\forall u, w \in V$, if a path from u to w exists, $(u, w) \in \mathbb{P}$

[1] `https://rtsys.informatik.uni-kiel.de/en/archive/kieler`

[2] `https://www.lf-lang.org`

2.10 Definition (Cycle)**.** A *cycle* is a path for which $v_1 = v_n$ holds.

2.11 Definition (Strongly connected component)**.** A Strongly Connected Components (SCC) is a set of nodes S for which the following holds. $v \in V, S \subseteq V : v \in S \Leftrightarrow \forall u \in S : (v, u) \in \mathbb{P} \wedge (u, v) \in \mathbb{P}$

2.12 Definition (Detached Parts)**.** A *detached part* of a graph is a set of nodes D that is not connected to the currently *inspected section* of the graph. In turn the currently inspected section is a detached part of the detached part.

2.13 Definition (Acyclic graph)**.** A directed graph $G = (V, E)$ is called *acyclic* if the graph does not contain any cycles.

2.14 Definition (Layered graph)**.** A *layered* graph is defined by the triple $G = (V, E, L)$, where L is defined as a finite ordered set, of sets containing nodes, which partitions V into non-empty *layers*. For all $v \in V$ holds that v is assigned to exactly one layer $L(v)$.

2.15 Definition (Proper Layered Graph)**.** Let $G = (V, E, L)$ be a acyclic layered graph. G is called *proper layered graph*, if the following holds: $\forall v, w \in V : \exists (v, w) \in E \Rightarrow L(w) = L(v) + 1$.

2.16 Definition (Hierarchical Node)**.** A node is called *hierarchical*, if it contains child nodes. In the graphical representation, this is displayed by drawing *child nodes* inside of the hierarchical node. The containing hierarchical node is called *parent node* for all contained nodes. An example is the node labeled `node_1` in Figure 1.2b.

2.17 Definition (Feedback set)**.** Let F be a subset of nodes of a given cyclic graph $G = (V, E)$. F is called *feedback set* if reversing all edges contained in F results in G being acyclic.

2.18 Definition (Graph Layout)**.** For a given graph $G = (V, E)$ a *layout* (or *drawing*) Γ is defined as a function that maps each node $v \in V$ to a 2-dimensional point $\Gamma(v) \in \mathbb{R}^2$ in the drawing plane. Edges $(u, v) \in E$ are mapped to simple curves $\Gamma(u, v)$ with the endpoints $\Gamma(u)$ and $\Gamma(v)$.

2.19 Definition (Dimensions)**.** Each node $v \in V$ is associated with a *width* $w_v \in \mathbb{R}$ and a *height* $h_v \in \mathbb{R}$.

2.20 Definition (Ports)**.** Nodes may have *ports* that are attached to v's boundaries and serve as start- and endpoints for edges.

2.2 Multi-Edge Connections

The node count of a directed graph can be a bound for the maximum edge count. For a directed graph $G = (V, E)$, any node can be connected once to every other node and once to itself. This leads to an upper bound of $|E| = |V|^2$ if self-edges are allowed and $|V| \cdot |V - 1|$ otherwise. However, this limit does not exist for directed graphs that allow *multi-edge connections*. An edge is a multi edge connection if another edge in the graph starts in the same node and targets the same node, as shown in Figure 2.1, where all connections are multi-edge connections. The count of the nodes and edges often determines time complexity estimations for graph algorithms. Consequently, algorithms that have to evaluate some conditions on all edges have a longer execution time. This bound on the edge count can be reestablished for algorithms that apply the same changes for all edges of a multi-edge. This is shown in-depth in Section 3.3.3.

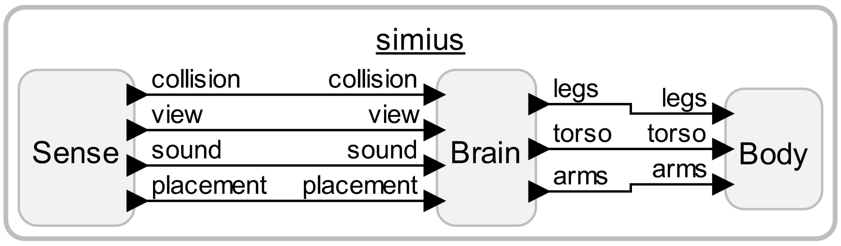


Figure 2.1. A Lingua Franca model where every connection is a multi-edge connection.

2.3 Layout Quality Metrics

The quality of graph layouts is generally referred to as aesthetics, and whether they are considered "good" has been an ongoing research topic in the Graph Drawing community. Some of the earliest researches on the topic of Graph Drawing focused on finding objective metrics for layouts [STT81; TDB88; EHN+17].

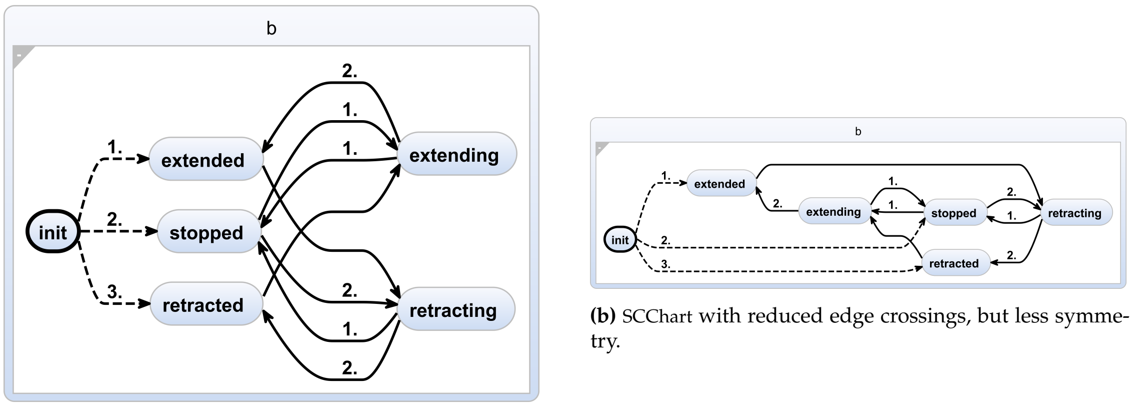
The metric with the most significant impact on the readability of a layout is the count of edge crossings [Pur97]. Additional criteria include the number of edge bends, white-space utilization, edge length, and many more, with their effects extensively researched [PCA02b; WPC+02; PCA02a].

The different metrics used in the analysis in Chapter 5 are given in Table 2.1.

Table 2.1. List of metrics used in the analysis.

Metric	Description
Aspect Ratio	Analyses the dimensions of the diagram, by dividing the total width by the total height.
Backward Edges	Counts the number of edges that go against the layout direction.
Edge Crossings	Counts the number of edge crossings.
Execution Time	Measures the execution time of one or more layout phases.
Multi Edge Connection	Counts the number of edges, where another edge exists with the same start and end node.

Different types of data require different layout algorithms, each having specific use cases in which they excel [DCS+23]. While generally layout algorithms are evaluated for the standard quality metrics, secondary notation and with it the intention is often not evaluated. This complexity increases the difficulty of objectively measuring layout quality, especially considering that some criteria may conflict with others [BRS+07]. Additionally, while these objective measurements serve as a good starting point, certain layout decisions contradict objectively measurable criteria. These decisions often stem from the subjective preferences of individual users [Rie22]. An example of this is illustrated in Figure 2.2. While the underlying model is the same, participants of a survey were evenly split between favoring or disliking the option shown in Figure 2.2a. This graph represents the controls of a piston. The second and third layers in Figure 2.2a create somewhat of a *semantic* grouping. The labels of these nodes seem to be connected by meaning, with the second layer describing the motionless state of the piston, while nested in the third layer are the nodes that describe the behavior of motion for the piston. Additionally, symmetry is essential for human perception, and a study in 2018 showed the effects of symmetry in graph drawing [LKP18].



(a) SCChart with unnecessary edge crossings, but higher symmetry and semantic grouping.

(b) SCChart with reduced edge crossings, but less symmetry.

Figure 2.2. SCCharts used in a empirical study in [Rie22].

2.4 Eclipse Layout Kernel

The Eclipse Layout Kernel (ELK) is a framework that provides a list of different standard automatic layout algorithms (e.g. layered, force-based).

These data types for nodes and edges contain information like the dimensions or the placement of the object. Additional data may be added by using definable properties stored in a map. In ELK, a graph consists of a hierarchical root node that holds the actual graph. This root-node contains a property that indicates the layout algorithm to use for the direct children of this node. Consequently, a different layout algorithm may be employed for each hierarchy level. The layout is created recursively following a bottom-up principle, creating the layout of the child nodes to get the required dimensions of the parent node.

The layout algorithms in ELK are highly modular. This modularity is particularly beneficial for algorithms divided into different layout phases. This modularity enables the easy exchange of algorithms for these phases, as illustrated in Figure 2.3[3]. In addition to the required layout phases, *Intermediate Layout Processors* may be employed. These processors can perform calculations or computations on a given graph that are not part of a specific algorithm. Layout phases may specify the need for a specific layout processor to be executed before or after any given stage. While a phase may create temporary changes to enable later phases to function as intended, these changes should be reversed before the final result is created and presented to the developer. Consequently, the initial phase declares the use of a processor to revert its changes after the phase that required those changes. A representation of the layered algorithm and its phases is shown in Figure 2.6.

Generally, a layout phase and any processor has pre-conditions for the input and guarantees post-conditions for the output. The conditions for the main layout phases of the layered approach are listed in Table 2.2.

[3] `https://eclipse.dev/elk/documentation/algorithmdevelopers/algorithmimplementation/algorithmstructure.html`

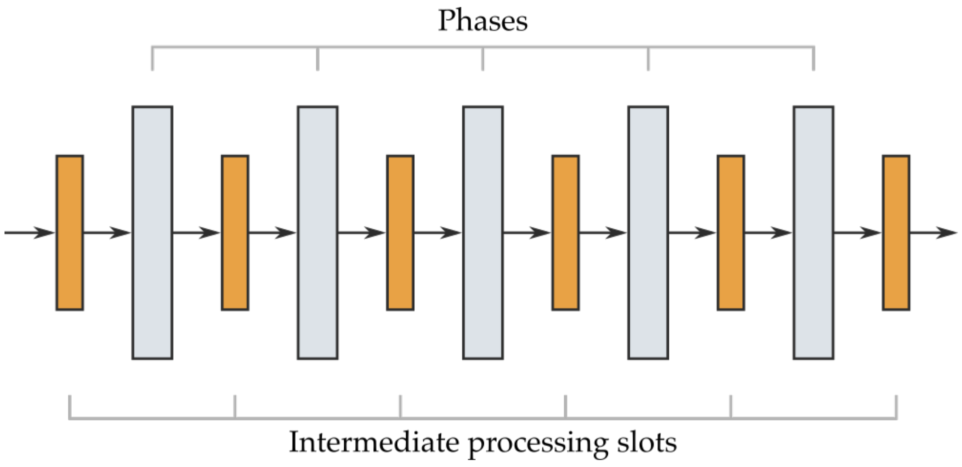


Figure 2.3. Structuring of layout algorithms in ELK.

2.5 Kiel Integrated Environment for Layout Eclipse Rich Client

The Kiel Integrated Environment for Layout Eclipse Richt Client (KIELER) is a research project by the Kiel University's Real-Time and Embedded Systems group, aimed at enhancing graphical model-based designs of complex systems. KIELER provides an *Integrated Development Environment* (IDE) for different modeling languages, with a specific focus on Sequentially Constructive Statecharts (SCCharts) [HDM+14]. KIELER also includes a GUI, which incorporates features for the layout algorithms of ELK, like an interactive diagram viewer, editor, and many more. Additionally, it offers an integrated graph analysis tool called GrAna [Rie10]. GrAna allows the creation of custom graph analysis options, enabling objective measurements for different layout strategies and given aesthetic criteria, as described in Section 2.3.

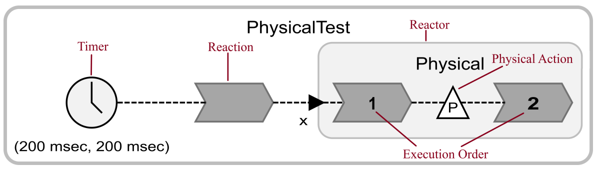
2.6 Lingua Franca

Lingua Franca (LF) is a polyglot reactor-oriented coordination language that facilitates the creation and composition of reactive components. It is not a standalone programming language; instead, it enhances targeted standard programming languages with deterministic reactive concurrency and enables the specification of timed behaviors. Currently, the supported languages are C, C++, Python, TypeScript and Rust. LF provides a model of time that incorporates a clear and precise notion of simultaneity. This feature allows developers to concentrate on solving a specific task without concerning themselves with thread management, synchronization, and race conditions.

Lingua Franca employs various types of nodes, such as *reactors*, *reactions*, *actions* and *timers*, as depicted in Figure 2.4a[4]. The documentation[5] of Lingua Franca lists the following description principles for reactor-oriented programming:

- ▻ **Components**: Reactors can have any of the following elements, which are all *triggers*: input ports, actions, timers. Additionally, they can have any number of the following elements: output ports, local state, parameters, and an ordered list of reactions.

[4] `https://www.lf-lang.org/docs/writing-reactors/actions`
[5] `https://www.lf-lang.org/docs/`



(a) Example of a Lingua Franca Model, with labels for different node types.

```
1  target C;
2  reactor Physical {
3    input x:int;
4    physical action a;
5    reaction(x) -> a {=
6    /* Host Code in C */
7    =}
8    reaction(a) {==}
9  }
10
11 main reactor{
12     p = new Physical();
13     timer t(200 msec, 200 msec)
14
15     reaction(t) -> p.x {==}
16 }
```

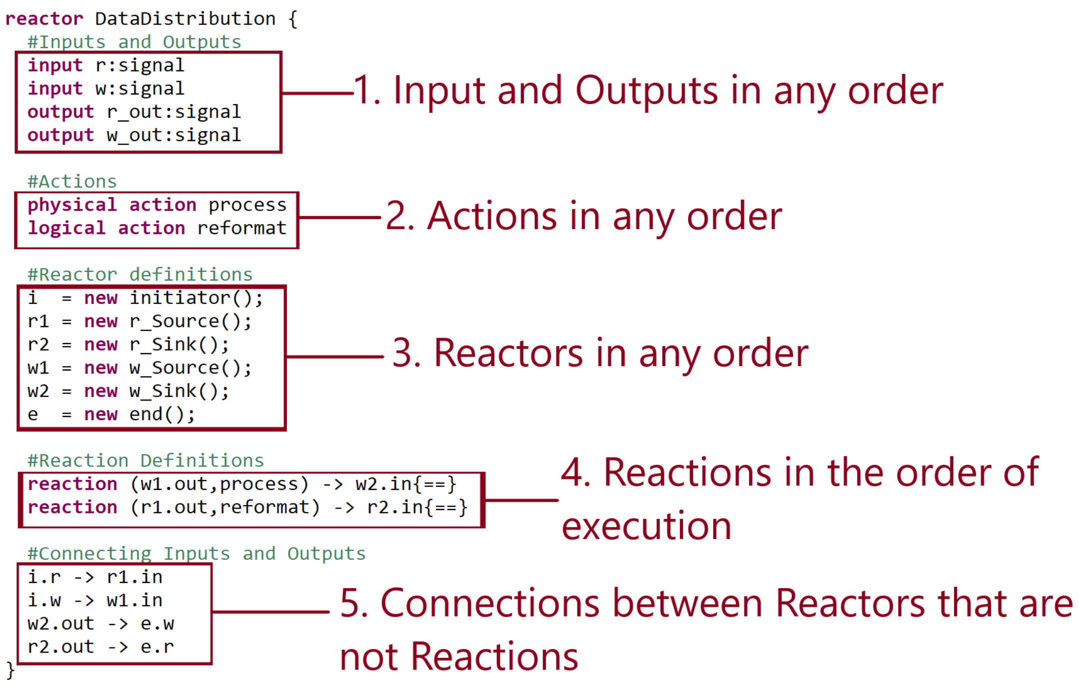
Listing (2.1) Textual definition of the Model. Line 12 declares the reactor "Physical" inside of the main reactor.

Figure 2.4. Example Lingua Franca model and diagram.

- **Composition**: Reactors may contain other reactors and define the connections of their ports. These connections define a message flow in a one-to-many relationship between input and output ports. Connections can be created for reactors contained in the same reactor or connections of a reactor with the parent of the reactor.
- **Events**: Messages between reactors, timer outputs or action events have a tag associated with them (i.e., a timestamp). These *tagged events* may trigger reactions, with the previously defined triggers having at most one event for a given tag. The event may contain a value that is passed along to the reaction.
- **Reactions**: An event may trigger a procedure in the target language as a reaction. This procedure can only be invoked by a trigger event and will never execute without one. A reaction may read input ports, even if these ports are not the origin of the invoking trigger, and it may write to output ports. These ports have to be predefined, as all inputs that are read and all outputs that are produced bear the same timestamp, the timestamp of the triggering event. This simulates a logically instantaneous reaction, meaning all output events are logically simultaneous with the trigger.
- **Flow of Time**: Any successive invocation of a reaction occurs at a strictly increasing timestamp. Messages that are not read by the reaction they triggered at the timestamp of the message are lost.
- **Mutual Exclusion**: Any two reactions in the same reactor are *mutually exclusive* to guarantee deterministic behavior. This means they execute as if they were atomic statements, with respect to one another. To allow multiple reactions in the same reactor to be executed at the same timestamp (simultaneously), the definition order of the reaction in the reactor decides which reaction is executed first. This eliminates race conditions.
- **Determinism**: Lingua Franca is designed to be deterministic unless the design pattern is an explicitly nondeterministic construct. For a given execution cycle of a reactor composition with the same input, the output should always be identical. This results in an environment where distributed, threaded components can be tested for correct behavior.

▻ **Concurrency**: With the dependencies of reactions being part of the explicit declaration of a reaction, threaded execution of independent reactions is possible. With multi-core CPUs, this allows true parallelism of independent code. A reactor may also be declared as `federated`, meaning that execution may be distributed across a network while maintaining the same behavior.

For general readability, a general structure, or style guide, for Lingua Franca is introduced in the documentation. Although not strictly enforced, it is generally adhered to. Figure 2.5 shows an annotated example model in LF. While reactor declarations are more or less freely moveable, they tend to be declared ahead of all reactions, as shown in this figure. This structure introduces a grouping of nodes, where actions, reactions, and reactors are grouped by type. This represents the behavior explained in the introduction and limits the ability to use model order in layout phases, as is utilized in cycle breaking for SCCharts.



```
reactor DataDistribution {
  #Inputs and Outputs
  input r:signal
  input w:signal
  output r_out:signal
  output w_out:signal

  #Actions
  physical action process
  logical action reformat

  #Reactor definitions
  i  = new initiator();
  r1 = new r_Source();
  r2 = new r_Sink();
  w1 = new w_Source();
  w2 = new w_Sink();
  e  = new end();

  #Reaction Definitions
  reaction (w1.out,process) -> w2.in{==}
  reaction (r1.out,reformat) -> r2.in{==}

  #Connecting Inputs and Outputs
  i.r -> r1.in
  i.w -> w1.in
  w2.out -> e.w
  r2.out -> e.r
}
```

1. Input and Outputs in any order
2. Actions in any order
3. Reactors in any order
4. Reactions in the order of execution
5. Connections between Reactors that are not Reactions

Figure 2.5. General structure of a LF file.

While developing programs with LF is possible in any text editor, an interactive diagram view is very helpful, as reported by LF developers. Two options exist for working with Lingua Franca and a dedicated diagram view:

▻ **Visual Studio Code**: *Visual Studio Code* (VS Code)[6] is a source-code editing tool developed by Microsoft. A Lingua Franca Plugin[7] is available in the VS Code marketplace. This plugin depends on the KLighD[8] [SSH13; SSH12] project, which is part of KIELER.

▻ **Epoch IDE**: The Epoch IDE[9] is an Eclipse-based IDE. Epoch also utilizes the KLighD project.

[6] https://code.visualstudio.com
[7] https://marketplace.visualstudio.com/items?itemName=lf-lang.vscode-lingua-franca
[8] https://github.com/kieler/KLighD
[9] https://github.com/lf-lang/epoch

2.7 Layered Algorithm

The layered algorithm, as originally proposed [STT81], has undergone significant improvements [SFH09; Sch11; SSH14]. The variant utilized in this thesis comprises five main layout phases in the following order: *cycle breaking*, *layer assignment*, *crossing minimization*, *node placement*, and *edge routing*. Table 2.2 provides the input and output of these phases. Chapter 3 provides a detailed look into the cycle-breaking phase, and Chapter 4 delves into the crossing-minimization phase.

Table 2.2. Input and output of the layered algorithm phases.

Layout Phase	Input (and Pre-Conditions)	Output (and Post-Conditions)
Cycle Breaking	Directed graph $G = (V, E)$	Directed acyclic graph $G' = (V, E')$, edges may be reversed.
Layer Assignment	Acyclic directed graph $G = (V, E)$	Layered graph $G' = (V', E', L)$ could contain additional dummy nodes and dummy edges to break long edges.
Crossing Minimization	Layered graph $G = (V, E, L)$	G, where all $l \in L$ are ordered sets.
Node Placement (horizontal layers)	Layered graph $G = (V, E, L)$ all layers are ordered sets.	x coordinates for all $v \in V$ for $u, v \in L_i$ and u is ordered ahead of v , it is required that $x_u + w_u < x_v$
Edge Routing (horizontal layers)	Layered graph $G = (V, E, L)$ all nodes have x-coordinates	y coordinates for all $v \in V$ routings for all $e \in E$

In addition to these five layout phases, multiple *Intermediate Layout Processors* are incorporated. Figure 2.6 shows a detailed presentation of the layered algorithm, with all its phases and intermediate processors for ELK. One example here is the REVERSED_EDGE_RESTORER, which is used to reverse the edges reversed by cycle breaking. The intermediate layout processor SORT_BY_INPUT_ORDER_OF_-MODEL presorts the layers to adhere to the model order. The crossing minimization phase utilizes this presorting as a initial "good guess" for in-layer sorting, as this is the order in the textual model.

2.8 Detecting Cycles using Tarjan's algorithm

Some of the cycle-breaking strategies employ Tarjan's algorithm for finding strongly connected components [Tar72], which has a runtime complexity of $O(V + E)$. The graph is traversed using the depth-first approach. Each node receives an index based on exploration time and remembers the lowest index (the low-link) of the reachable nodes. Recursively the outgoing edges of the node are traversed (assigning an index and the low-link). Nodes for which the low-link is smaller than the index are part of a strongly connected component S with $|S| > 1$. While the original approach classifies sets with a power of one as a strongly connected component, for cycle breaking, only strongly connected components of power greater than 1 are relevant. This simple modification is done in the implementation used for the cycle breakers, and the pseudo-code showing the original approach with this modification is shown in the appendix in Listing A.1.

2.9 Stability

Stability can be described by the magnitude of change to the structure of the layout by changing small parts of the model order interacting with the layout. The advantage of working with an interactive layout window is the ability to focus on certain parts of a layout. The terminology *focus & context*, as coined by Card et al. [CMS99], refers to visualization techniques that provide detailed information as

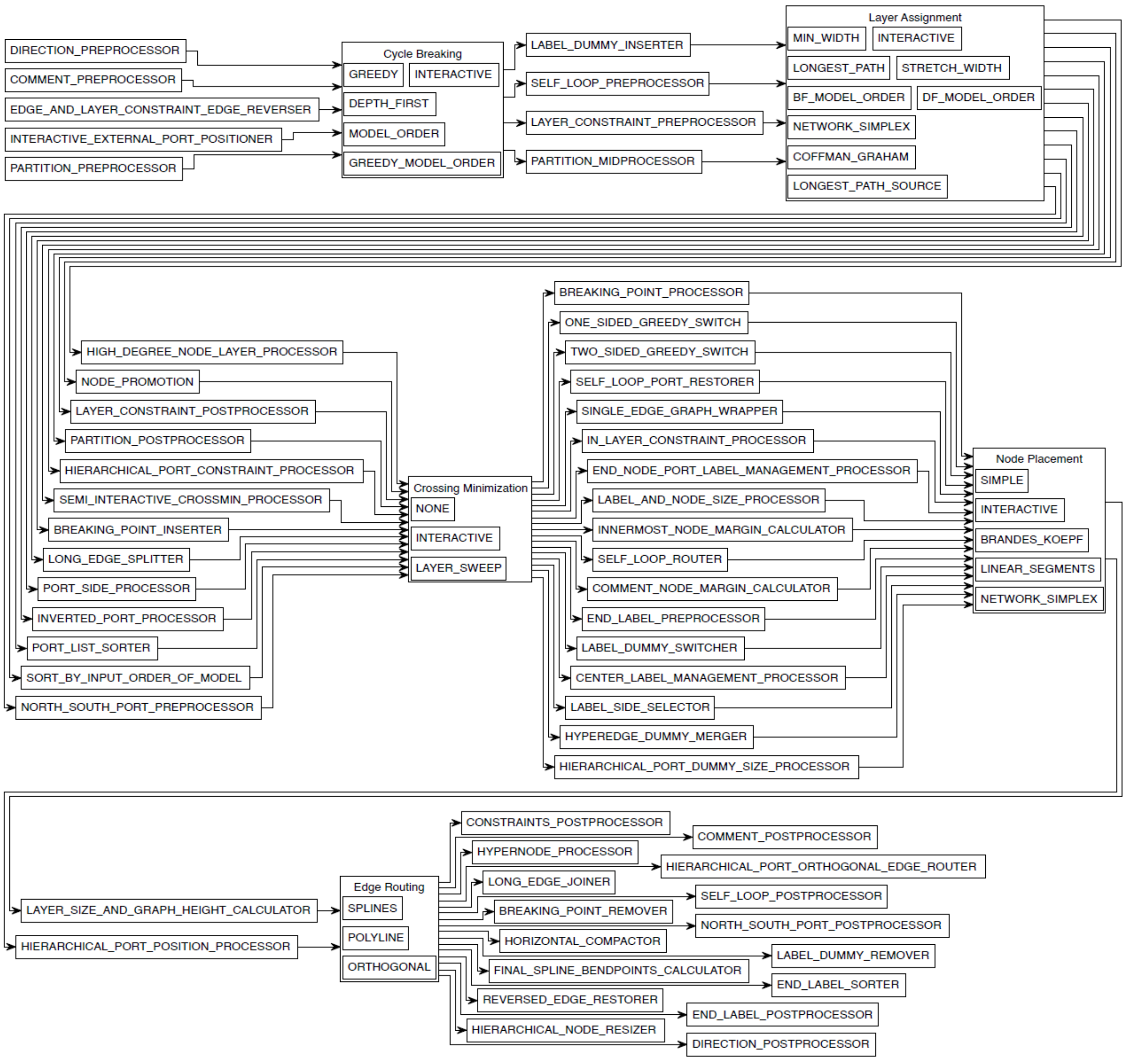


Figure 2.6. All ELK Layered processors, and the different options for the main layout phases [DHS+23].

well as a general overview of the visualized information. According to the authors and the works they refer to, the following aspects are part of focus & context:

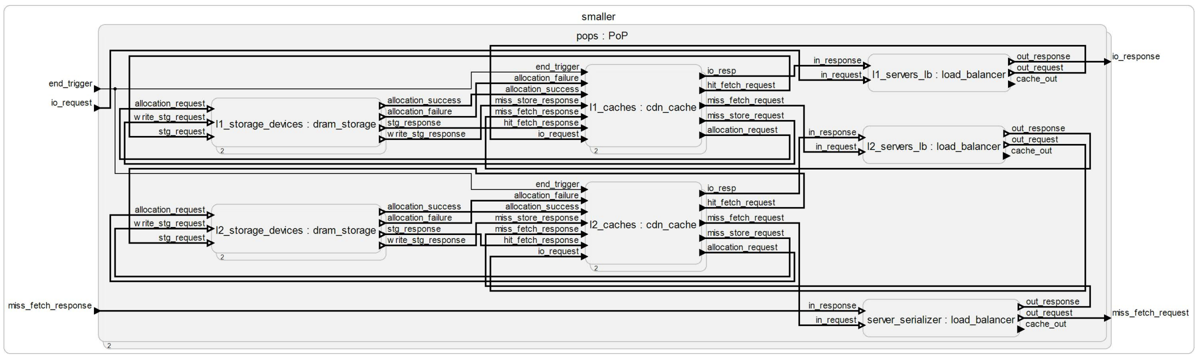
- ▻ The visualization should encourage the viewer to think about the information, be it a broad overview or a fine detail.
- ▻ The visualization tools (i.e. a layout view) requires specific interaction methods for specific datasets.
- ▻ The detailed and broad visualizations can be in a combined dynamic visualization.

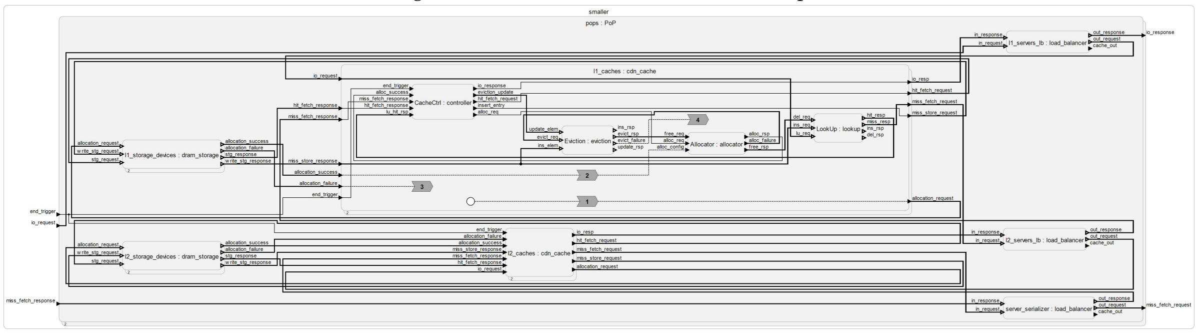
A vital tool for controlling the presented details of graphs with hierarchical nodes is the collapsing and expansion of these hierarchical nodes. *Collapsing* a hierarchical node means hiding the contained elements and showing the node as if there was no internal elements. *Expanding* a hierarchical node means visualizing the contained elements of the hierarchical node.

The crossing minimization phase reorders ports to minimize crossing, this can lead to a stability problem. Expanding and collapsing nodes may force a new order for the ports. This reordering of ports can have big implications for edge-routing and, therefore, the layout in general. I.e., edges may move from the very top of the layout to the bottom. Graph layout stability helps with the preservation of a mental map and the ease of working by reducing the required effort to follow node and edge placements in a layout [PHG06; ZKS11; LCG+15].

Stability is hard to measure, as this cannot be done as a post-layout analysis; rather, it is experienced while actively working with graphs and interacting with them. Furthermore, presenting stability in a paper format is difficult as well, as the motion of a changing graph cannot be depicted in a single snapshot of the graph. Stability issues are much worse for larger graphs, as the position of more nodes and edges may change.

Figure 2.7 tries to show stability issues for one large model. Starting with the initial layout with most nodes collapsed in Figure 2.7a. Figure 2.7b shows the layout result for the model with a fixed port order and the node `l1_caches: cdn_cache` expanded. Except for the expansion of this node, no changes have been made. The remaining nodes remain close to their original position, and the order of edges and nodes remains the same. Figure 2.7c shows the model with the same node, the node `l1_caches: cdn_cache` node. While it looks like the node `l2_caches: cdn_cache` was expanded, these nodes have switched positions, as well as the two `dram_storage` nodes in the first layer. The position changes conflict with the mental map of the layout, and it requires time to discover where the nodes are positioned now.


(a) Lingua Franca model with most nodes collapsed.


(b) Expanding the top `cdn_cache` node with fixed port order.

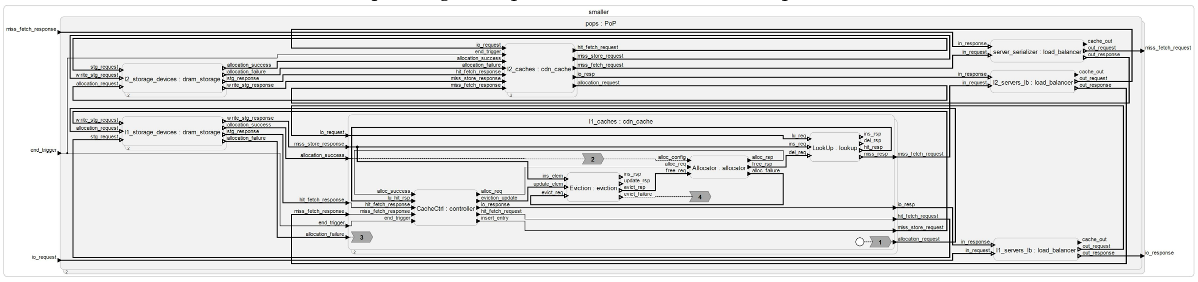
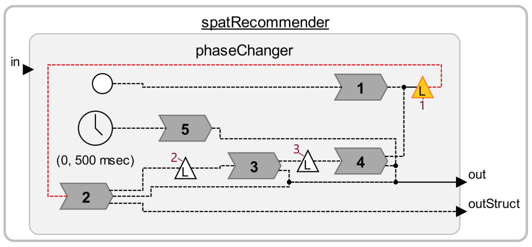

(c) Expanding the top `cdn_cache` node without fixed port order.

Figure 2.7. Enforcing port order and increasing stability.

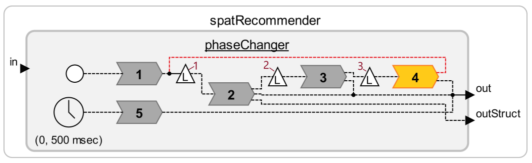
Chapter 3

Cycle Breaking

As mentioned in Section 2.7, the cycle-breaking phase is the initial phase of the layered layout algorithm. This chapter explains the fundamentals and various strategies of cycle breaking, along with their *Worst-Case Execution Time* (WCET) in Big $\mathcal{O}$ notation. The examples in this chapter are extracted from larger models to reduce the size and highlight the import parts. As shown in the introduction, in Figure 1.7, the use of the raw model order is not sensible. Different approaches for cycle-breaking are introduced here, using different approaches to cope with the textual grouping.



(a) Reversing the edge from a logical action to reaction 2.



(b) Reversing the edge from reaction 4 to a logical action.

Figure 3.1. Two different options of cycle breaking, with highlighting for the reversed edges and their starting nodes and annotation of the model order of the actions.

3.1 Fundamentals

Fundamentally, cycle-breaking is about creating an acyclic graph. The cycle-breaking phase is based on finding a feedback set. While finding a feedback set of arbitrary size is easy, finding a minimal solution is proven NP-complete [GJ79; CTY07]. Consequently, the strategies employed in the layered algorithm are not designed to find minimal solutions with perfect accuracy. However, they are designed to find a minimal solution in a reasonable time. Figure 3.1 shows two different options for cycle breaking for the same graph. This visualization shows that cycle breaking modifies the *flow* of the layout. The option presented in Figure 3.1b delays the edge reversal to the latest possible node, interrupting the flow at the latest point. The other option creates a backward edge as early as possible, reducing the diagram's width. A backward edge could be intentional to represent a feedback loop. These intentions are secondary notation and may be extracted from the textual order and utilized. This use of the model order enhances the flow from a mere graphical artifact to an expression of intentional data-flow in the model.

The different strategies explored in this section have different approaches to finding *good* edges to reverse. Generally, edge reversals should be kept as minimal as possible, as these create backward edges in the final layout, reducing the layout's readability. However, different approaches can find a solution of the same size but with different edges. Identifying which solution is the best is ambiguous.

All strategies traverse the graph in some way, and reverse edges until it is ensured that no cycles remain. This traversal can utilize model order, i.e., selecting the next node based on model order. This order of node traversal is called *discovery order*. The discovery order can induce problems. The *discovery order problem* is defined as follows: A node that is not part of a cycle is discovered but has a connection to a node with an earlier discovery time. Thus, the edge to the already discovered node is reversed when discovering the node. An example of this is shown in Section 3.3.1 and visualized in Figure 3.4, where the edge from `intermediate_2` to `end` is reversed, as `end` has an earlier discovery time.

The following sections introduce the different approaches. The order does not imply that a later strategies explicitly tries to improve a previous strategy. The evaluation of the strategies is given in Chapter 5. Some of these strategies define a strict order based on the group type while other strategies use the node type to modify the behavior.

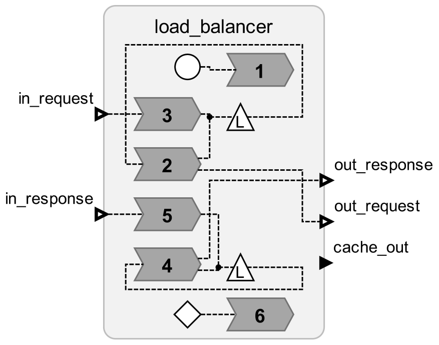
3.2 Greedy Cycle Breaker

The main design focus of the *Greedy Cycle Breaker* (G-CB) is to find small feedback sets [ELS93]. This strategy uses model order as a tie-breaker, for a heuristic metric called *outflow*, as defined in Definition 2.7. An order is defined using the outflow, as shown in Listing 3.1. In the original approach a node is chosen randomly if multiple nodes have the same outflow. The seed for this random node choice is fixed to ensure the same layout on all devices.

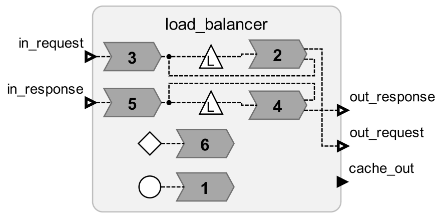
```
unprocessedNodes = V
sinkOrder = []
sourceOrder = []
nodeOrder =[]
while unprocessedNodes not empty:
   for v in unprocessedNodes.sinks:
      sinkOrder.append(v)
      unprocessedNodes.remove(v)
      updateAdjecentNodes(v)
   for v in unprocessedNodes.sources:
      sourceOrder.append(v)
      unprocessedNodes.remove(v)
      updateAdjecentNodes(v)
   maxOutflowNodes = findMaxOutflowNodes(unprocessedNodes)
   v = maxOutflowNodes.random()
   nodeOrder.append(v)
   unprocessedNodes.remove(v)

function updateAdjacentNodes(v):
   //... update the in- and outdegree of adjacent nodes, as if v would not exist in G

function findMaxOutflowNodes(v):
   //... loop over unprocessedNodes and return a list of nodes with the max outflow
```

Listing 3.1. Pseudo code showing an order extraction using the outflow.

The algorithm reverses edges that go against the order induced by the outflow. The approach for utilizing model order in SCChart cycle breaking was presented by Domrös et al. [DRv23]. This approach uses the model order as a tiebreaker for nodes with the same outflow. Utilizing model order for languages that group nodes based on their types results in selecting the node type that is defined first. Ultimately, this selection leads to similar problems, as the standard model order approach mentioned in Section 1.3.1, if the two nodes of different groups have the same outflow. Figure 3.2a shows the same model as shown in Figure 1.7 but using the G-CB. Figure 3.2a shows that the layouts with this strategies enforce edge reversals strictly from one type to another, creating more edge crossings.



(a) Limitations of the G-CB.



(b) Cycle-breaking considering group order.

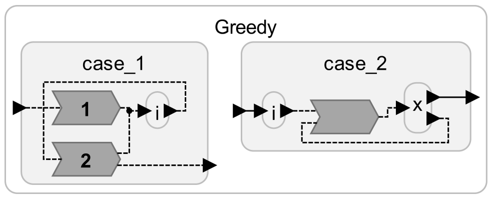
Figure 3.2. Limitations of the standard G-CB.

The following approach is suggested and implemented to introduce group model order:

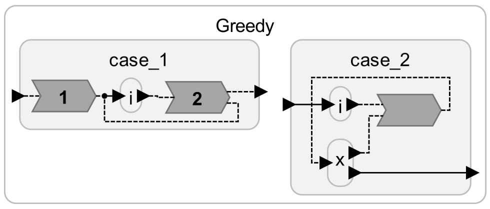
▻ Define an additional order for the different node types.

▻ If a node has to be chosen from multiple nodes with the same outflow, choose all nodes of the type with the highest priority and use the node with the minimal model order with this type.

With this approach the G-CB can achieve the same layout as shown in Figure 3.2b (also used in the introduction). However, the order defined for the node types does not eliminate the problem of strictly reversing edges between different node types. Figure 3.3 shows that altering the fixed order can improve some layouts, but switching the types of all nodes recreates this problem.



(a) Limitations of the fixed order for G-CB.



(b) Cycle-breaking considering group order.

Figure 3.3. Limitations of the G-CB with a fixed group order.

Both approaches have a runtime complexity of $\mathcal{O}(V + E)$.

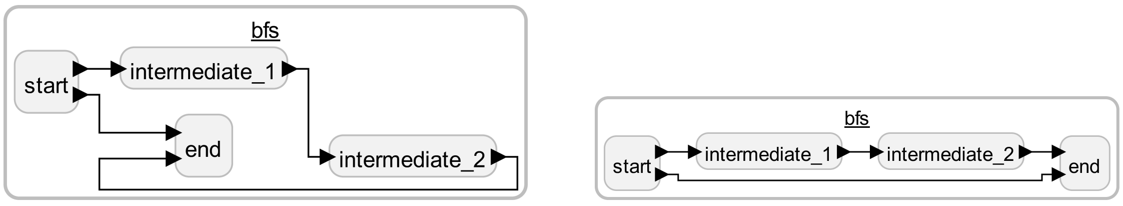
3.3 Breadth-First Cycle Breaker

The *Breadth-First Cycle Breaker* (BF-CB) utilizes the *Breadth-First Search* (BFS) to traverse the graph, remembering which nodes have been visited. If an edge of the current node ends in a node that was previously visited, a cycle is identified, and this edge is reversed. Since BFS finds the shortest path, the layout tends to be compressed, with the drawback of creating more edge-crossings with a high probability [Rie22].

The standard implementation of the BFS has additional drawbacks if employed in cycle breaking. The following two subsections explain and solve these problems. Finally, two different options for the discovery orders are given.

3.3.1 Sources and Sinks

Edges that start (end) in sources (sinks) can never be part of a cycle since sources and sinks can, by definition, never be part of a cycle. Figure 3.4a shows an ONO example regarding cycle-breaking, where the edge from `intermediate_2` to `end` is reversed, even though the graph is inherently acyclic.



(a) ONO layout, due to unnecessary edge reversal with a sink.

(b) OYES layout, due to eliminating the easily identifiable unnecessary edge reversal.

Figure 3.4. Difference of filtering edge reversals for sink and source connections.

The problem arises due to the discovery order problem. In terms of BFS, the discovery time is linked to the shortest path from the source of the search. For Figure 3.4a, this problem arises as the node discovery happens as follows: the initial step of BFS discovers the `start` node, followed by the nodes labeled `intermediate_1` and `end`. Finally, the `intermediate_2` node is discovered in the second step, and reversing the edge connecting to `end` since it leads to an already discovered node. Figure 3.4b shows an OYES layout, achieved by adjusting the BF-CB to check if the target (starting) node is a sink (source), and if this is the case, do not reverse any of the edges.

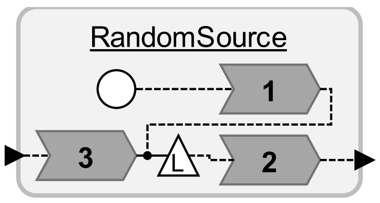
3.3.2 Cycle Detection Pre-processing

The same problem, as mentioned in the previous subsection, may also happen for nodes that are not sinks or sources. There are two reasons why a single BFS could not suffice to traverse the entire graph:

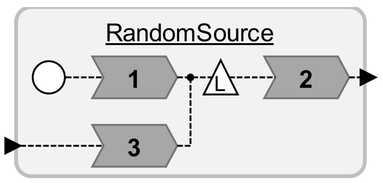
▻ The graph has detached parts, running a BFS on one of these parts does not traverse the others.

▻ The graph has multiple sources, running a BFS from one source does not discover any other sources or any nodes only reachable by different sources.

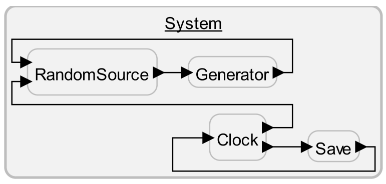
For the detached parts, cycle breaking is applied independently. However, for the case of multiple sources, discovery order problems can arise again, as seen in Figure 3.5a, where the edge from `1` to `L` is reversed. Figure 3.5b shows that introducing a flag for nodes that indicates if they are part of a cycle



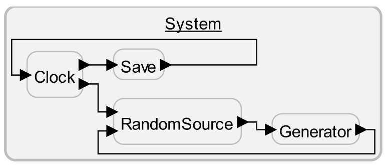
(a) Unnecessary edge reversal due to discovery order problem.



(b) Eliminating the unnecessary edge reversal by only allowing edge reversals of edges in a cycle.



(c) Unnecessary edge reversal due to the discovery order problem of a node in a different SCC.



(d) Eliminating the unnecessary edge reversal by allowing edge reversals only within the same SCC.

Figure 3.5. Improving the layout by filtering edge reversals for connections that are not part of a (same) cycle.

and not reversing edges if the source or the target node is not part of a cycle eliminates the discovery order problem. Furthermore, in Figure 3.5c the edge from `Clock` to `RandomSource` is reversed, as both nodes are flagged as part of a cycle. Figure 3.5d shows that eliminating unnecessary reversals can be done by remembering which SCC the node is a part of and only allowing edge reversals if the starting and target nodes are part of the same SCC. This eliminates the need for checking if a node is a source or a sinks, since these nodes are never part of a cycle.

3.3.3 Edge- or Node-Order

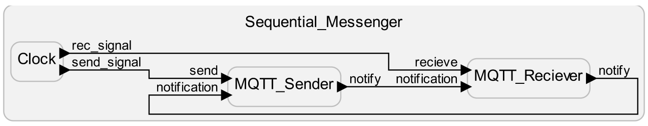
A way to alter the BF-CB is the order of discovery, if multiple options are available. The following options are possible:

- ▻ Discover by the order of edge-declarations.
- ▻ Discover by the order of the target.

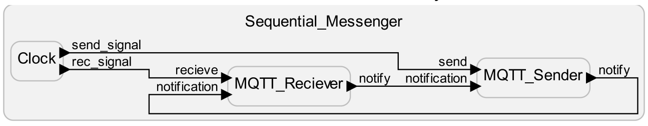
The first option requires that edges have an assigned model order. The second option requires that all nodes have a model order. Figure 3.6 shows a model and the layout variants for using node or edge order. The layout in Figure 3.6a uses the node order, therefore, the node `MQTT_Sender` is discovered first. The layout in Figure 3.6b uses the edge order, therefore, the node `MQTT_Reciever` is discovered first. While the edges declaration order in this model is freely modifiable, the edge order for edges connected to reactions is dictated by the declaration order of the reaction. The order of reactions is not freely modifiable, which means that for reactions both approaches fail to introduce control to modify the layout. However, for connections between reactors both versions work.

```
reactor Sequential_Messenger{
  clock = new Clock()
  sender = new MQTT_Sender()
  reciever = new MQTT_Reciever()


  clock.rec_signal -> reciever.recieve
  clock.send_signal -> sender.send

  sender.notify -> reciever.notification
  reciever.notify -> sender.notification
}
```

Listing (3.2) Model with crossing declaration for node and edge order.



(a) Use node order for discovery.



(b) Use edge order for discovery.

Figure 3.6. Comparison of layouts that prefer node or edge order using the BF-CB.

Listing 3.3 shows how the order of discovery is modified to the node model order. This implementation shows one approach to reduce the size of edges to evaluate, in lines 5-13. Here, edges are grouped into a set of edges if they connect to the same target. Later, the strategy evaluates only one edge of every set and applies the decision to reverse all edges of the set based on the decision for the representative edge. This grouping effectively limits the edges to check for BFS to $E \leqslant V^2$, the upper limit for fully connected digraphs that do not allow multiple edges to start and end in the same nodes. While this does not reduce the runtime approximation in Big $\mathcal{O}$ notation, it reduces the real execution time by reducing the size of E. Algorithms could utilize this strategy in steps that are indifferent to singular or multiple connections between two nodes, like cycle breaking or layer assignment. However, it would require heavy restructuring of these algorithms.

```
1  v = currently inspected node
2  modelOrderDict = new Dict<Integer, Set of Edges>
3
4  maxVal = Integer.MAX_VALUE
5  for e in v.outgoing:
6     model_order = e.target.model_order
7     if model_order is null:
8        modelOrderDict[maxVal--] = new Set().add(e)
9     else:
10       if modelOrderDict[model_order] is null:
11          modelOrderDict[model_order] = new Set().add(e)
12       else:
13          modelOrderDict[model_order].add(e)
14
15 sortedKeys = modelOrderDict.keys.sort()
```

Listing 3.3. Sort outgoing edges by the model order of the target node.

3.3.4 Complexity

The complexity of raw BFS is $\mathcal{O}(V + E)$. The mentioned modifications alter the complexity as follows:

- ▻ Sources and Sinks: Adding the check if a node is a source or sink does not add to the complexity. Nodes have a list for in-coming and out-going edges. Checking if a node is a source (sink) is done by checking if the out-going (in-coming) list is empty. $\mathcal{O}(V + E)$
- ▻ Cycle Detection Pre-processing: Tarjan's algorithm also has the complexity of $\mathcal{O}(V + E)$, resulting in $\mathcal{O}((V + E) + (V + E)) = \mathcal{O}(2(V + E)) = \mathcal{O}(V + E)$

Ordering the edges by their model order or the model order of their target node does not make a difference concerning runtime complexity. Both cases require that the out-going edge list of each node is sorted which comes with a runtime of $\mathcal{O}(E \log E)$. In detail explained by lines of Listing 3.3:

- ▻ Lines 5-13: Each outgoing edge is handled once, giving a complexity of $\mathcal{O}(E)$
- ▻ Line 15: Creating a sorted-set from an unsorted set requires the usual time complexity of sorting a set of values $\mathcal{O}(E \log E)$.
- ▻ Wrapper: Since this is run in a loop over all nodes a fully connected graph creates a runtime of $\mathcal{O}(V \cdot (E \log E))$

The resulting complexity scales poorly, especially for graphs with high connectivity.

3.4 Depth-First Cycle Breaker

The *Depth-First Cycle Breaker* (DF-CB) uses the *Depth-First Search* (DFS) to traverse the graph. Edges that end in a node on the *active path* are reversed. The active path is the path from the source node to the currently inspected graph. DFS has the advantage of having a dynamic active path, which automatically prevents unnecessary edge-crossings, as only edges to nodes in the active path are reversed and these edges always indicate a cycle.

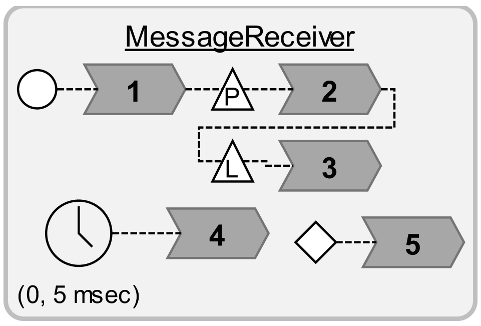
Similar to the BF-CB, the discovery order can be altered based on the model order of the edge or the target, resulting in the same runtime complexity ($\mathcal{O}(V + E)$) and effects, and restrictions for reactions, on the discovery order. The DF-CB performed extremely well in the analysis for SCCharts [Rie22]. Additionally, the DF-CB created the same layouts as the strict model order approach in the most cases, which hints at a mental map in a depth-first layout.

3.5 Model Order Cycle Breaker

The *Model Order Cycle Breaker* (MO-CB) was initially created for SCCharts and is designed to utilize secondary notation and increase the control for the developer. As described in Section 1.3.1, nodes receive a model order based on declaration order in the textual file, as shown in Listing 3.4. If an edge starts in a node with a higher model order than the target node, it is reversed. The MO-CB may achieve any feedback set that another strategy creates for graphs where node declarations are freely movable [Rie22]. This gives the developer full control regarding backward edge creation. However, for languages where different nodes are grouped by their type this approach does not work. Figure 3.7a shows an example of this. As the reaction with label 2 is declared below the logical action, the edge connecting these nodes is declared as a backward edge. The MO-CB creates unnecessary backwards edges, as using it with the restricted model order does not work as intended. The following strategy, called Strict Type Cycle Breaker has a similar problem with the layout creation. The MO-CB has a complexity of $\mathcal{O}(E)$.

```
reactor MessageReceiver {
  physical action ros_message_a:
        instant_t
  logical action ros_message_l

  timer t(0, 5 msec)

  reaction(startup) ->
       ros_message_a
  reaction(ros_message_a) ->
       ros_message_l
  reaction(ros_message_l)
  reaction(t)
  reaction(shutdown)
}
```

Listing (3.4) Model used to show limitations of the MO-CB



(a) Unnecessary edge reversal using the MO-CB.

Figure 3.7. A small model that shows problems for the MO-CB

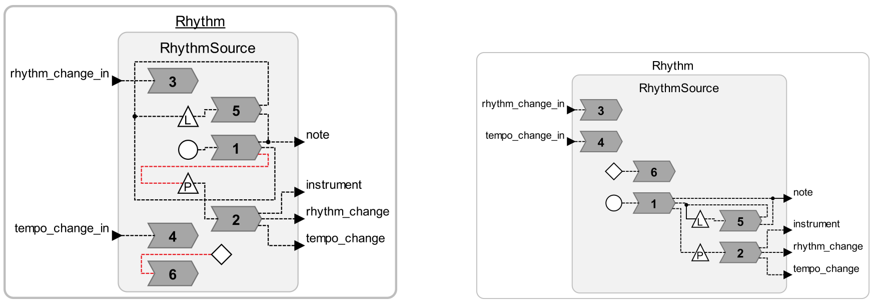
3.6 Strict Type Cycle Breaker

The *Strict Type Cycle Breaker* (ST-CB) utilizes the MO-CB for nodes of the same type and enforces a strict relation for connection between different node types. I.e., reverse all edges from type `A` to type `B`, as seen in Listing 3.5, where all edges from reactions to actions are reversed. For nodes where the declaration order is freely moveable using model order works as intended. However, for nodes like reactions the order is part of the semantics, removing the free moveability.

```
for (LNode source : layeredGraph.getLayerlessNodes()) {
   int modelOrderSource //...
   int groupIDSource //...
   source.getOutgoingEdges().forEach(edge -> {
      int groupIDTarget = //...
      int modelOrderTarget = //...
      // If groups do not match, use groupID
      if (groupIDTarget < groupIDSource) {
         revEdges.add(edge);
      }
      // If groups match use Model Order
      else if (modelOrderTarget < modelOrderSource && groupIDTarget == groupIDSource) {
         revEdges.add(edge);
      }
   });
}
```

Listing 3.5. Implementation of the ST-CB.

Modeling languages where the node-type switches occur frequently rarely utilize the MO-CB, as the strict order between nodes is used, leading to strict segregation of nodes into different layers, as seen in Figure 3.8a. Figure 3.8b shows a layout for the same model with the BF-CB. While the ST-CB cycle breaking approach might be sensible for other languages, for LF, with high probability, this results in bad layouts. Therefore, this approach will not be explored further here but should be remembered for exploration in other languages.



(a) ONO layout created with the ST-CB. This creates unnecessary backwards edges.

(b) OYES layout created with the BF-CB.

Figure 3.8. Issues of the ST-CB for languages like LF.

3.7 Model Order Look Ahead Cycle Breaker

The *Model Order Look Ahead Cycle Breaker* (MOLA-CB) utilizes the group model order by only comparing model order for nodes of the same type, outsourcing the decision to a subsequent edge that ends in a node with the original node type. If an edge that starts in type A goes to type B, the algorithm begins a search in all subsequent paths for the next node with type A. Listing 3.6 shows pseudo-code for this. The search is based on a BFS, using a queue to check the subsequent nodes. The base version of this strategy can be modified with the options described in the following sections, which tackle different problems. However, finding a good balance between these options to create good layouts for all Lingua Franca models used in the analysis is difficult, as some model layouts are better with different options than other models.

3.7.1 Reducing Edge Reversals

Using the model order to determine edge reversals can lead to unnecessary edge reversals for groups with restrictions on the declaration order. Figure 3.9a and Figure 3.9b shows an example, where the reversed edges are not required, but induced by the semantically enforced model order. Figure 3.9c shows that using Tarjan's algorithm as a pre-processor and only reversing edges if the starting and ending node are in the same SCC can eliminate these reversals.

3.7.2 Preferred Type For Order

With multiple node types, it is possible that the order of some types is of higher relevance for the developer. Defining an order for the different node types and using this order in cycle breaking can incorporate this order. While the ST-CB uses this order strictly between different types, in the MOLA-CB the order of priority indicates the order of cycle-breaking by type. This can remove cycles with different node types but this is not as strict.

Let type A be of higher relevance than type B. Filtering the nodes in V to only include nodes of type A in line 6 of Listing 3.6 modifies the algorithm to only compare nodes of type A and break

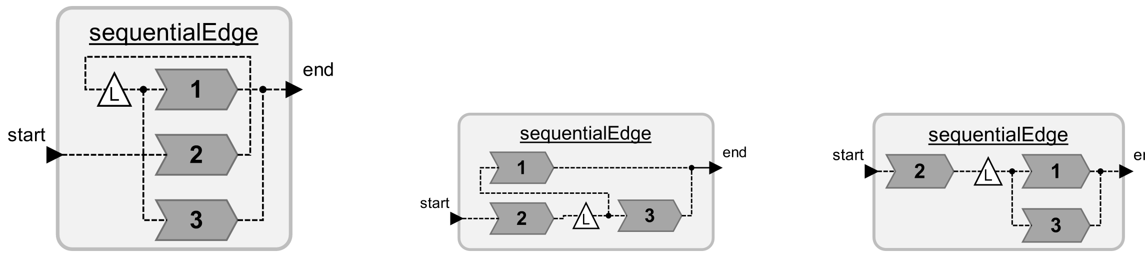
```
input: directed graph G = (V,E)
output: acyclic graph G‘ = (V,E‘)

subsequentNodes = Queue of Nodes

for v in V:
   for (v,w) in E: // accessible by v.outgoing
      if v.type == w.type :
         //reverse (v,w) if v.modelorder <= w.modelorder
         compareByModelOrder((v,w),w)
      else:
         w.visited = true
         if findNextNodes(v):
            (v,w).reverse()


function findNextNode(v):
   while subsequentNodes not empty:
      w = subsequentNodes.dequeue()
      for (w,x) in E:
         //node of same group and reversal inducing
         if v.type == x.type and v.modelorder > x.modelorder:
            return true
         //not the same type, add the subsequent nodes for evaluation.
         if v.type != x.type and not x.visited:
            x.visited = true
            subsequentNodes.enqueue(x)
   //No node of the same type or with a lower model order was found, dont reverse (v,w)
   return false
```

Listing 3.6. Pseudocode of the basic MOLA-CB implementation.

cycles based on this type's model order. After all nodes of type A are handled, the nodes of type B are handled to ensure an acyclic graph. However, all cycles containing nodes of type A and B have already been cleared based on the preferred model order. For languages like Lingua Franca, where a specific type has restrictions on the declaration order (reactions), the order should prioritize nodes without restrictions, as this increases the control of the developer.

3.7.3 Skip Sequential Edges

Instead of reversing the immediate outgoing edge of a node, reverse the incoming edge of the last node on the path to the node of the same type. This delays the creation of backward edges, moving backward edges to a later point of the diagram and creating a linear beginning, as seen in the differences between Figure 3.9a and Figure 3.9b. This option can increase the number of edge reversals, as every path that has a model order violation creates a new backward edge. However, as seen in the outgoing edge from 2 in Figure 3.9a, an initial outgoing edge can have contradictory preferences regarding the reversal for different paths. The edge (2,L,1) dictates a reversal of the outgoing edge from 2, while the edge (2,L,3) prefers the edge not to be reversed. Regarding the pseudo-code in Listing 3.6, instead of returning a boolean value, the function findNextNode is altered to return the sequential edges that should be reversed. This moves the flow interruption to a later point, creating layouts that follow the layout direction longer.



(a) Layout using the basic version of the MOLA-CB. The outgoing edge of reaction 2 is reversed, since reaction 1 has a lower model order and is on a subsequent path

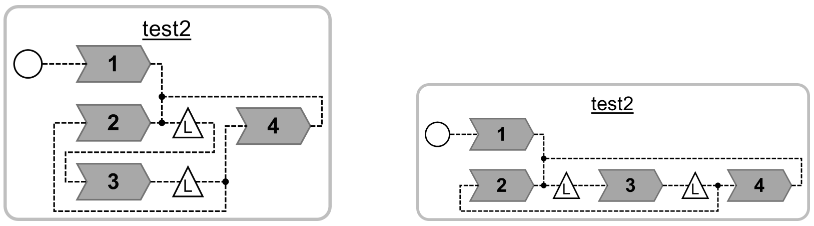
(b) Layout using the MOLA-CB using the skip sequential edges option. The incoming edge of reaction 1 is reversed, since this edge violates model order.

(c) Layout using the MOLA-CB with Tarjan's algorithm as preprocessor. The edge reversal is removed, since the graph is inherently acyclic.

Figure 3.9. Layouts using the basic MOLA-CB or using the skip sequential edges option or with a Tarjan's algorithm pre-processor.

3.7.4 Fallback Edges

Contradicting preferences for edge reversals can also form for the sequential edges. The layout in Figure 3.10a shows such a conflict; the path (2,L,3) does not require a reversal based on model order but the path (4,L,3) introduces the edge reversal of (L,3). The outgoing edge of 4 is later reversed due to the declaration order of the logical actions. An option to prevent the contradicting preference for sequential edges would be fixing the direction of incoming edges if they obey model order for the lowest connecting model order. The initial outgoing edges are remembered, and if any subsequent path leads to a fixed edge, only the initial edge is reversed, as shown in Figure 3.10b. The path (2,L,3) fixes the edge direction of (L,3), forcing the fallback to (4,L). This can be done by altering Listing 3.6, keeping a list of edges to reverse. Finally, if all paths are explored and `subsequentNodes` is empty, reverse all edges in the list. If an edge that has a fixed direction is encountered that violates the model order, during the traversal of all paths, clear the list and only reverse the initial edge.



(a) Layout using MOLA-CB with skip sequential edges option. The incoming edge of reaction 3 is reversed due to the connection of reaction 4.

(b) Layout using MOLA-CB with skip sequential edges and fallback edges option. The outgoing edge of reaction 4 is reversed, as it is the fallback edge and the direction of the incoming edge of reaction 3 is fixed.

Figure 3.10. Reversal conflicts for the MOLA-CB.

3.7.5 Complexity

This strategy initially visits every node and edge once for a runtime of $\mathcal{O}(V+E)$. A new search is started for every connection to a different node group. If an edge is reversed, it is not considered for reversal in any following run, reducing the search space for subsequent nodes. The worst case is a fully connected graph, where every node is of a different type. This strategy has a runtime approximation of $\mathcal{O}(V \log V + E \log E)$. For LF, the search process for subsequent edges of the same type is negligible, as only three relevant node types exist. As mentioned, nodes that are sinks or sources can be skipped since they cannot be part of a cycle, which applies inherently to startup, shutdown and timer nodes. As reactors tend to be connected to other reactors, and actions or reactions can not follow their node type, enforcing the switch on every other node, while this does not change the runtime approximation for LF it behaves like a approximation of $\mathcal{O}(V+E)$, since the breadth-first searches most likely only need to look at the second layer.

3.8 Strongly Connected Component Cycle Breaker

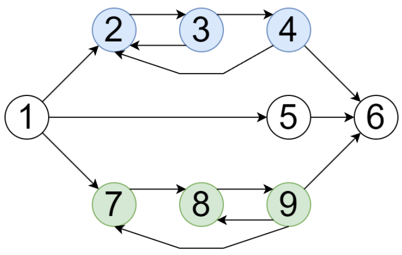
The final cycle breaking strategy is the *Strongly Connected Component Cycle Breaker* (SCC-CB). This strategy uses Tarjan's algorithm multiple times. The idea is to identify strongly connected components and choosing a "good" node to reverse edges in each SCC to effectively remove the cycles.

3.8.1 Node Selection

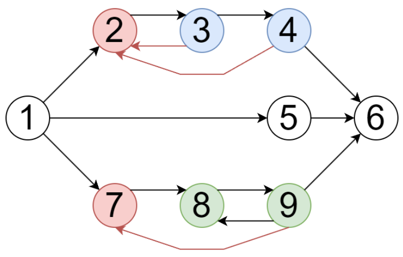
The following section explores multiple approaches for selecting "good" nodes, starting with general approaches applicable for any modeling language that uses model order — followed by an option to alter the behavior for languages with node type grouping. The first two approaches in this section explore the selection of model order maxima (or minima) in the SCC. The third approach explores the selection of a maxima or minima based on the in- and out-degree inside the SCC, combining the previous approaches to reduce the repetitions of Tarjan's algorithm.

Minimum Model Order

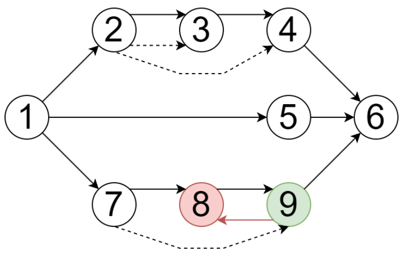
The first approach finds the node with the minimum model order. Since model order indicates that this node should be placed as far to the left as possible, all incoming edges coming from a node in the same SCC are reversed. This selection process can be incorporated into Tarjan's algorithm for better efficiency. This approach is visualized in Figure 3.11. The node labels describe the model order of the node. For this example graph, Tarjan's algorithm has to be run thrice. Initially identifying strongly connected components in `Step 1`, the repetition of this `Step 1`, shown in Figure 3.11c, and finally, to guarantee that the graph is acyclic. This number of repetitions is not optimal for this model, as two could suffice, shown in Section 3.8.1.



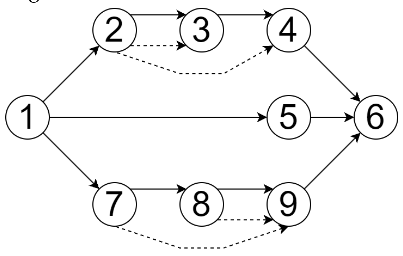
(a) Step 1: Identify strongly connected components.



(b) Step 2: Find node with minimum model order and reverse incoming edges.



(c) Repeat step 1 and 2. Reversed edges are dashed.

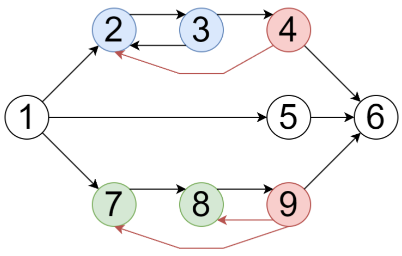


(d) Acyclic Graph.

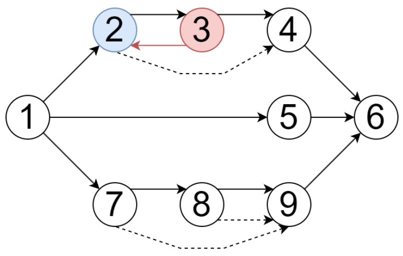
Figure 3.11. Illustration of the SCC-CB, using minimum model order selection.

Maximum Model Order

The maximum model order node selection selects the node with the maximal model order in the SCC. Since the definition of this node is the latest definition for all nodes in the SCC, it should be placed right-most, which is achieved by reversing all outgoing edges to nodes of the same SCC. The different behavior this has is illustrated in Figure 3.12, altering `Step 2` to `Step 2'`. For this example, the number of repetitions of Tarjan's algorithm remains the same, but it is not optimal.



(a) Step 2′: Select the node with the maximum model order and reverse outgoing edges.



(b) Repeat step 1 and 2′

Figure 3.12. Illustration of the SCC-CB, using maximum model order selection.

Connectivity Model Order

Combining the minimum and maximum model order selection can be achieved by using the in-degree and out-degree of the nodes. For each SCC, the minimum and maximum model order nodes are saved.

If the minimum node has a higher in-degree (filtered to nodes of the same SCC) than the maximum node's out-degree (filtered to nodes of the same SCC), it is chosen. Otherwise, the maximum node is chosen. Figure 3.13 illustrates this. This approach reduces the required repetitions for the example graph to two. If the in-degree equals the out-degree for the respective nodes either option can be chosen, as both approaches reverse the same edges, just in a different order. The connectivity approach tries to reduce the execution time.

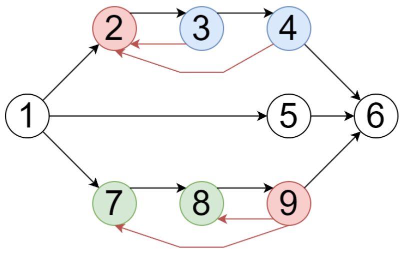


Figure 3.13. Illustration of the SCC-CB, using connectivity model order selection.

Behavior by Node-Type

For languages with multiple node types and grouping, these options so far do not deal with the problem of incomparable model order. However, the node type may alter the node selection behavior by using the same approach explored in the "preferred type for order" option of the MOLA-CB, explained in Section 3.7.2. A specific type can be prioritized, i.e., for Lingua Franca, this order could be defined as follows:

- ▻ If the minimum and maximum model order nodes are of the same type, choose the node according to the in- and out-degree.
- ▻ If the minimum node is a reactor and the maximum node is a different type, choose the minimum node.
- ▻ If the maximum node is a reaction, select this node.

This order allows that reactors, which are freely movable, are prioritized, to increase the control.

As a conclusion to the different options, the connectivity model order selection combines the minimum and maximum approach and reduces execution time, therefore, this approach is used in the analysis. The behavior by node-type approach introduces a priority order for node types to prefer freely moveable nodes, therefore, this approach is also evaluated in the analysis in Chapter 5.

3.8.2 Complexity

In each iteration, at least one node is removed from every SCC since it is transformed into a source or sink for this SCC. Therefore, the worst case is a fully connected graph and requires N iterations to remove all cycles. This results in a WCET of $\mathcal{O}((N+E)\cdot N)$.

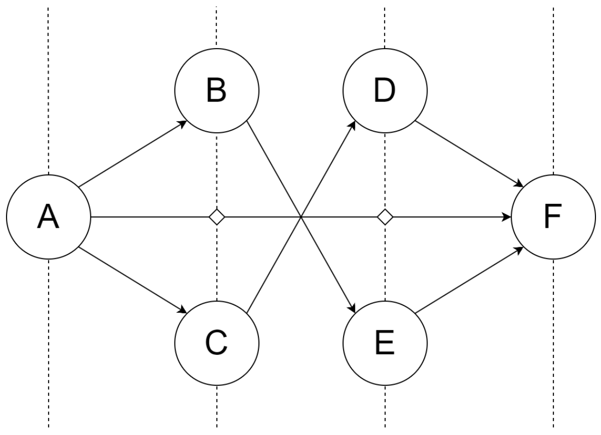
Chapter 4

Crossing Minimization

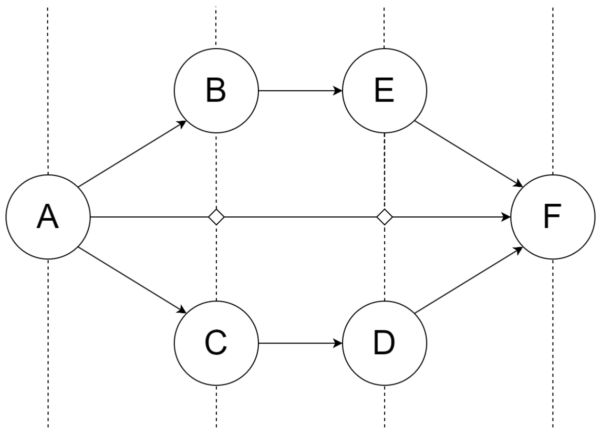
As previously mentioned, edge crossings are bad for the readability of a layout, as following edges is a vital part of working with graphs. This chapter presents the basics of crossing minimization by introducing the layer sweep crossing minimization algorithm, to explain where group model order can be used to alter this approach. Additionally, the challenges and opportunities for modeling languages with node-type grouping are explored.

Crossing minimization works by altering the node order within a layer (and the port order of a node), for which a *proper layered graph* is needed. A proper layered graph is ensured by the previous phase, layer assignment, by introducing *dummy nodes* and *dummy edges* to replace edges that stretch over multiple layers. Generally, when referring to the target of an edge the *long edge target* is meant. The long edge target is the original target of the edge if it was split into multiple edges and dummy nodes. Figure 4.1 shows an example of a proper layered graph and crossing minimizing this graph by altering the order in the third layer (swapping D and E). The multi-layer edge from A to F is replaced by dummy nodes (represented by diamonds) and edges.

While the in-layer ordering is altered by this strategy, using model order as secondary notation to dictate the in-layer order improves stability and increases control. The newly proposed approaches, shown in Section 4.4, try to improve the stability and control by introducing group model order. The approach in Section 4.5 evades the problem of node-type grouping by fixing the port order.



(a) Simple graph before crossing minimization.



(b) Simple graph after crossing minimization.

Figure 4.1. Illustration of crossing minimization.

```
input: proper layered graph G = (V,E,L)
output: proper layered and ordered graph G = (V‘,E,L‘)
r = randomSeed // Fixed random seed
t = thoroughness
sweepForward = randomSweepDirection(r)
bestOrder = null
for i = 0 ; i < t ; i++:
   G = randomizeLayers(G,r,sweepForward)
   do:
      for each l in L:
         minimizeCrossings(l)
   while improved(G)
   if crossings(G) < crossings(bestOrder):
      bestOrder = G
   sweepForward = !sweepForward
```

Listing 4.1. Pseudo code showing the layer sweep crossing minimizer.

4.1 Layer Sweep Crossing Minimizer

Listing 4.1 shows pseudo-code of the *Layer Sweep Crossing Minimizer*. Crossing minimization is NP-complete, even for bipartite graphs [GJ83]. Since the graphs considered in this thesis utilize ports, a heuristic is required that includes ports [SSH14]. This approach is prone to local minima and is, therefore, run multiple times, defined by the thoroughness `t` of the algorithm. The seed for random decisions is fixed to ensure the same behavior for a given graph. The initial *sweep direction*, the direction of traversing the layers, is chosen randomly in `randomSweepDirection`, using the fixed random seed. This sweep direction can either be forward or backward (along the layout direction or against the layout direction). The node order within each layer and the port order of each node are randomized in the `randomizeLayers` function, if their order is not otherwise constraint (i.e., the order of reactions). The edge order is extracted from the port order, to match the edge and port order.

The function `minimizeCrossings` uses two layers: the current layer, or the *fixed layer*, and the next layer, the *free layer*. In Figure 4.1 the second layer (containing `B` and `C`) is the fixed layer, and the third layer (containing `D` and `E`) is the free layer. The next layer is either the immediate layer to the right or left based on the direction of the layer sweep. The nodes and ports in the free layer are ordered to minimize crossings, utilizing a minimization strategy like the barycenter heuristic [SFH+10]. Simplified the barycenter heuristic works as follows:

- ▻ 1: For each node in the free layer, calculate the average position, the barycenter, of the connected nodes in the fixed layer.
- ▻ 2: Order the free layer based on the barycenter values.

Finally, edge crossings are counted using the node and port order described in [BMJ04] and the best order for the graph, with respect to edge crossings, is remembered and used after `t` runs. The following section explains how the barycenter approach can be modified to include model order.

4.2 Model Order In Crossing Minimization

Recent endeavors in utilizing the model order for the layered algorithm have tackled the crossing minimization phase. "Preserving Order during Crossing Minimization in Sugiyama Layouts" [DH21] explores this in-depth, with further exploration in "Model Order in Sugiyama Layouts" [DRv23]. The proposed approach defines a vertex order (node order) and a port order based on the initial order of the graph. While the initial graph may define orders for nodes and edges, which should be reflected in the final layout, sometimes it is impossible to express both orders since they may contradict each other. Therefore, a flag is introduced to decide if the node or the port order is prioritized. The following modifications to the layer sweep crossing minimizer are introduced:

- Before beginning the layer sweep this approach order the ports and layers based on these orders. Let G_I be the graph with this order (Introduced prior to the for loop in line 7).
- The *bestOrder* is defined with these initial orders applied to the graph, G_I.
- The initial randomization is skipped and the layer sweep is run on G_I.
- Continue with the original random approach for subsequent runs to improve the layout if the initial ordering fails.

Another problem with using the model order crossing minimization is the introduction of dummy nodes and edges in layer assignment. As these elements are not part of the textual model these elements do not have an initial order. Dummy edges use the model order of the initial edge they replace. For dummy nodes the model order of the *long edge target node* is used. Real nodes can be compared by model order, but comparing a real node directly with a dummy node by model order cannot be done.Comparing a real node and a dummy node is done by selecting the nodes with the lowest model order in the previous layer that connect to the real and dummy node.

4.3 Enforcing Node Order

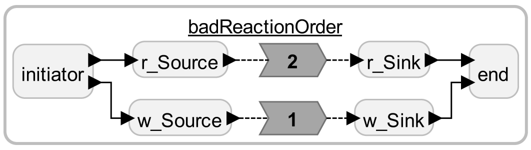
For some nodes, the declaration order is part of the syntax. As mentioned, this syntax sensitive order is the case for reactions in LF. The in-layer ordering of these nodes with syntax sensitive declaration order uses the model order instead of the barycenter heuristic. This use of model order forces the in-layer order of reactions to be identical to the execution order, eliminating the behavior shown in Figure 4.2a. The order shown in Figure 4.2a violates model order for reactions and does not reflect the execution order, which can be used as secondary notation. This behavior as this should be avoided.

Currently reactions and reactors use the model order in LF. With the introduction of group model order this can be changed as shown in Section 4.4. Figure 4.2b shows that using the model order for both reactors and reactions, while increasing control, may introduce edge crossings. This, however, indicates a crossing declaration order. One could argue that it is the responsibility of the developer to eliminate these edge crossings by avoiding crossing declarations shown in Figure 4.2c.

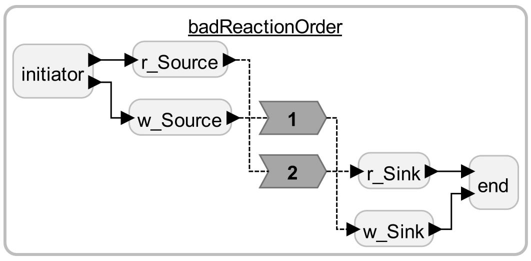
Using the model order for in-layer ordering of nodes with different types creates a separation between different node types. I.e., let type A be the node type that is (generally) defined ahead of type B; the in-layer order reflects this declaration behavior and orders nodes of type A to the top and nodes of type B to the bottom.

Enforcing the node order for all nodes reduces the effectiveness of crossing minimization, as the only changes that are applied in the crossing minimization phase would then affect dummy nodes.

Generally, one has to evaluate the trade-off for control and ease of use, which may change depending on the expertise of the developer.



(a) Undesired ordering of reactions, not following model order and execution order.



(b) Bad declaration order forces edge crossing when using model order.

```
main reactor {
  i = new initiator();
  r1 = new r_Source();
  r2 = new r_Sink();
  w1 = new w_Source();
  w2 = new w_Sink();
  e = new end();

  reaction (w1.out) -> w2.in
  reaction (r1.out) -> r2.in

  i.r -> r1.in
  i.w -> w1.in
  w2.out -> e.w
  r2.out -> e.r
}
```

(c) Bad declaration order, with crossing orders. Declaring $r1, r2$ before $w1, w2$ but declaring the reaction of $w1 \rightarrow w2$ before $r1 \rightarrow r2$

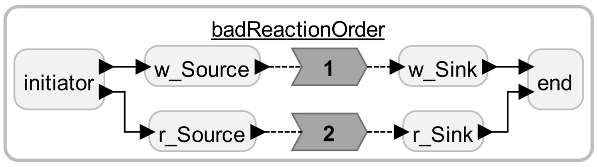
Figure 4.2. Reason with for fixing node order based on model order and the problems this brings.

4.4 Introducing New Behavior For Node Types

Using the model order within the same node type is sensible for in-layer node ordering. However, the same problem that exists for cycle breaking, that comparing the model order of different types, emerges when comparing different node types in the same layer. The following sections explore options for this problem by introducing group model order.

4.4.1 Model Order For One Type

The first option is restricting the use of model order to only one node type. This is shown in Figure 4.3b in a simplified version. If only one node type utilizes model order, the remaining types are handled by using the barycenter method. This approach reduces the use of secondary notation, as the model order for all other types is not used. While this can improve the layout it defeats the goal of this thesis, increasing . However, it reduces the workload for developers (by being less restrictive in the declaration order) while maintaining the model order for the node type of highest preference and reducing crossings for the different types, as seen in Figure 4.3a. As this distinction between different node types can violate the transitive constraint of orderings an order preserving sorting algorithm like insertion sort has to be used [DRv23].



(a) Enforcing node order for reactions and crossing minimizing reactors.

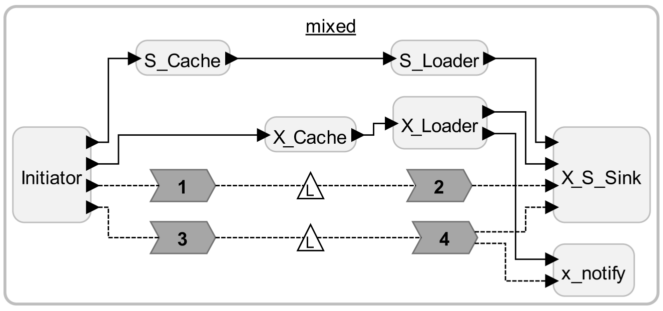
```
1 Node n1,n2
2 if (n1.hasModelOrder
3       AND n2.hasModelOrder
4       AND n1.type == Reaction
5       AND n2.type == Reaction):
6    // Use ModelOrder
7 else :
8    // Use Barycenterheuristic
```

(b) Adding a filter in the `ModelOrderBarycenterHeuristic.java` to check the type of the nodes (lines 4 and 5).

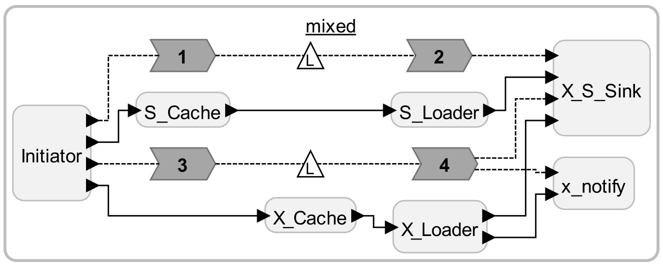
Figure 4.3. Enforcing node order for one specific node type.

4.4.2 Barycenter For Different Types

The second approach utilizes the newly introduced definition of group model order to ensure the model order is only used within the same type and uses the barycenter approach for the comparison of different node types regarding the in-layer ordering. This can be done by altering lines 4 and 5 of Figure 4.3b to check if `n1` and `n2` are of the same type. This retains the control for the developer while improving edge crossings between different types the developer can not adjust. The differences shown in Figure 4.4a and Figure 4.4b shows the improvements this approach gives over simply using the restricted model order. This approach, however, is again susceptible to crossing declaration order and edge crossings based on this crossing declaration order.



(a) Using the model order to order layers. Creating a separation between reactions and reactors.



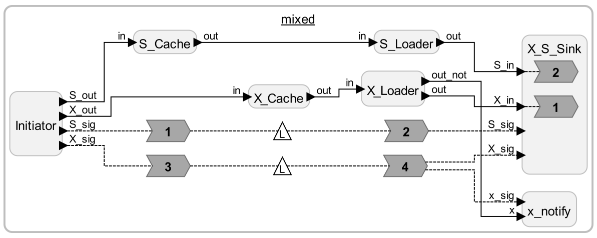
(b) Using the barycenter method to compare different node types. This mixes reactors and reactions and reduces crossings.

Figure 4.4. Using model order between different node-types or only for nodes of the same node-type.

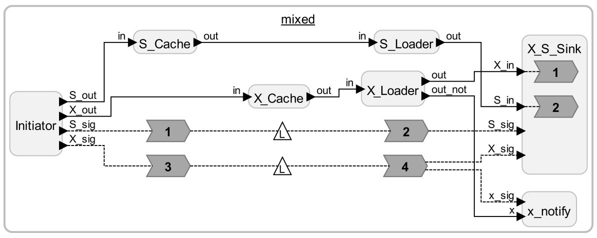
4.5 Enforcing Port Order

This section explores the approach of fixing the port order. This approach works for languages that explicitly define ports like LF. For languages like SCCharts the ports are created implicitly and use the edge order.

As port declarations are freely movable in LF, this could be used instead of the model order of nodes. This approach can be combined with the approach of still enforcing the model order for a specific node type like reactions. This may introduce edge crossings, which could be resolved by altering the port order. Figure 4.5 shows the complications of enforcing the port order for models that were not created with this option, as presented in Figure 4.5c. Figure 4.5a shows that this port order enforcement can create in-layer order violations for the execution order of reactions and the creation of unnecessary crossings. Figure 4.5b shows these problems can easily be eliminated by reordering the ports.



(a) Enforcing the port order creates an unnecessary edge crossing for the outgoing edges of X_Loader and forces the reactions order in X_S_Sink to go against execution order.



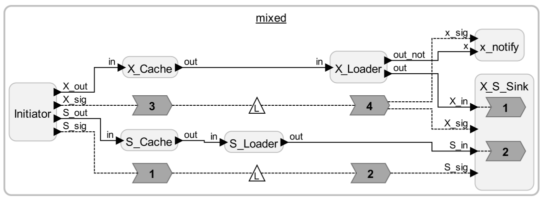
(b) Fixing the port order eliminates the unnecessary edge crossing and fixed the reaction order

```
reactor X_S_Sink{
   input S_in:int
   input X_in:int
   input S_sig:signal
   input X_sig:signal

   reaction (X_in){==}
   reaction (S_in){==}
}

reactor X_Loader{
   input in:int
   output out_not:int
   output out:int
}
```

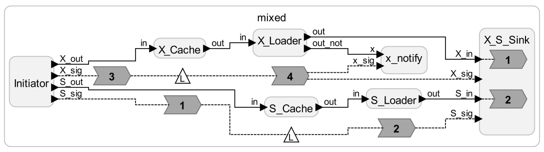
(c) The declaration of the the ports for the nodes X_Loader and X_S_Sink in Figure 4.5a

Figure 4.5. Reducing edge crossings and fixing model order violations for reactions by modifying the fixed port order.

Furthermore, removing the in-layer order constraint of reactions may reduce edge crossings. This would give full control to the developer. The order may be adjusted to reduce edge crossings to an optimal solution with the applied layering. However, this allows in-layer model order violations for reactions, as shown in Figure 4.6a, where the in-layer order of all reactions contained in mixed is violated, but the edge crossings in this layout are reduced to the minimal amount of two with this layering. While the optimal solution for this graph is 0 edge crossings, by moving the node x_notify to a layer ahead of the layer with X_S_Sink, which can be achieved with the *Stretch Width Layerer* [RAC+17] and is visualized in Figure 4.6b.



(a) Violation of the in-layer order for reactions to reduce crossings



(b) Violation of the in-layer order for reactions to reduce crossings

Figure 4.6. Using a fixed port order without node order constraints to reduce crossings.

Something to consider is that the port order has to match the internal and external graph for hierarchical nodes. The different port orders of the node `X_S_Sink` in Figure 4.5b and Figure 4.5a. While the port order used in Figure 4.5a reduces edge crossings in the external graph the model order of the reactions contained in `X_S_Sink` is violated. The port order used in Figure 4.5b fixes the model order violation for the reactions contained in `X_S_Sink` but creates an edge crossing in the external graph. Keeping track of both portion of the entire graph may be tedious, especially if the reactor is defined in a different file.

Enforcing the node order gives the developer a high degree of control; however, this comes with some overhead. Ultimately, enforcing the port order creates a massive benefit for interactively working with the diagram: stability, as introduced in Section 2.9.

Chapter 5

Analysis

This chapter gives an analysis of the different approaches for cycle breaking and crossing minimization. Additionally, some feedback from the LF development team and a team of developers at Magnition is evaluated.

5.1 Evaluation Basics

This chapter utilizes the GrAna tool from KIELER. This tool allows easy evaluation of graphs in the *elkg*[1] format. This tool exports the result of the analysis in a *Comma-Separated Value* (CSV) file. The *R-Project*[2] was used for the evaluation of those and the creation of the diagrams used in this chapter. In this section, general statistics that do not change for the different approaches, like the average node count, are given.

5.1.1 The Dataset(s)

Four different datasets are analyzed in this chapter:

Table 5.1. The datasets used in the analysis.

Dataset Name	Source	Description	Size
LF Unit Tests (LFT)	Lingua Franca Github [3]	The models of this dataset are used in the unit tests of LF, to ensure intended behavior.	139
LF Experiments (LFEP)	Lingua Franca Playground Github[4]	This dataset contains models that are actively developed internally to demonstrate proposed language features.	70
LF Examples (LFEA)	Lingua Franca Playground Github[5]	These models are examples for beginners to learn LF.	131
Magnition Colaboration (COLAB)	Magnition Github[6]	This dataset contains real-world models, shared by Magnition.	38

This brings the total dataset size to 378 models. However, as shown in Section 5.1.5 and Section 5.2.1, most of these graphs are very small and, the same layout is created. In addition to comparing the different approaches as introduced in Chapter 3 and Chapter 4, the evaluation will compare the datasets with each other to see if fundamental differences exist when it comes to different sources.

[1]https://eclipse.dev/elk/documentation/tooldevelopers/graphdatastructure.html
[2]https://www.r-project.org/
[3]https://github.com/lf-lang/lingua-franca/tree/master/test
[4]https://github.com/lf-lang/playground-lingua-franca/tree/main/experiments
[5]https://github.com/lf-lang/playground-lingua-franca/tree/main/examples
[6]https://github.com/MagnitionIO/LF_Collaboration/tree/main

5.1.2 Data Normalization

The significant differences in graph size are beneficial for gathering data of a broader range; however, the context and, with that, the comparability of the statistic is lost. Comparing raw edge crossings or backward edges in a violin plot shows the average distribution for the dataset; it does not, however, show if the models with the larger numbers are the same models for all strategies. Additionally, for statistics, like edge crossings, the analysis data consists of very small results for the majority of the models and some large outliers. Showing both the small and large examples in one readable diagram is not possible without spanning the diagram over an entire page. Therefore, *normalization* is sensible. The proposed normalization approach reduces the size of the y-axis and increases comparability, for this, normalization is defined as follows.

5.1 Definition (Normalization)**.** Let $\mathbb{A}_i^x$ be a ordered set of result data for strategy i and analysis x. $\mathbb{A}^x$ contains all result sets $\forall i\, 1 \leqslant i \leqslant |\mathbb{A}^x| : \mathbb{A}_i^x \in \mathbb{A}^x$. Normalization for a data set $\mathbb{A}^x$ is defined as:
$\rhd$ $\texttt{norm}(\mathbb{A}^x) = \forall \mathbb{A}_i^x \in \mathbb{A}^x : \texttt{norm}(\mathbb{A}_i^x)$
Let $v_j^{i,x}$ be the resulting value of analysis x and strategy i for model j. Therefore, $\texttt{norm}(\mathbb{A}_i^x)$ is given as:

$$\rhd\ \texttt{norm}(\mathbb{A}_i^x) = \forall j\, 1 \leqslant j \leqslant |\mathbb{A}_i^x| : \begin{cases} v_j^{i,x} / \sum_{k=1}^{|\mathbb{A}^x|} \frac{v_j^{k,x}}{|\mathbb{A}^x|} & \text{, if} \sum_{k=1}^{|\mathbb{A}^x|} \frac{v_j^{k,x}}{|\mathbb{A}^x|} > 0 \\ 1 & \text{, otherwise} \end{cases}$$

Or in words, divide the value of a specific analysis by the mean value of all strategies for this model and analysis.

Datasets normalized with this method now express how a strategy performs in relation to the mean of all strategies. If the normalized value is greater than 1, the strategy performs worse than the average (i.e., creating 1.4 times the number of backwards edges as the mean of all strategies). If the normalized value is 1 is equivalent with performing just like the average. This explains the *otherwise* case of the formula; if all strategies evaluate to 0 the average would be 0 but this would indicate that all strategies performed better than the average. As they performed exactly as the average, they are normalized to 1. If the normalized value is smaller than 1 the strategy performs better than the average (i.e., creating 0.6 times the backwards edges as the mean). For the example of backward edges, if a normalized value is 0 it indicates that for this model the strategy was able to create a layout without introducing backwards edges, while other strategies required backwards edges.

5.1.3 Kruskal-Wallis Test

The results of the analysis in this chapter are evaluated with the Kruskal-Wallis test [KW52] to check for statistical significance. It is a non-parametric[7] test for samples of equal or different sample sizes and tests if the samples originate from distributions around the same median value. The Kruskal-Wallis test is an extension of the Mann-Whitney-U-Test [MW47], which is only capable of comparing two samples. As the p-value to indicate significance the standard value of 0.05 is used. The null hypothesis is: The samples originate from distributions with the same median. If the p-value is smaller than 0.05 this hypothesis is discarded. The implementation of this test is given by the `R` package `rstatix`[8].

5.1.4 Willcoxon Test

As a post hoc analysis, if the Kruskal-Wallis test shows significance, the pairwise Willcoxon signed rank test [Wil45] is used. This test analyses paired data to indicate where the differences are in a

[7] https://en.wikipedia.org/wiki/Nonparametric_statistics

[8] https://cran.r-project.org/web/packages/rstatix/index.html

multi-sample analysis. To counteract the multiple comparisons problem[9] for statistical tests that consist of multiple comparisons, the Bonferroni correction [Bon36] was used. The Willcoxon test indicates significance if the *p*-value is smaller than the *p*-value of the Kruskal-Wallis test divided by the number of samples. The implementation of this test is given by the `rstatix` package as well.

5.1.5 Node / Edge Count Analysis

Figure 5.1 shows the distribution of the node count for all datasets, not including dummy nodes. The ggbreak [XCF+21] package was used to create a break in the y scale. It is apparent that the majority of the examples have a relatively small node count. With some out-lier models with up to 178 nodes.

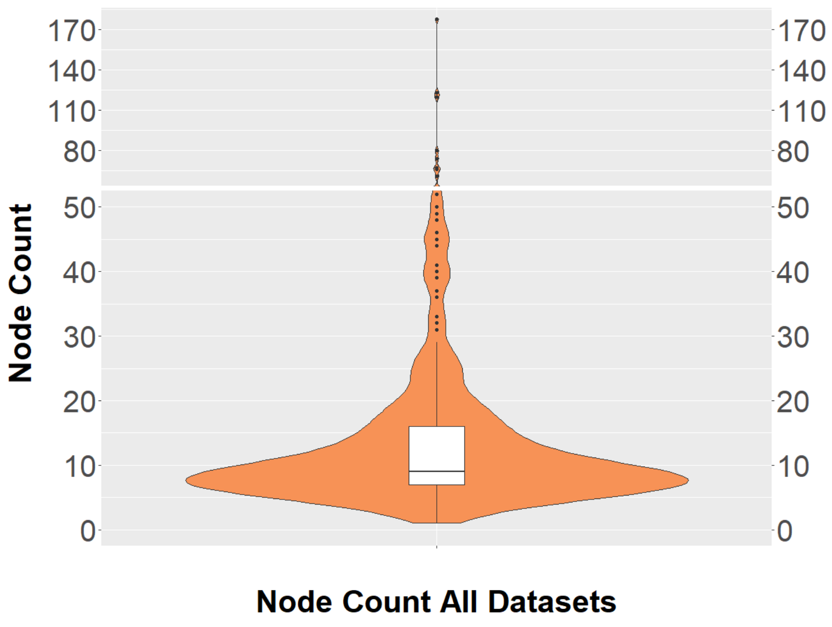
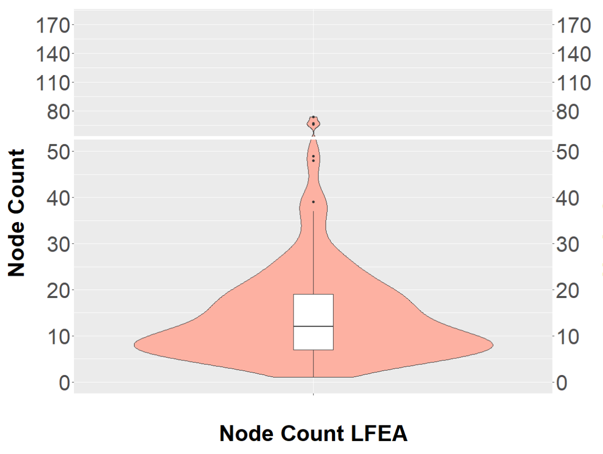


Figure 5.1. Node count distribution and statistics for the entire dataset.

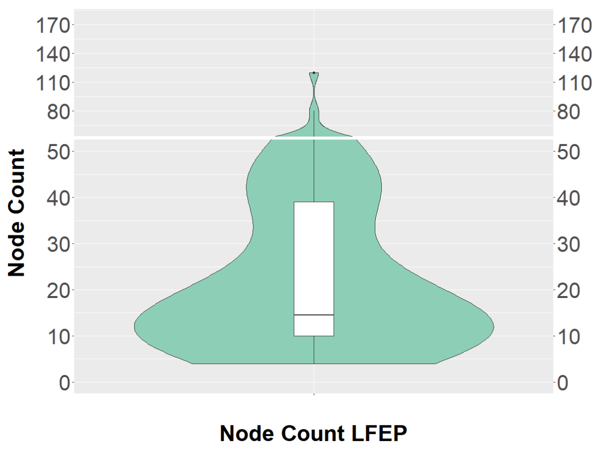
Predominantly, the `LFT` dataset consists of extremely small examples, as seen in Figure 5.2c. The `COLAB` and `LFEP` datasets include the largest samples, as presented in Figure 5.2b and Figure 5.2d. These samples consist of a composition of many small models. The large samples are particularly interesting, as these show the highest amount of differences for the tested approaches. Furthermore, testing the differences in layout creation times should be more conclusive for larger diagrams.

With increasing node count, the edge count should increase. The plots for the edge count analysis of the different datasets are shown in the appendix, in Figure A.1. These plots, together with the plots for the edge count, show different behaviors when working with compositions in Lingua Franca. Figure A.1b shows a relatively equal distribution for the edge count. This indicates that the dataset has a wide range of models, regarding the size. On the contrary, as seen in Figure A.1d, the `COLAB` dataset consisting of many small models and some very large models. This evaluation can be explained by taking a closer look at the composition of a given model. A large model often consists of the composition of many small models.

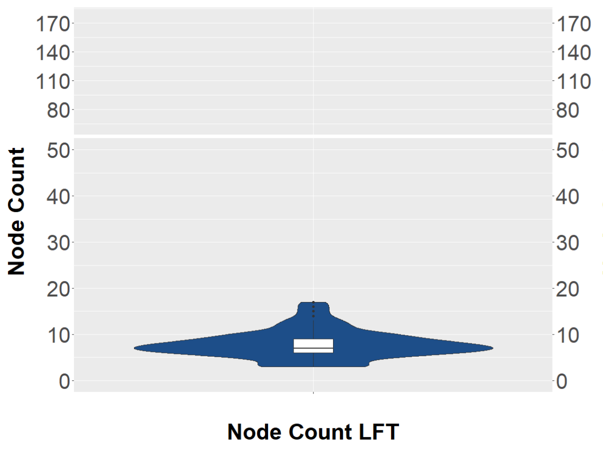
[9]https://en.wikipedia.org/wiki/Multiple_comparisons_problem



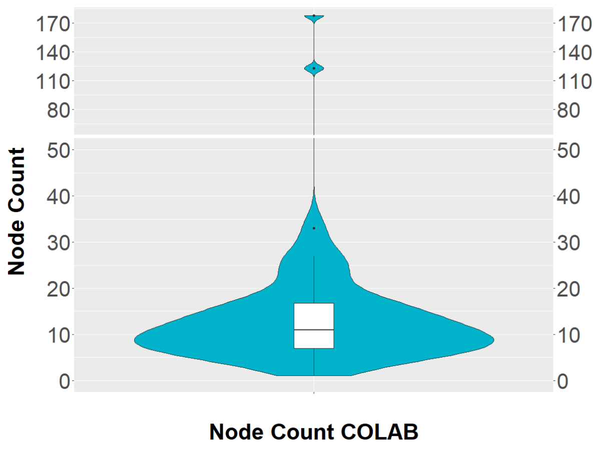
(a) LFEA Node Count



(b) LFEP Node Count



(c) LFT Node Count



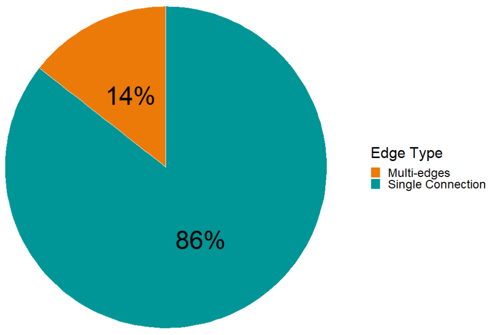
(d) COLAB Node Count

Figure 5.2. Comparison of the node count in the different datasets.

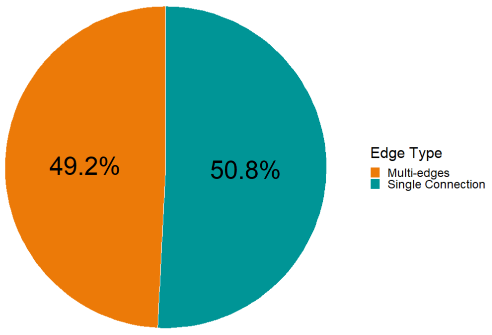
As explained in Chapter 2, due to the fact that LF models may have multi edge connections the edge count is no longer bound by the squared node count. However, in all of the models, the number of edges never exceeds the number of nodes squared.

5.1.6 Multi-Edge Connections

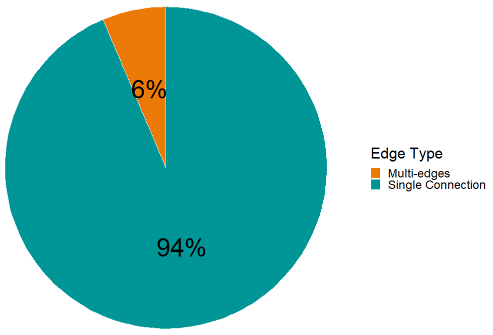
As mentioned in Section 3.3.4, there is a way to reduce number of evaluation for multi-edge connections to only a single evaluation. Figure 5.3 shows the average percentage of multi-edge connections per model in each dataset. The data shows that an average of 9.67% of the edges in the models of the LFEA, LFT and COLAB datasets are multi-edges. The LFEP dataset has an abnormally high percentage of average multi-edge connections, with 49.2%. For five models of the entire dataset all edges are multi-edge connections. This shows that the proposed approach may drastically reduce the evaluation for edges. The average model in the entire dataset has 27% multi-edges.



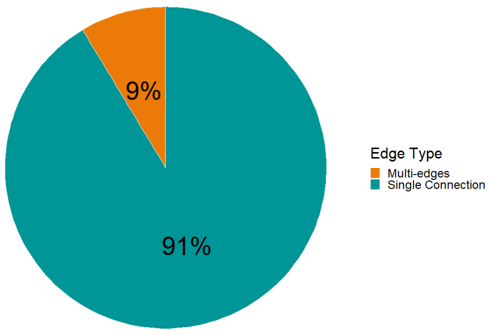
(a) `LFEA` multi-edge connection percentage.



(b) `LFEP` multi-edge connection percentage.



(c) `LFT` multi-edge connection percentage.



(d) `COLAB` multi-edge connection percentage.

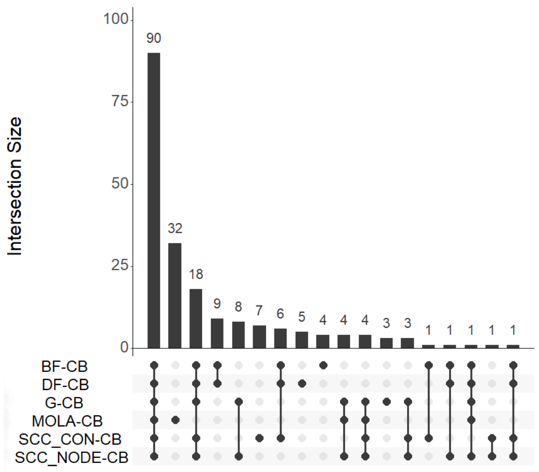
Figure 5.3. Comparison of the multi-edge connection percentage in the different datasets.

5.2 Cycle Breaking Evaluation

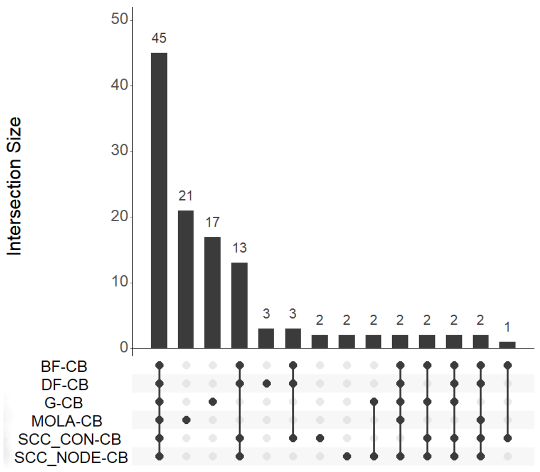
This section evaluates six different cycle breaking strategies, namely: BF-CB, DF-CB, G-CB, MOLA-CB, *Strongly Connected Components Connectivity Cycle Breaker* (SCC_CON-CB) and *Strongly Connected Components Node Type Cycle Breaker* (SCC_NODE-CB). The BF-CB and DF-CB strategies use the node order option. The results for the MOLA-CB strategy uses the skip sequential edges with fallback edges option, as the evaluation for other options performed worse in all categories. The SCC_CON-CB strategy is the `SCC-CB` with the connectivity model order option, so that they rely on an explicitly defined model order. Finally, the SCC_NODE-CB is the SCC-CB with the behavior by node-type option, preferring to reverse the incoming edges of the reactor with the lowest model order. Five different analyses are done for these strategies.

5.2.1 Matching Layouts

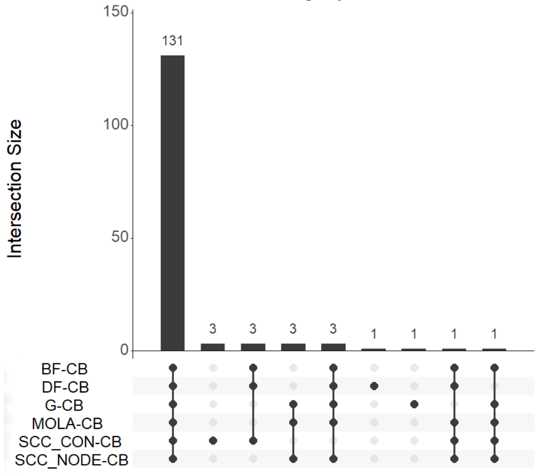
For inherently acyclic models, the layouts are indifferent. Furthermore, small models not only have a high probability of not containing any cycles but also have small cycles with few options for choosing different edges to reverse. In this section, the layouts are evaluated for equality. Figure 5.4 shows the



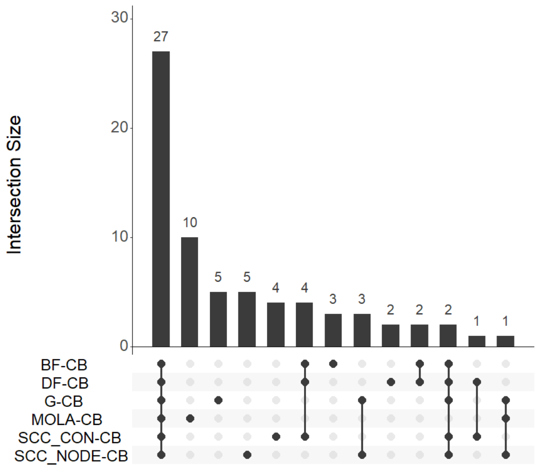
(a) `LFEA` matching layouts.



(b) `LFEP` matching layouts.



(c) `LFT` matching layouts.



(d) `COLAB` matching layouts.

Figure 5.4. Comparing how often different cycle breaking strategies create the same layout for all datasets.

results of this analysis using an UpSet[10] diagram. UpSet graphs show the intersections of sets as a matrix. Each column in the matrix below the bar chart describes an intersection, where the filled-in dot highlights the sets that are part of this intersection. The bar above a given set indicates the size of this intersection. Usually, UpSet diagrams include an indicator for the size of the individual sets, as all sets have the same size in this example, this indicator is omitted. Figure 5.4 shows that, for all datasets, the majority of the layouts are indifferent. In total, of the 378 models in all datasets, 294 models have the same layout for all options, equating to $77.\overline{7}\%$ of the models. For the `LFT` dataset of the 139 models 131 (or $94,2\%$) are indifferent to altering the cycle breaking strategy. Of the 378 models, only 3 have unique layouts for the six different approaches.

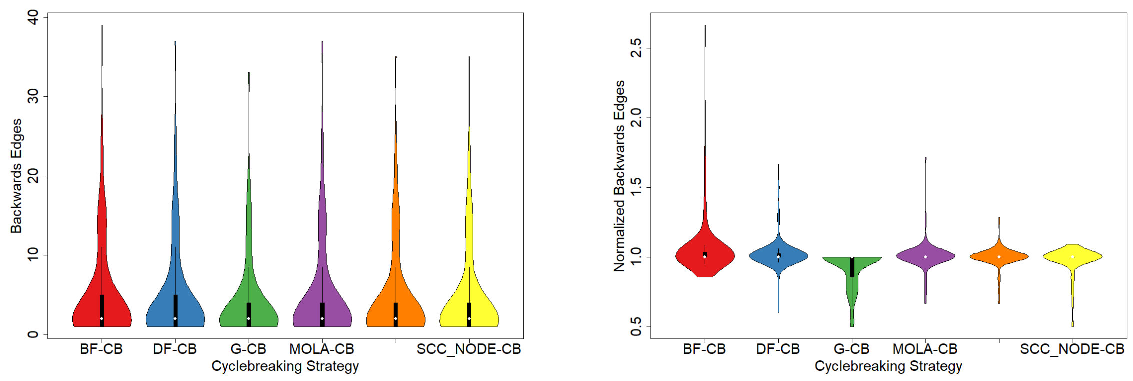
Any statistic for the entire dataset would be vanishingly small, as they would be overshadowed

[10] `https://upset.app`

by the large amount of indifferent layouts. Additionally, all layouts where no edges are reversed can be excluded, as any differences in these layouts originate in *stability-inaccuracies* in other phases of the layered algorithm. Stability-inaccuracies mean that even though the layered algorithm is designed to replicate the same layouts if the model and the settings do not change this behavior can not be guaranteed entirely, i.e., detached parts are sometimes ordered differently. Therefore, the analysis is done on the datasets where the entries with equal layouts or without edge reversals for all strategies are removed. This reduces the dataset size from 378 models to 71 models.

5.2.2 Backward Edge Analysis

As mentioned, the feedback set problem is the underlying problem for cycle breaking. Therefore, a smaller solution for cycle breaking is a smaller solution for the feedback set problem, creating a metric to evaluate the heuristics behind each approach. Figure Figure 5.5a shows the raw results for the backward edge analysis for the entire dataset, and Figure 5.5b shows the normalized graph. These graphics show why normalization is helpful for displaying differences. While the results for the different strategies in Figure 5.5a look fairly similar, Figure 5.5b shows that the G-CB performs extremely well. For all examples, it performs at least as well as the average of the different strategies but, in most cases, creates fewer edge reversals. In fact, for only one model of the 71 models is this strategy outperformed by the DF-CB. The BF-CB performs the worst of them, which is expected considering how the breadth-first search discovers nodes.



(a) ALL backwards edge count.

(b) ALL normalized backwards edge comparison.

Figure 5.5. Analysis data for the backwards edge analysis

The Kruskal-Wallis test returns a p-value of 0.9, failing to show significant differences for the backwards edge count. The mean value for the G-CB is 4.92 while the mean values for the other are between 5.75 and 6.38.

5.2.3 Edge Crossing analysis

While edge crossings are further evaluated in Section 5.3, crossings are also induced by differences in cycle breaking, as the nodes can be sorted into different layers. The settings for crossing minimization are kept consistent for this analysis. Here, the crossing minimization uses the model order even between different node types with a fixed port order to reduce the impact of the crossing-minimization step.

A similar analysis for SCCharts has shown that graphs with fewer layers tend to have more edge crossings, as edges cluttered between fewer layers [Rie22]. This was particularly critical for the BF-CB for SCCharts. Figure 5.6 shows the normalized results of the edge-crossing analysis.

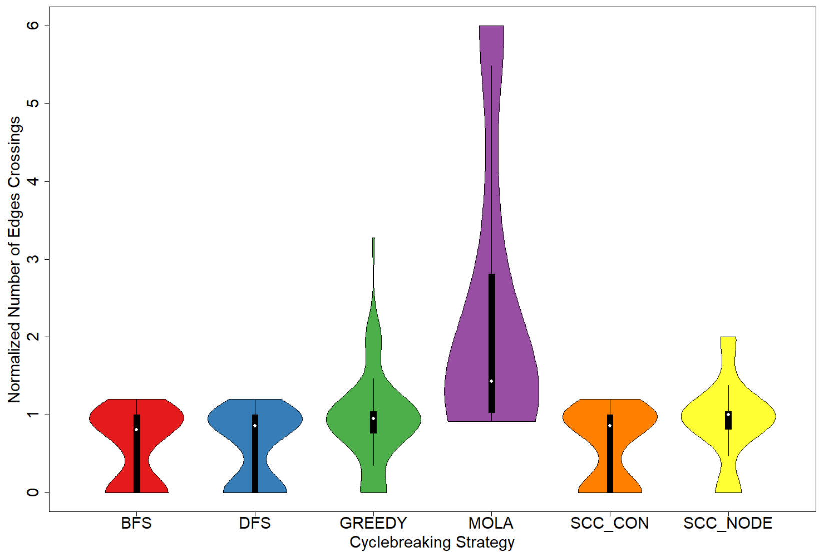


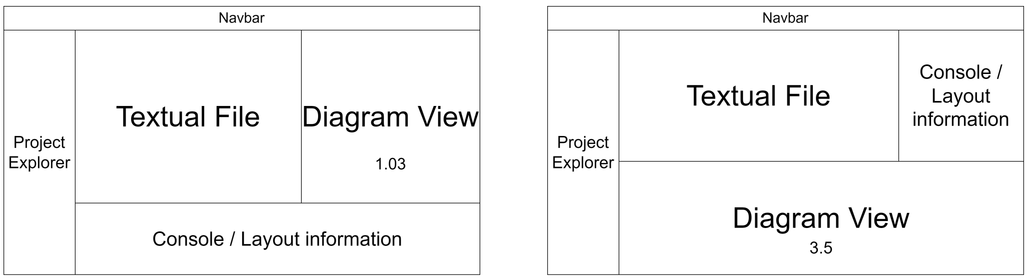
Figure 5.6. Normalized data for edge crossings.

The visualization of the analysis data for the BF-CB, DF-CB and the SCC_CON-CB are extremely similar. Taking the matching layouts presented in Figure 5.4 into consideration, this should not be surprising. Of the 378 models, these three strategies create identical layouts in 367, or 97.1%, of the models. The average edge crossing number is 14 ± 1 for the entire dataset and the different strategies: Therefore, the Kruskal-Wallis test fails to indicate significant differences with a p-value of 0.292. The MOLA-CB performs worse in most cases, with some examples that are especially bad.

5.2.4 Aspect ratio Analysis

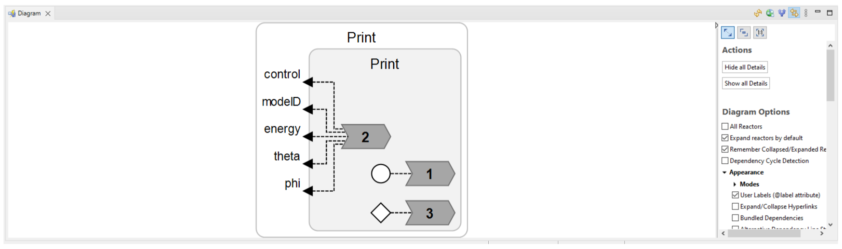
5.2 Definition (Aspect ratio)**.** The *aspect ratio* of a layout is defined by w_{width}/w_{height} where w is the root node of the graph.

The aspect ratio of a graph is a relevant metric for readability. Figure 5.7 shows how differences in the aspect ratio of the drawing area and the layout of the graph impact the readability of a model. A mismatch results in layouts with decreased readability, as the layout can not utilize the drawing area, creating unused whitespace around the layout, as seen in Figure 5.7a.



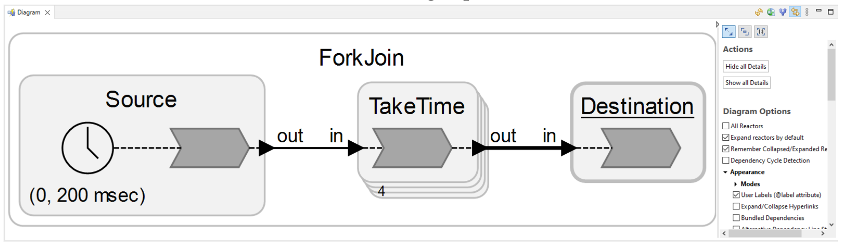
(a) Diagram view next to the textual file. **(b)** Diagram view below the textual file.

Figure 5.8. Different IDE layouts and their estimated aspect ratio for the diagram view.



Actions
Hide all Details
Show all Details
Diagram Options
☐ All Reactors
☑ Expand reactors by default
☑ Remember Collapsed/Expanded Re
☐ Dependency Cycle Detection
Appearance
Modes
☑ User Labels (@label attribute)
☐ Expand/Collapse Hyperlinks
☐ Bundled Dependencies

(a) Mismatching aspect ratio.



Actions
Hide all Details
Show all Details
Diagram Options
☐ All Reactors
☑ Expand reactors by default
☑ Remember Collapsed/Expanded Re
☐ Dependency Cycle Detection
Appearance
Modes
☑ User Labels (@label attribute)
☐ Expand/Collapse Hyperlinks
☐ Bundled Dependencies

(b) Matching aspect ratios.

Figure 5.7. Relation of the drawing area aspect ratio and the graphs aspect ratio.

Taylor and Rodgers [TR05] name multiple pleasing aspect ratios like 1 : 1, the golden ratio 1 : 1.618 or 1 : 2. They describe that the layout's overall aspect ratio should match the drawing area (e.g., a page, screen, or the containing hierarchical node). As mentioned, LF is mostly used in the Epoch IDE or in VS Code with the LF-Plugin. These IDEs have a similar layout structure. Figure 5.8 shows the two main configurations for working with LF files and an estimated aspect ratio of the diagram view on a 16 : 9 screen. These layouts are altered mostly to adhere to the preferences of the developer. All six strategies and every model in the entire dataset result in 2.268 layouts. Only 234, or 10.3%, of these layouts have an aspect ratio below one, which would mean that the height is greater than the width.Overall, the mean aspect ratio for all these layouts is 2.49, and the median is 2.26. This indicates that the number of layers is usually greater than the number of nodes in any layer. Additionally, the visualization of reactions have an aspect ratio of 2.05, which inherently increases the layout width. All this indicates that the IDE structure shown in Figure 5.8b is a better fit for LF.

The results for the aspect ratio analysis are fairly consistent across all datasets, Figure 5.9 shows the results for the dataset described in Section 5.2.1 (removed indifferent layouts).

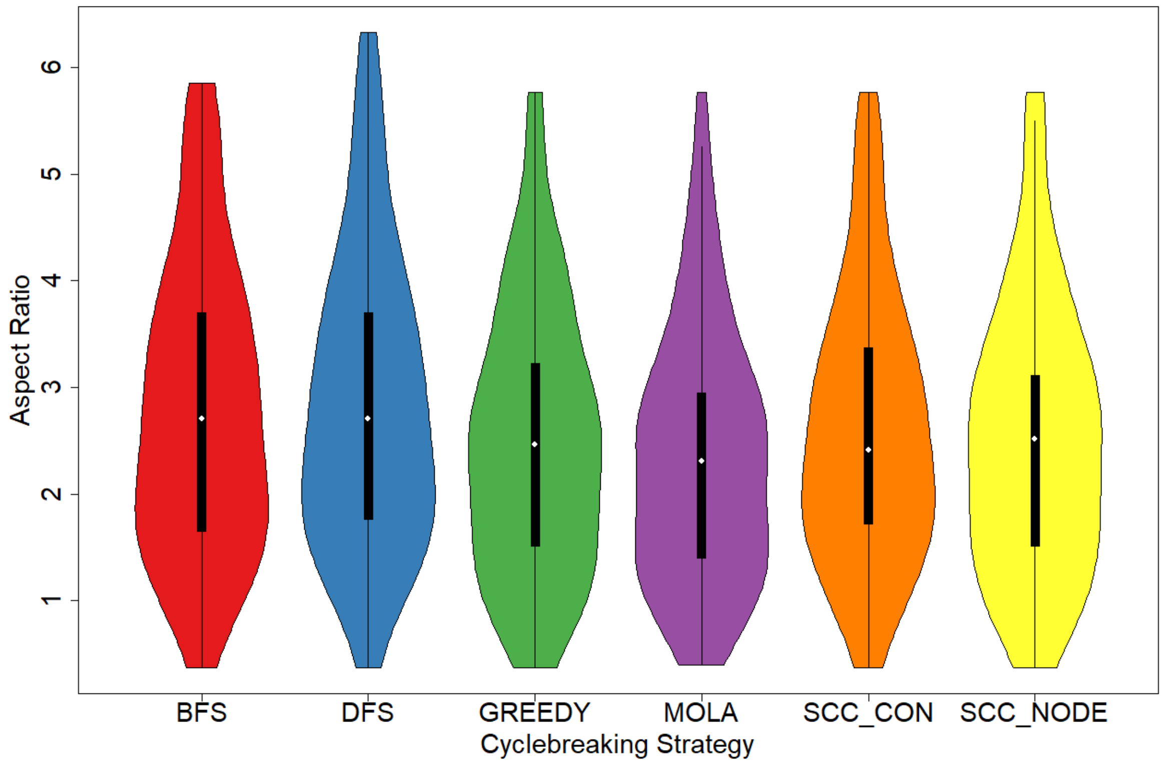


Figure 5.9. Results of the aspect ratio analysis.

For this reduced dataset, the mean values range from 2.36 for the MOLA-CB and 2.85 for the DF-CB. No pattern can be found for the relation between aspect ratio and node/edge count. The p-value for the Kruskal-Wallis test is 0.18, failing to indicate a significant difference.

5.2.5 Execution Time Analysis

In Chapter 3 the WCET for the different strategies is given in $\mathcal{O}$ notation; this section evaluates real execution times measured in nanoseconds. The entire dataset was used for this analysis, as the execution times are different even for identical layouts. The load on the system was decreased to a minimum, stopping all unnecessary processes. Resulting in a background load of 2% for the *Central Processing Unit* (CPU), 3% for the *Graphics Processing Unit* (GPU) and 40% of the *Random-Access Memory* (RAM). Table 5.2 shows the system specifications used for this test. Figure 5.10 shows the execution time for the different cycle-breaking strategies, with a maximum of 0.001s. Table 5.3 presents the raw data for key values of the analysis. The Kruskal-Wallis test has a p-value of $6.5e-15$, indicating significant differences in the datasets. The Wilcoxon test indicates differences between all datasets except between the SCC_CON-CB and SCC_NODE-CB.

Table 5.2. System specifications for the execution time analysis.

Component- / Softwarename	Specification
CPU	AMD Ryzen 7 5800X 8-Core 3.8GHz
GPU	NVIDIA GeFroce RTX 4070 Ti
RAM	32 GB at 2133 MHz
Operation System	Windows 10.0.19045
Java	Java 17.0.4.1 2022-08-18 LTS
Epoch	0.5.10-Snapshot

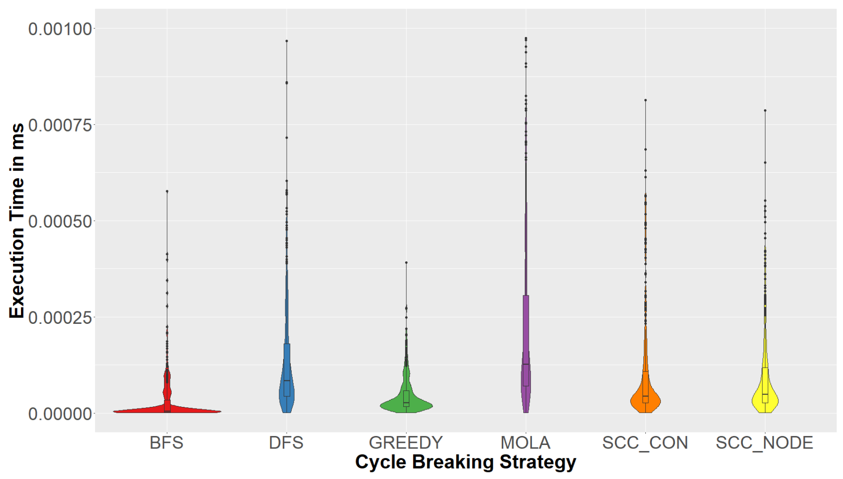


Figure 5.10. Visualization of execution times capped at 0.001s.

Table 5.3. Exact values for key statistics of the execution time values in seconds.

Statistic	`BF-CB`	`DF-CB`	`GREEDY-CB`	`MOLA-CB`	`SCC_CON-CB`	`SCC_Node-CB`
min	$8e-07$	$7e-07$	$1.4e-06$	$1e-06$	$1e-06$	$1e-06$
max	$3.7e-03$	$6.4e-03$	$3.9e-04$	$5.8e-03$	$7.0e-03$	$7.9e-4$
mean	$6.0e-05$	$1.6e-04$	$4.7e-05$	$3.1e-04$	$1.3e-4$	$9.6e-05$
median	$4.9e-06$	$8.4e-5$	$2.7e-05$	$1.4e-04$	$4.5e-05$	$4.8e-05$

An interesting observation is visible in the difference between the BF-CB and the DF-CB. While the BF-CB has the overhead of using Tarjan's algorithm as a preprocessor, the mean and median execution time is lower. Especially the mean execution time is lower, which can be a result of entirely skipping cycle breaking if no cycles are found in the preprocessing step. Additionally, despite running Tarjan's algorithm multiple times, the differences in execution times for SCC_CON-CB and SCC_NODE-CB not as significant as the WCET would suggest.

While the statistical tests show significant differences, the differences are mostly in the magnitude of fractions of milliseconds. Even though the hardware used for these tests is rather powerful for simple layout tasks, the differences found here should not be the reason to prefer any strategy.

5.3 Crossing Minimization Evaluation

In this section, three options for utilizing model order or group model order in crossing minimization are evaluated. As cycle breaking has an influence on layer assignment and, therefore, on crossings, the options are evaluated for different cycle-breaking strategies. To reduce the number of different combinations some cycle-breaking strategies will be grouped. As mentioned for the set of BF-CB, DF-CB and SCC_CON-CB, 97.1% of the layouts are identical. Therefore, these strategies are represented by the DF-CB. The G-CB and the SCC_NODE-CB create equal layouts in 353, or 93.4% of the models and are therefore grouped as well, represented by the G-CB.

The following options for crossing minimization are compared:

▻ The original approach by Domrös and von Hanxleden [DH21]. Referred as `DvH`.

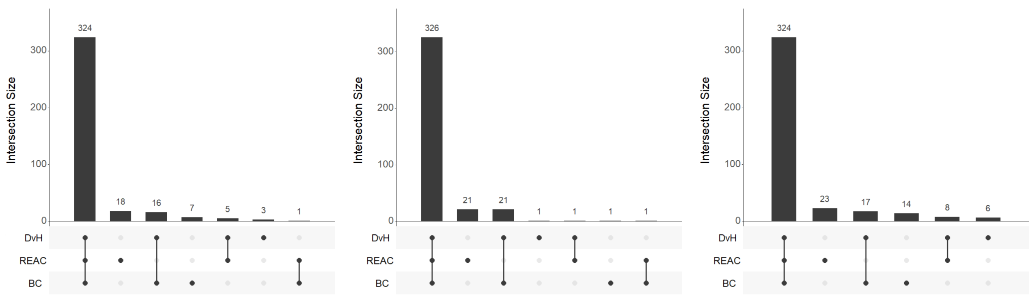
▻ Utilization of model order only for reactions. Referred as `REAC`.

▻ Barycenter for different Types. Referred as `BC`.

The approach of enforcing port order will not be evaluated, as enforcing the port order drastically changes the way LF models should be implemented; comparing this approach with a more lax approach would not be sensible. Models would require adjustments to utilize the full potential of fixed node orders. The analysis will additionally be done on the entire dataset without the distinguishment in the evaluation of cycle breaking. This creates up to nine different approaches to compare.

5.3.1 Matching Layouts

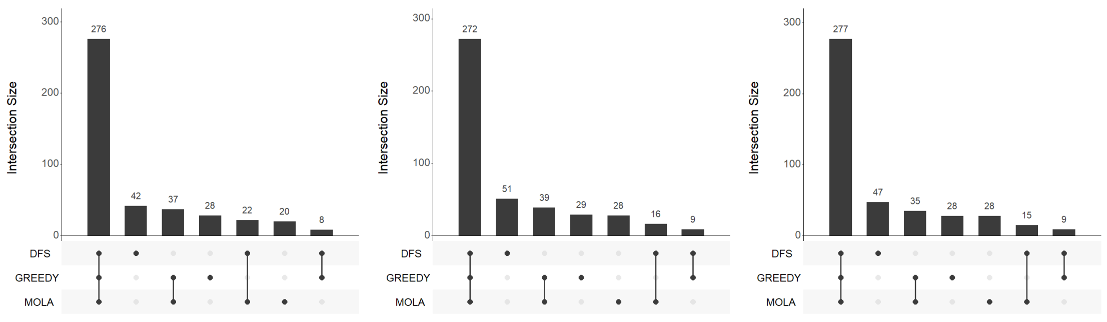
This analysis shows the effect size of the different crossing minimization options using UpSet plots. Figure 5.11 shows the matching layouts when using the same cycle breaker and switching the crossing minimization options.



(a) `DF-CB`. **(b)** `GREEDY-CB`. **(c)** `MOLA-CB`.

Figure 5.11. Differences induced by the crossing minimization option, for different cycle breaking strategies.

This figure shows that using the same cycle-breaking strategy and only adjusting the crossing minimizer produces the same amount in up to 94% of the models. Contrary to that, Figure 5.12 shows the differences induced by altering the cycle breaker and fixing the crossing minimization option. While this still creates identical layouts in up to 80% of the cases, the effect of cycle breaking on the layout is greater than the influence of crossing minimization.



(a) BC. **(b)** DvH. **(c)** REAC.

Figure 5.12. Differences induced by the crossing minimization option, when using different cycle breaking strategies.

For the following analysis in this chapter use data set with removed identical layouts based on cycle breaking.

5.3.2 Edge Crossing Analysis

Figure 5.13 shows the normalized analysis data for the edge crossing analysis based on crossing minimization options by combining the different cycle-breaking strategies. The data shows that the BC approach has a higher stability compared to the other approaches. It does not create layouts with a massively larger edge crossing count than the other options.

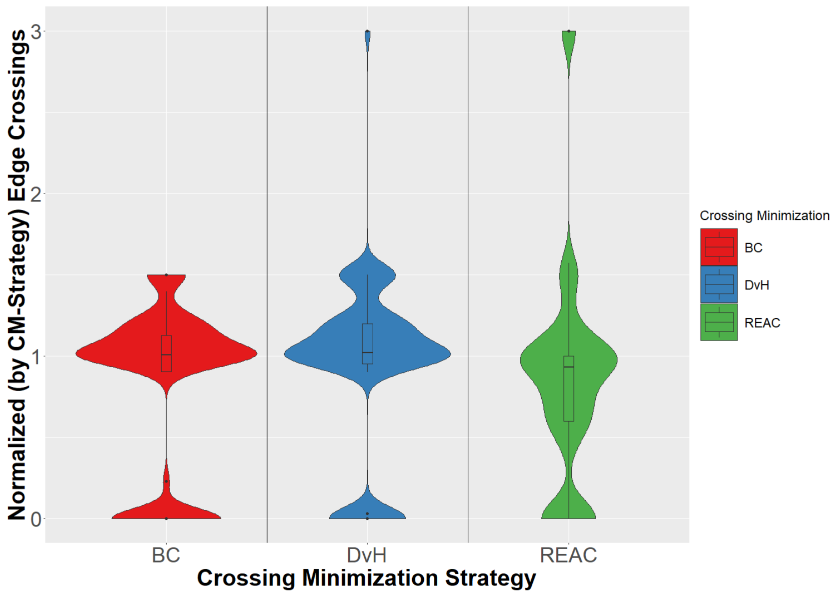


Figure 5.13. Normalized edge crossings, grouped by the crossing minimization options.

Additionally, the BC approach creates the most layouts without edge crossings where at least one other strategy could not eliminate all crossings. The REAC approach has the lowest average amount of edge crossings. The p-value comparing these datasets with the Kruskal-Wallis test is 0.0019, indicating statistically significant differences. The Willcoxon test only indicates significant differences between the REAC and DvH datasets, with a p-value of $4e-4$.

Figure 5.14 shows the differences induced by modifying crossing minimization for different cycle breakers. This data confirms the behavior shown in Figure 5.13. However, this data uncovers that the cycle breaker heavily influences the layouts in the scope of crossings; as for the DF-CB, no outliers are created.

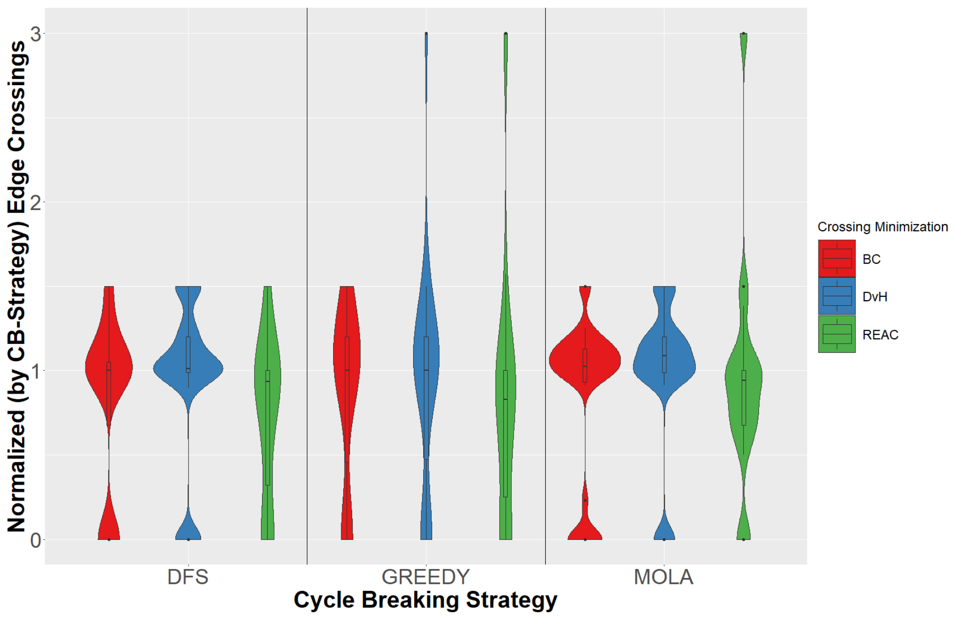


Figure 5.14. Affects of changes to crossing minimization for different cycle breakers.

Still, the DvH approach gives the highest normalized mean and median values for all different strategies. The lowest mean and mean values differ based on cycle breaking strategy and the REAC approach has the lowest median values for all cycle-breaking strategies.

Something to consider is that most models have not been created with a model order strategy for crossing minimization, and if without proper introduction into the effects of model order. The BC approach would benefit from this awareness, possibly reducing the edge crossing.

5.3.3 Aspect-ratio

When changing the in-layer order, the width and height of the entire layout change. Figure 5.15 shows the analysis data for the aspect ratio analysis. This data shows that the aspect ratio is relatively consistent for the same cycle breaker with different crossing minimization options. For all cycle breakers combined, the Kruskal-Wallis test fails to show statistically significant differences for the different crossing minimization options. With mean values of 3.11, 3.12 and 3.08 for the BC, DvH and REAC approaches, respectively.

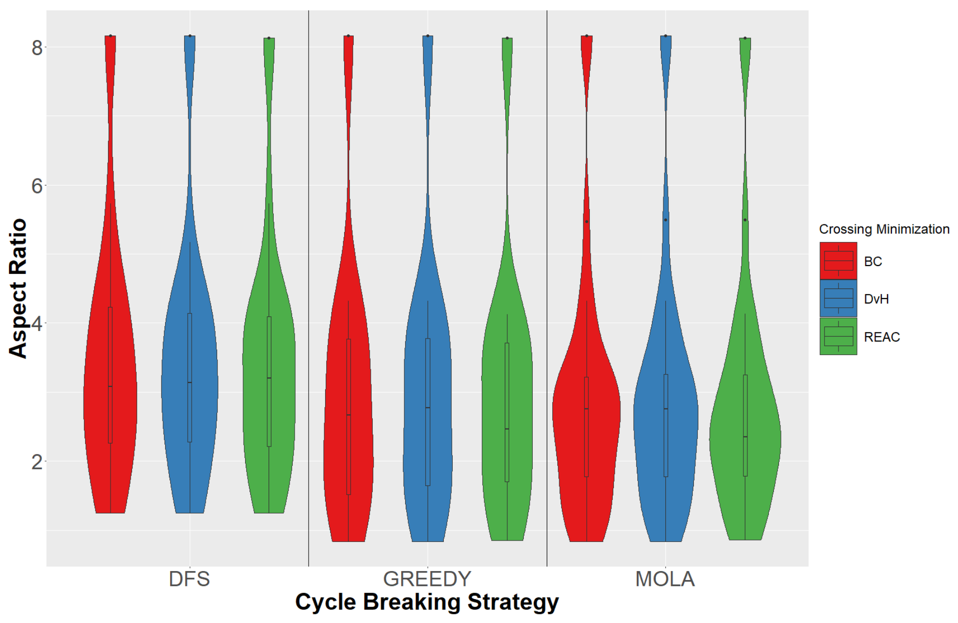


Figure 5.15. Aspect ratio comparison for different combinations of cycle breaking and crossing minimization

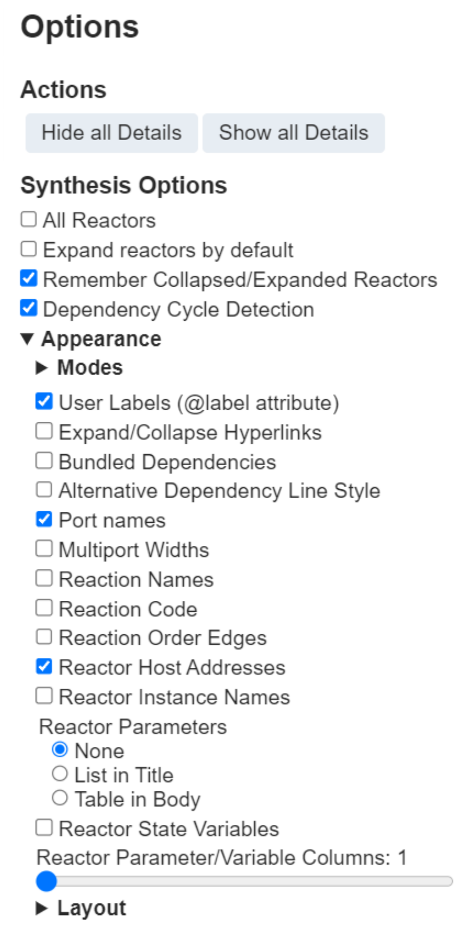
5.4 Feedback

Retrieving feedback from developers that actively use a programming language, or in this case, a modeling framework, is difficult for multiple reasons. Some of these reasons are listed below:

- Knowledge about layout algorithms. Explaining different approaches for a layout algorithm requires some knowledge about layout algorithms, to fully comprehend the expected induced differences.
- Personal preferences. Developers have different preferences for layout decisions. Some of these decisions are difficult to measure with objective metrics.
- Small number of participants. Methods like surveys require time and effort, which reduces the number of developers that are willing to participate. Additionally, getting in contact with knowledgeable persons can be difficult.
- Finding good examples. With the analysis results presented here it should be apparent that the dataset for showing differences is very small. Only three models show unique behavior for cycle breaking.

Two sources for feedback have been utilized. The different cycle-breaking strategies and approaches to modify crossing minimization have been introduced in the weekly Lingua Franca meeting. The second source of feedback is a meeting with a development team from Magnition, who actively use LF in their products.

The first response of feedback generally was: "This should be an option." However, this is not desirable, as explaining the induced differences in a tooltip , i.e., for swapping the cycle-breaking strategy, is not feasible. Figure 5.16 shows some of the currently displayed options for diagram manipulation, with the sections `Modes` and `Layout` collapsed.


Options

Actions

Hide all Details | Show all Details

Synthesis Options

- [ ] All Reactors
- [ ] Expand reactors by default
- [x] Remember Collapsed/Expanded Reactors
- [x] Dependency Cycle Detection

▼ Appearance

▶ Modes

- [x] User Labels (@label attribute)
- [ ] Expand/Collapse Hyperlinks
- [ ] Bundled Dependencies
- [ ] Alternative Dependency Line Style
- [x] Port names
- [ ] Multiport Widths
- [ ] Reaction Names
- [ ] Reaction Code
- [ ] Reaction Order Edges
- [x] Reactor Host Addresses
- [ ] Reactor Instance Names

Reactor Parameters
- ● None
- ○ List in Title
- ○ Table in Body

- [ ] Reactor State Variables

Reactor Parameter/Variable Columns: 1

▶ Layout

Figure 5.16. Diagram options in the VSCode extension.

The options windows seems overwhelming for first time users. The team at Magnition mostly uses three options of the list presented in Figure 5.16. These options being: `Multiport Widths`, `Port Names` and `Reactor Instance Names`. The presentation of too many choices can lead to indecision; in psychology this is known as choice paradox. A study about consumer behavior regarding the purchase of jam found that consumers bought jam in 40% of the cases with only six options. In contrast, consumers with 24 different options of jams only bought any jam in 3% of the cases [IL00]. Now replace jams with intricate algorithms with different options, where further explanation of these differences is not given in a tooltip but rather some guide on a website or a markdown file, this should further limit the desire to explore different decisions. Furthermore, Hick and Hyman found in their research that with increasing choices, the decision for any given choice increases logarithmically, known as the Hick-Hyman law [Hic52].

Initially, most feedback stated that reducing control is okay if it benefits the layout. However, after some reconsideration most participants argued that control and especially stability are more important. Using model order in the algorithm inherently creates stability, as drastic model order changes require drastically changing the entire model, at which point stability can no longer be given.

Finally, feedback concerned the ability to follow edges. Reducing edge crossings is vital for clear edges, but Magnition mentioned a feature used in a different modeling tool to reduce the number of visualized edges, further options are explored in Section 6.3.1.

Chapter 6

Conclusion

This chapter summarizes the results found in this thesis, followed by some aspects of the evaluation that could be improved. Afterwards, an outlook that explains future options to improve feedback generation and general options for better readability.

6.1 Summary

Previous analysis has shown that using model order in automatic graph drawing is beneficial for the layout creation. The current approaches utilize the model order for languages with an unimpaired model order. This thesis evaluates the possibilities of using the model order for languages where node declarations are grouped by their type. Lingua Franca models are used as a case study for languages with this behavior. A new property is added to each node for working with node groups, representing the group as an integer value.

Several options are explored to utilize this newly introduced group ID in the cycle-breaking phase of the layered algorithm. The results of the analysis fail to show a favorable strategy. The `MOLA-CB` performs worst when comparing edge crossings and backward edges. However, this strategy creates the layouts with the smallest aspect ratio. The model order greedy approach still achieves its original goal of creating a minimal number of backward edges. However, for some models, this approach creates layouts with a relatively high number of edge crossings. The `DF-CB` and the `SCC_CON-CB` perform well in all aspects, with slight advantages for the `SCC_CON-CB`. Additionally, this approach provides more control for the developer.

Another phase of the layered approach that was modified is the crossing minimization phase. The results show that the newly proposed approaches reduce the number of edge crossings compared to the current approach. While the `REAC` approach produces the best results on average, the `BC` approach reduces edge crossings, eliminates bad outliers completely, and retains the highest control. Arguably, with the `BC` approach enabled, the implementation of models will change and the edge crossing will be reduced even further.

The surveyed developers generally preferred the option to fix the order of ports. Even if this increases the initial overhead, stability is a big concern. This option would require developers to use it for some time to give a fair evaluation, as creating a good port order could become quite complex for large models, which are often divided into multiple files. However, the option of a fixed port order could be exposed to the developer and could be deactivated for extremely large.

General feedback from Lingua Franca's developers and developers who actively use it indicates that higher control is favored. While the overhead of working with the model increases in the short term, working with clear and readable layouts reduces the susceptibility to bugs and erroneous configurations.

6.2 Evaluation Revision

The biggest problem for evaluating different layout strategies is the sample size of models. This problem is aggravated for small samples if the compared strategies only induce minor differences or none for the small samples. As mentioned in the analysis, $77.\overline{7}\%$ of the models in this dataset are indifferent. Furthermore, only three models produce different results for all cycle-breaking strategies. This problem transfers to the evaluation with human feedback. Comparing different strategies is difficult if the sample size for different models is this small. One approach that could fix this problem is presented in the following section.

6.3 Future Work

Gathering more models for an evaluation is difficult for many reasons, depending on the source of the models.

- ▻ **Test Samples**: This dataset has the potential to grow as new features that require testing are introduced into the language. However, the test models are relatively small, as shown in the analysis. As a result, 94% of the samples are identical, with only three models showing more than two options.
- ▻ **Example Samples**: This dataset grows under the same circumstances as the test dataset. Exemplary models may explain new features for a given language. However, only a small number of models probably suffice to fully explain a feature, while testing is more extensive.
- ▻ **Experiments Sample**: This dataset is described as some experiments with the language and showcases future feature ideas. This dataset is the smallest of the open-source datasets because these new ideas only emerge occasionally.
- ▻ **Real World Samples**: While real-world samples are the most relevant, they are also the hardest to obtain. Real-world, large-scale development is rarely open-source. Obtaining these samples requires good connections with the development team.

One approach to gathering more real-world samples could utilize a tool that removes the source code of the target language. This code is irrelevant for evaluating layouts; however, it is why these models cannot be shared openly. The source code in the target language may be classified by the company. A tool that would allow development teams to remove this code and other classified parts like comments or even obfuscate the names of the nodes could increase the willingness to share models. This tool could be a command-line tool distributed with the main framework with a hint of helping development and improving the framework on the website.

6.3.1 Wireless Connections

The general feedback of the developers using Lingua Franca with an interactive diagram view concerned the ability to follow edges.

The diagram can get rather cluttered in large-scale diagrams with many multi-edge connections. One approach hinted at is an approach that is utilized in the design tool Origami Studio[1] by Meta. This tool utilizes what they call *Wireless Broadcasters* and *Wireless Receivers*. These are essentially connections

[1] `https://origami.design/`

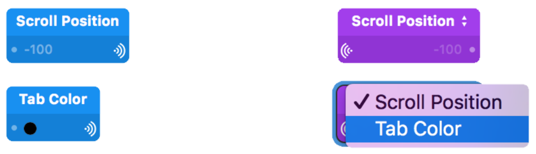


Figure 6.1. Wireless Broadcasters and Wireless Recievers in Origami Studio[2].

without the visual representation of an edge, as shown in Figure 6.1. These are especially useful for edges that target multiple nodes. Using a label for these connections, similar to port labels, would allow for multiple connections of this type without confusion.
These wireless edges could be created by introducing a new keyword or identifier like `-|>`, with the benefit of eliminating these edges from the layout creation, even in different instances of either Epoch or VS Code. Another approach would be to introduce a context menu for edges with the ability to transform the edge. The second approach would be a transient decision, which most likely would reduce the stability of the layout.

[2]https://origami.design/documentation/workflow/patchorganization

Appendix A

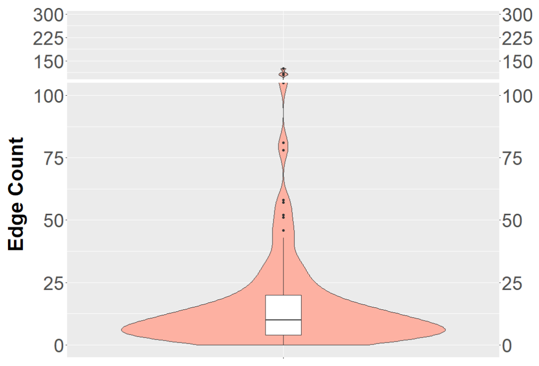
Appendix

A.1 Tarjan's Algorithm

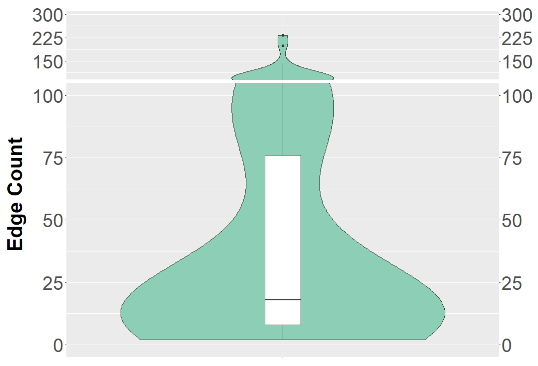
```
input: directed graph G = (V,E)
output: set of strongly connected components (vertex sets)

stronglyConnected = empty set of sets of nodes
index = 0
S = empty stack
for v in V:
   if v.index is undefined :
      scc(v)

function scc(v):
   v.index = index
   v.lowlink = index++
   S.push(v)
   v.onStack = true

   for (v,w) in E: //outgoing edges of E
      if w.index is undefined: //if a successor of v has not been visited, check it.
         scc(w)
         v.lowlink = min(v.lowlink,w.lowlink)
      else if w.onstack:
         // w is on stack and reachable it might be a lowlink
         v.lowlink = min(v.lowlink,w.index)

   if v.lowlink = v.index: //v is a root node
      strong = empty set nodes //new strongly connected component
      strong.add(v)
      do:
         w = S.pop
         w.onStack = false
         strong.add(w)
      while v != w
      if strong.size() > 1: //modification only power > 1
         stronglyConnected.add(strong)
```

Listing A.1. Pseudo code for Tarjan's algorithm

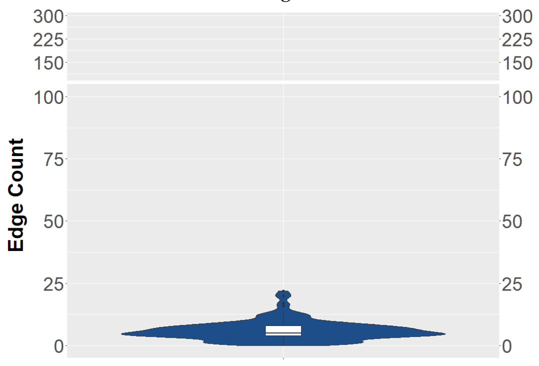
A.2 Edge Count Analysis



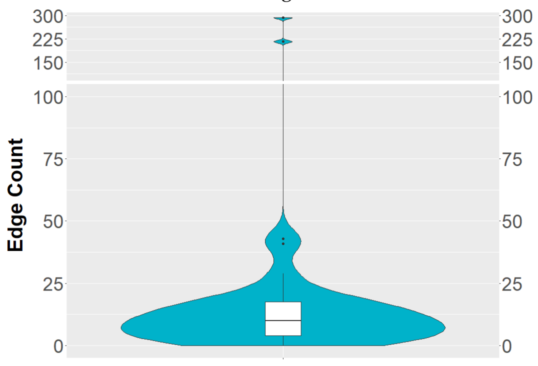
(a) LFEA Edge Count



(b) LFEP Edge Count



(c) LFT Edge Count



(d) `COLAB` Edge Count

Figure A.1. Comparison of the edge count in the different datasets.

Bibliography

[BMJ04] Wilhelm Barth, Petra Mutzel, and Michael Jünger. "Simple and efficient bilayer cross counting". In: *Journal of Graph Algorithms and Applications* 8.2 (2004), pp. 179–194.

[Bon36] C.E. Bonferroni. *Teoria statistica delle classi e calcolo delle probabilità*. Pubblicazioni del R. Istituto superiore di scienze economiche e commerciali di Firenze. Seeber, 1936. URL: https://books.google.de/books?id=3CY-HQAACAAJ.

[BRS+07] Chris Bennett, Jody Ryall, Leo Spalteholz, and Amy Gooch. "The aesthetics of graph visualization". In: *Proceedings of the International Symposium on Computational Aesthetics in Graphics, Visualization, and Imaging (CAe'07)*. Banff, Alberta, Canada: Eurographics Association, 2007, pp. 57–64.

[CMS99] Stuart K. Card, Jock Mackinlay, and Ben Shneiderman. *Readings in information visualization: using vision to think*. Morgan Kaufmann, Jan. 1999. ISBN: 1558605339.

[CTY07] Pierre Charbit, Stéphan Thomassé, and Anders Yeo. "The minimum feedback arc set problem is np-hard for tournaments". In: *Combinatorics, Probability & Computing* 16.1 (2007), pp. 1–4. DOI: 10.1017/S0963548306007887.

[DCS+23] Sara Di Bartolomeo, Tarik Crnovrsanin, David Saffo, and Cody Dunne. *Evaluating graph layout algorithms: a systematic review of methods and best practices*. 2023. DOI: 10.31219/osf.io/ms27r.

[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.

[DHS+23] Sören Domrös, Reinhard von Hanxleden, Miro Spönemann, Ulf Rüegg, and Christoph Daniel Schulze. *The eclipse layout kernel*. 2023. arXiv: 2311.00533 [cs.DS].

[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.

[EHN+17] Peter Eades, Seok-Hee Hong, An Nguyen, and Karsten Klein. "Shape-based quality metrics for large graph visualization". In: *Journal of Graph Algorithms and Applications* 21.1 (2017), pp. 29–53. ISSN: 1526-1719. DOI: 10.7155/jgaa.00405.

[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-O.

[FH10] Hauke Fuhrmann and Reinhard von Hanxleden. "Taming graphical modeling". In: *Proceedings of the ACM/IEEE 13th International Conference on Model Driven Engineering Languages and Systems (MoDELS '10)*. Vol. 6394. LNCS. Springer, Oct. 2010, pp. 196–210. DOI: 10.1007/978-3-642-16145-2.

[FR91] Thomas M. J. Fruchterman and Edward M. Reingold. "Graph drawing by force-directed placement". In: *Software—Practice & Experience* 21.11 (1991), pp. 1129–1164. ISSN: 0038-0644. DOI: http://dx.doi.org/10.1002/spe.4380211102.

[GJ79] Michael R. Garey and David S. Johnson. *Computers and intractibility: a guide to the theory of NP-completeness*. New York: W. H. Freeman & Co, 1979.

[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.

[Gra14] Martin Grandjean. "La connaissance est un réseau. perspective sur l'organisation archivistique et encyclopédique". In: *Les cahiers du numérique* 10.3 (2014), pp. 37–54. ISSN: 14693380. DOI: 10.3166/LCN.10.3.37-54.

[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.

[HFS11] Reinhard von Hanxleden, Hauke Fuhrmann, and Miro Spönemann. "KIELER—The KIEL Integrated Environment for Layout Eclipse Rich Client". In: *Proceedings of the Design, Automation and Test in Europe University Booth (DATE '11)*. Grenoble, France, Mar. 2011.

[Hic52] W. E. Hick. "On the rate of gain of information". In: *Quarterly Journal of Experimental Psychology* 4.1 (1952), pp. 11–26. DOI: 10.1080/17470215208416600.

[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.

[IL00] S. S. Iyengar and M. R. Lepper. "When choice is demotivating: can one desire too much of a good thing?" In: *Journal of personality and social psychology* 79.6 (2000), pp. 995–1006. ISSN: 0022-3514. DOI: 10.1037/0022-3514.79.6.995.

[KDM+16] Steve Kieffer, Tim Dwyer, Kim Marriott, and Michael Wybrow. "HOLA: human-like orthogonal network layout". In: *IEEE Trans. Vis. Comput. Graph.* 22.1 (2016), pp. 349–358. DOI: 10.1109/TVCG.2015.2467451.

[KW52] William H. Kruskal and W. Allen Wallis. "Use of ranks in one-criterion variance analysis". In: *Journal of the American Statistical Association* 47.260 (1952), pp. 583–621. ISSN: 0162-1459. DOI: 10.1080/01621459.1952.10483441.

[LCG+15] Alfredo Lezama, Irene-Angelica Chounta, Tilman Göhnert, and H. Hoppe. "Exploring visual stability in dynamic graph drawings: a case study". In: 2015. DOI: 10.1145/2808797.2809341.

[LKP18] Felice de Luca, Stephen Kobourov, and Helen Purchase. "Perception of symmetries in drawings of graphs". In: *Graph Drawing and Network Visualization - 26th International Symposium, GD 2018 Barcelona, Spain, September 26–28, 2018 Proceedings*. Ed. by Therese Biedl and Andreas Kerren. Lecture Notes in Computer Science. Springer, 2018, pp. 433–446. ISBN: 9783030044138. DOI: 10.1007/978-3-030-04414-5_31.

[MW47] H. B. Mann and D. R. Whitney. "On a test of whether one of two random variables is stochastically larger than the other". In: *The Annals of Mathematical Statistics* 18.1 (1947), pp. 50–60. ISSN: 0003-4851. DOI: 10.1214/aoms/1177730491.

[PCA02a] Helen C. Purchase, David Carrington, and Jo-Anne Allder. “Empirical evaluation of aesthetics-based graph layout”. In: *Empirical Software Engineering* 7 (3 2002), pp. 233–255. ISSN: 1382-3256.

[PCA02b] Helen C. Purchase, David Carrington, and Jo-Anne Allder. “Graph layout aesthetics in UML diagrams: User preferences”. In: *Journal of Graph Algorithms and Applications* 6.3 (2002).

[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.

[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.

[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.

[RAC+17] Ulf Rüegg, Marc Adolf, Michael Cyruk, Astrid Mariana Flohr, and Reinhard von Hanxleden. *Minimum-width graph layering revisited*. Technical Report 1701. ISSN 2192-6247. Kiel University, Department of Computer Science, Feb. 2017.

[Rie10] Martin Rieß. “A graph editor for algorithm engineering”. Bachelor Thesis. Kiel University, Department of Computer Science, Sept. 2010.

[Rie22] Max Riepe. “Model Order and Cycle Breaking in SCCharts”. Bachelor Thesis. Kiel University, Department of Computer Science, Mar. 2022.

[Sch11] Christoph Daniel Schulze. “Optimizing automatic layout for data flow diagrams”. Diploma Thesis. Kiel University, Department of Computer Science, July 2011.

[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.

[SSH12] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. *Transient view generation in Eclipse*. Technical Report 1206. ISSN 2192-6247. Christian-Albrechts-Universität zu Kiel, Department of Computer Science, June 2012.

[SSH13] Christian Schneider, Miro Spönemann, and Reinhard von Hanxleden. “Just model! – Putting automatic synthesis of node-link-diagrams into practice”. In: *Proceedings of the IEEE Symposium on Visual Languages and Human-Centric Computing (VL/HCC ’13)*. San Jose, CA, USA: IEEE, Sept. 2013, pp. 75–82. DOI: 10.1109/VLHCC.2013.6645246.

[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.

[STT81] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko 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.

[Tar72] Robert E. Tarjan. “Depth-first search and linear graph algorithms”. In: *SIAM Journal of Computing* 1.2 (1972), pp. 146–160.

[TDB88] Roberto Tamassia, Giuseppe Di Battista, and Carlo Batini. “Automatic graph drawing and readability of diagrams”. In: *IEEE Transactions on Systems, Man and Cybernetics* 18.1 (1988), pp. 61–79. ISSN: 0018-9472.

[TR05] Martyn Taylor and Peter Rodgers. “Applying graphical design techniques to graph visualization”. In: *Proceedings of the Ninth International Conference on Information Visualization (InfoVIS'05).* July 2005, pp. 651–656.

[Wil45] Frank Wilcoxon. “Individual comparisons by ranking methods”. In: *Biometrics Bulletin* 1.6 (1945), p. 80. ISSN: 00994987. DOI: 10.2307/3001968.

[WPC+02] Colin Ware, Helen Purchase, Linda Colpoys, and Matthew McGill. “Cognitive measurements of graph aesthetics”. In: *Information Visualization* 1.2 (2002), pp. 103–110. ISSN: 1473-8716.

[XCF+21] Shuangbin Xu, Meijun Chen, Tingze Feng, Li Zhan, Lang Zhou, and Guangchuang Yu. “Use ggbreak to effectively utilize plotting space to deal with large datasets and outliers”. In: *Frontiers in genetics* 12 (2021), p. 774846. ISSN: 1664-8021. DOI: 10.3389/fgene.2021.774846.

[ZKS11] Loutfouz Zaman, Ashish Kalra, and Wolfgang Stuerzlinger. “The effect of animation, dual view, difference layers and relative re- layout in hierarchical diagram differencing”. In: 2011, pp. 183–190.