CHALMERS
UNIVERSITY OF TECHNOLOGY

UNIVERSITY OF GOTHENBURG

Graph drawing strategies for large UML State Machine diagrams

Improving graph drawings usability

Master's thesis in Computer Science — Algorithms, languages and logic

Juan Pablo Contreras Franco

Department of Computer Science and Engineering
CHALMERS UNIVERSITY OF TECHNOLOGY
UNIVERSITY OF GOTHENBURG
Gothenburg, Sweden 2017

Master's thesis 2017

Graph drawing strategies for large UML State Machine diagrams

Improving graph drawings usability.

Juan Pablo Contreras Franco

Department of Computer Science and Engineering
Chalmers University of Technology
University of Gothenburg
Gothenburg, Sweden 2017

Graph drawing strategies for large
UML State Machine diagrams
Improving graph drawings usability.
Juan Pablo Contreras Franco

Supervisor: Marco Fratarcangeli, Department of Computer Science
Examiner: Carlo A. Furia, Department of Computer Science

Master's Thesis 2017
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg
SE-412 96 Gothenburg
Telephone +46 31 772 1000

Typeset in LaTeX
Gothenburg, Sweden 2017

Graph drawing strategies for large
UML State Machine diagrams
Improving graph drawings usability

Juan Pablo Contreras Franco
Department of Computer Science and Engineering
Chalmers University of Technology and University of Gothenburg

Abstract

As systems grow in complexity, their development and maintainability cost increase since there is a cognitive effort involved in the process of understanding their state and the relationships of their parts. This report describes how two graph drawing strategies can improve the depictions of UML state machines from a particular business case. The intention is to show new options to improve the readability and overall quality of the outcome produced by an in-house graph drawing solution. This project address the features of the problem that are concerned about the graph quality of the software modeling tools in use. These features relate to how the user perceives the state machine drawings. An implementation of a proof of concept is the base to explore an alternative graph drawing framework with the purpose of motivating a discussion about the feasibility of migrating the current graph drawing engine into a new one.
The work concludes that it is possible to customize an existing framework to fulfill the usability standards for UML state machine layouts. Further improvements on the proof of concept are required. Mainly, the geometric information must get involved in realistic scenarios.

Keywords: Graph Drawing, UML 2, State Machines, Graph Algorithms, OGDF

Acknowledgements

Dedicated to Clemencia and Judith for their generous and unrelenting commitment to support my efforts toward the actualization of my potentialities.
In addition, I want to thank Tomas Nilsson, Martin Lanzén and Marco Fratarcangeli for the opportunity they gave me.

Juan Pablo Contreras Franco, Gothenburg, August 2017

Contents

List of Figures

List of Acronyms

AGD Algorithms for Graph Drawing

API Application Programming Interface

EPG Evolved Package Gateway

ESC Easy StateChart

GDT Graph Drawing Toolkit

GEXG Graph Exchange XML Format

GML Graph Modeling/Markup Language

GPLv2 GNU General Public License v2.0

GPLv3 GNU General Public License v3.0

GUBT Graph Usability Benchmarking Tool

JPEG Joint Photographic Experts Group

MSAGL Microsoft Automatic Graph Layout

OGDF Open Graph Drawing Framework

PNG Portable Network Graphics

SVG Scalable Vector Graphics

UML2 Unified Modeling Language 2

UML Unified Modeling Language

XML eXtensible Markup Language

YAGT Yet Another Graph Tool

1

Introduction

1.1 Background

The business market in which Ericsson operates is mainly oriented to the implementation and operation of packed-switched networks. The business model thriving around these networks depends upon data flow auditing, live package inspection and the interaction of a manifold of devices and software applications.

Evolved Package Gateway (EPG) is the infrastructure solution developed at Ericsson to look after those telecommunication business requirements. The size of the system, measured in terms of the number of states and quantity of involved assets, compels system administrators to find ways that reduce the amount of effort required to understand the system and its state.

Managing the complexity required a development effort from Ericsson whose result is a software product to interact and ameliorate network's governance by reducing the cognitive overload burdening its operators. Even though the system can be modeled as a collection of well-defined state machines, it is not easy to grasp for its operators. EPG eases system's comprehension by allowing the user to interpret the system by representing it as UML compliant state machines.

Previous efforts at Ericsson accomplished the objective by implementing a graphical tool to draw the state machines. The preceding tool was meant to be an Eclipse plugin that transforms a state machine text description into an SVG image [1]. Those tools follow the scheme shown at Figure 1.1

The graph depicted in the SVG image complies with most of the UML2 standard extended with some modifications. The results were good enough to guarantee the fulfillment of the primary goal; however, some software pieces that were used as intermediate steps have limitations impacting the overall usability of the drawing.

The output obtained from the previous approach can be enhanced if, instead of using the old intermediate steps (e.g., heavily dependent on Graphviz[1], as it is shown in the figure 1.2), a new procedure that uses libraries explicitly tuned for UML layout generation. Also, it becomes necessary to be able to classify what is a good state machine layout in order to decide if there has been a definite improvement.

[1] Graphviz is a software tool to generate graph drawings from a textual specification. See 2.4.1.2 for a detailed explanation

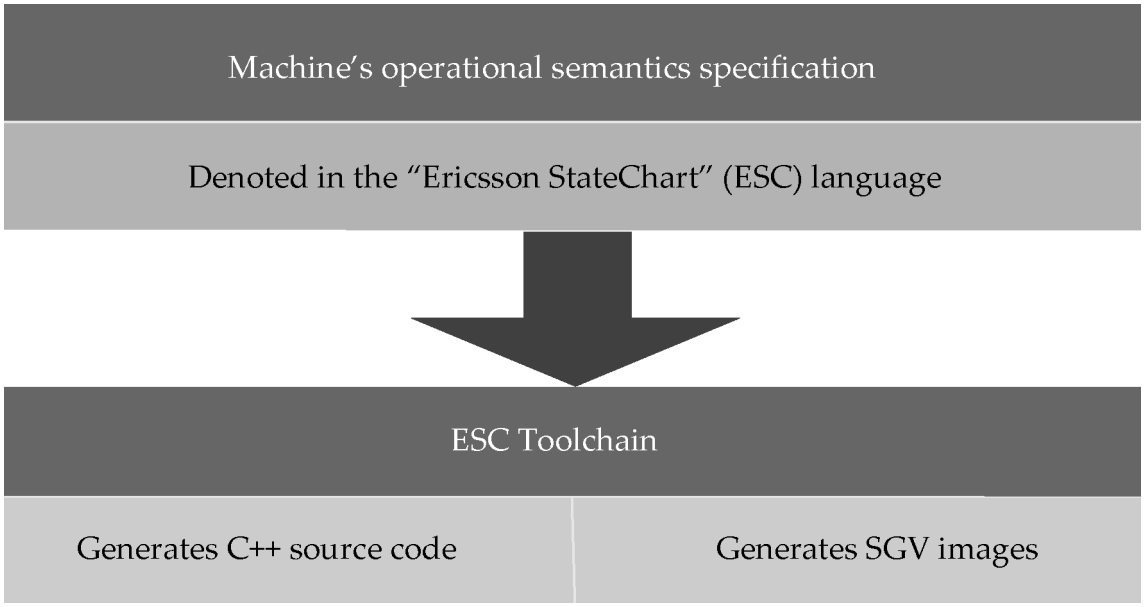


Figure 1.1: Current toolchain.

1.1.1 Large state machines as layout challenges: the Ericsson's experience

The company has developed a tool called Easy StateChart (ESC) meant to draw UML2 compliant state machines (also known as *statecharts*) from a homegrown language to denote state machines.
The use case's flow starts when the user generates a description of the state machine as a text file containing the state machine's description (a `.esc` file.) This description is the input both for the generation of further software artifacts and also for the drawing of UML2 State Machine diagrams.
The company has tried two different approaches, each one focusing on the possibilities of the previously mentioned tools. The first approach is to use `Graphviz`' `dot` language in such way that it is possible to produce UML state machine diagram

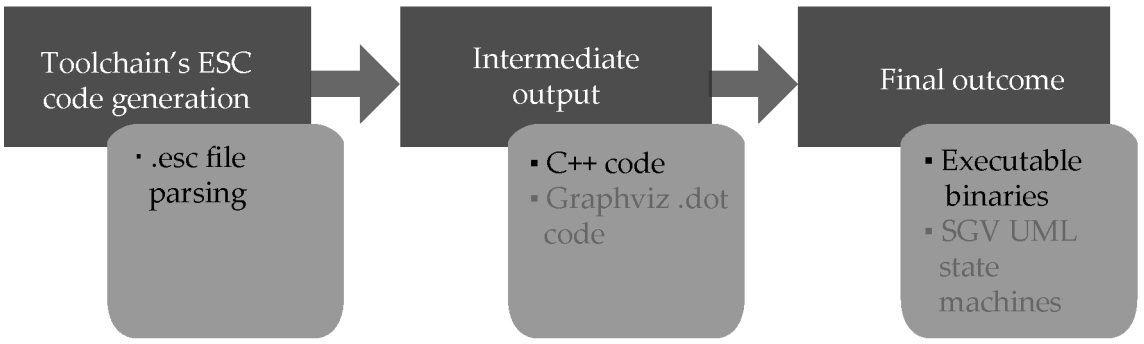


Figure 1.2: Current graph generation pipeline.

shapes by composing the default `Graphviz` shapes into UML resemblant forms. The second one is to use `PlantUML`[2] since it has the default UML forms and its language to denote state machine diagrams. It is important to mention that `PlantUML` uses `Graphviz` as its engine, so the potential and actual shortcomings of `Graphviz` are propagated to `PlantUML`.
Both approaches are not entirely satisfying in different ways. For some state machine sizes, the readability of the state machine diagram is heavily impaired by the geometric characteristics in the shapes. From figure 1.5, it is possible to enumerate the following undesirable characteristics:

- Most of the edges are candidates to simplification since they can fuse into a single line. At only one point of that line, it divides to show that the transition is triggered by a different event (referenced by the label.)
- Edges contain unnecessary bends.
- Each edge's label break its flow.
- States appearing at the same height are not aligned.

As previously mentioned, `Graphviz` does not possess the concept of a UML shape as a basic drawing object: the language it uses to describe the graphs is not fit for the representation of the UML state diagram entities by default.
The lack of primitive UML shapes in `Graphviz` begets layout drawing problems, that is, images are drawn either incorrectly or aesthetically flawed (i.e. asymmetrical shape arrangement) without a reasonable or evident cause. Another example is the absence of a consistent layout for edge labels (i.e. edge labels are described in the `dot` language as fine-tuned nodes having a translucent boundary.)

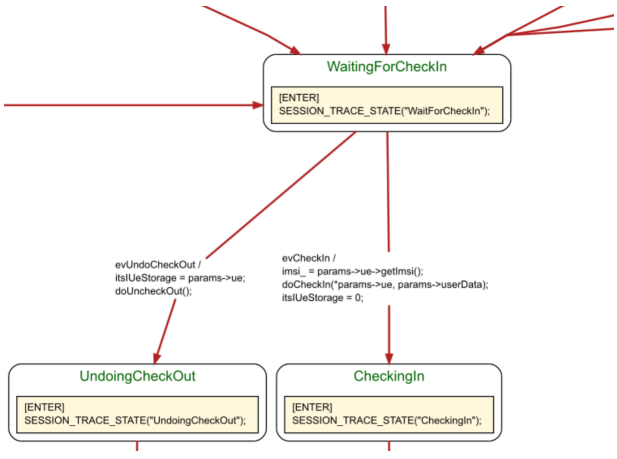


Figure 1.3: This figure shows some of the existing layout improvement opportunities in a small region of a larger diagram (see figure 1.5). The edge endpoints on the topmost state are drawn asymetrically. Similarly, other states are placed asymetrically and their corresponding edges are either broken or bent. Edge labels have that undesirable placement because ESC process the labels as nodes with translucent borders.

These shortcomings suggest that there is room for improvement for `Graphviz` by finding a way to make it UML friendly: this could be achieved if its algorithms are tuned to process the UML standard shapes. As for `PlantUML`, the shortcomings are not entirely related to the graph's layout drawing, but to the impact of model's

[2] `PlantUML` is a software tool based on `Graphviz` to generate UML graph drawings. See 2.4.1.2 for further reference

shape quantity in the final depiction layout quality. From the company's experience dealing with the processing of its typical state machines, it was that `PlantUML` was inadequate to draw large state machine diagrams.

1.1.2 Enhancing UML drawing strategies

The previous discussion is an offshoot of the Ericsson's use case and ongoing situation. It drives the inquiry about graph drawing enhancement into the topic of Information Visualization given that ESC users require apprehending the sense of large data amounts through abstracting the relevant issues from the context and dismissing negligible data.
Information Visualization comes as a field that, by fostering user's interaction with widgets, "enable users to explore patterns, test hypotheses, discover exceptions, and explain what they find to others... [and by] interacting with the dataset gives users the chance to rapidly gain an overview..." (Bederson and Shneiderman 2, preface) and is a "communication enhancer." [3, ch. 1]
Graph Drawing is the branch of Information Visualization dealing with the study of graph rendering for the purpose of human appreciation and analysis. Graph Drawing is applicable whenever the information elements being represented have significant relations between them [4] given the representation's relation to the knowledge field it belongs (e.g.,data is *structured*).

1.1.3 The Easy StateChart Language

As per Ericsson's technical documentation about ESC defines,

> ESC is a language for specifying state machines (known as *statecharts*), as well as a suite of tools for working with them. ([5])

A toolchain including code generators, parsers, and executable files provides the language implementation. The common use case starts when the user executes the code generator to produce `C++` code implementing the behavior on the state chart. The structure of an `.esc` file consists of textual declarations, states, and transitions. The tool suite supports two language variants: `Uml` and `Simplified`. The `Uml` variant is meant to support the UML2 standard completely while the `Simplified` vari-

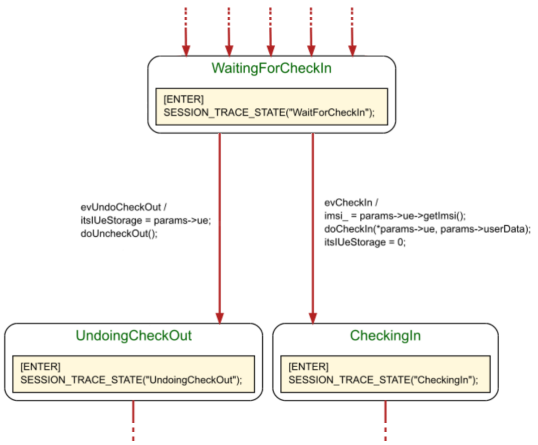
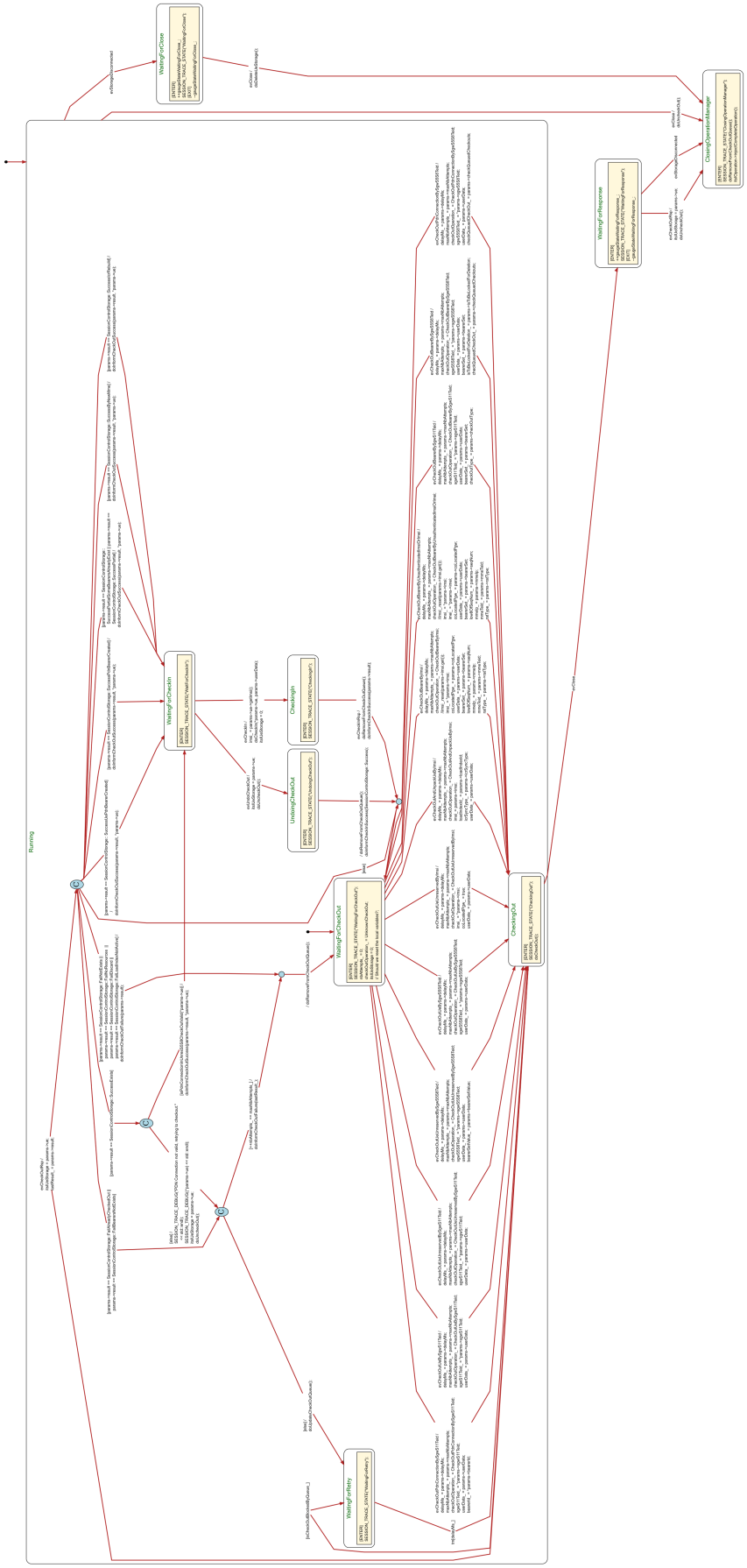


Figure 1.4: An enhanced version of figure 1.3. Here, the edge endpoints on the topmost state are drawn on the top of the shape without being merged in a single arrow, since they come from different transitions. The other states are distributed equitably with continuous, symmetric and straight edges. Edge labels are placed gracefully at the sides of their transition arrows.

Figure 1.5: State machine embodying some of the undesirable traits enumerated at 1.1.1.



ant is not exhaustive but generates optimal code in comparison: “Junction points,” “Choice points,” “History Sates” and a few other shapes belong to the `Uml` variant. As an example, the following fragment shows how to denote an event in the ESC language:

```
Events
evA
evB(int x)
evC(std::string y, std::auto_ptr<Imsi> z)
```

From this language, the toolchain can generate the following `C++` code:

```
class evA {
    public:
};

class evB {
    public:
    evB(int x) : this->x(x) {}
    int x;
};

class evC {
    public:
    evB(std::string y, std::auto_ptr<Imsi> z) : this->y(y), this->z(z) {}
    std::string y;
    std::auto_ptr<Imsi> z;
};
```

The language enforces naming conventions to relate state machine entities into language constructs. For instance, event names shall start with “ev” followed by any number of alphanumerical characters. Another relevant example is illustrated on the state transition signification:

```
Red
evChangeColor -> Green
Green
evChangeColor -> Red
```

The toolchain can produce SVG files. The files that function as the sources for the SVGs share the same structure.

1.2 Problem Formulation

This section introduces the different facets of the problem to be solved. The main topics are reviewed in such way that the expectations and limitations are made evident.

1.2.1 State machines in UML2

As it has been shown in Section 2.3, there are many graph rendering approaches. Some of them are unsuitable for certain problems (e.g.,drawing a tree might use techniques that are meaningless for class diagram rendering); still, others are advantageous within an application domain. Thereupon, for the sake of restricting the thesis scope, it is necessary to define accurately which UML state machine features are going to be implemented and which ones are going to be left behind.

In this fashion, a formal version of the problem statement is to *find a drawing of a graph that optimizes the graph usability.* For this purpose, the usability must be objectively measurable by thoughtfully defined metrics, as Section 2.2 will explain in more detail.
As a starting point, the state machine specification to be rendered is taken from the UML2 standard, as referred to earlier. The purpose of the state machine is thoroughly captured by the diagram's specification:

> The state machine view describes the dynamic behavior of objects over time by modeling the lifecycles of objects of each class. Each object is treated as an isolated entity that communicates with the rest of the world by detecting events and responding to them. ([6, ch. 7, p. 81, Overview])

Also, the meaning implicit in the product's diagram must fit the UML formal definition for state machine:

> A specification of the sequences of states that an object or an interaction goes through in response to events during its life, together with its responsive effects (action and activity). ([6, ch. 14, p. 604, Dictionary of terms])

These definitions are relevant because they confine the types of possible layouts to work with; consequently, they are helpful for the candidate approach evaluation.

1.2.1.1 Shapes for state machine diagrams

The UML standard sets up a variety of symbols to signify the understanding of the behavior conveyed in a machine state.
The official definition refers to them as "graphs containing *states* and *transitions*" as well as the "response of an instance of the class to *events*." They model *possible life histories of an object* and concurrency [6].
The previous definitions are relevant in as much as the nouns they contain classify and summarize the pertinent state machine's concepts and related shapes, including their varieties. They also suggest the outline for the relevant UML concepts to be implemented on this project:

- Events ('Call', 'Change', 'Signal' and 'Time')
- State ('Simple', 'Orthogonal', 'Nonorthogonal', 'Initial', 'Final', 'Terminate', 'Junction', 'Choice', 'History', 'Submachine', 'Entry point' and 'Exit point')
- Nested states (A state that groups many substates)
- Composite states (including 'orthogonal compositions' meant for the representation of concurrent executions.)
- Transition ('entry', 'exit', 'external', 'internal')

1.2.1.2 Layout quality

Besides the seminal work made by Purchase, there is recent work playing its part in the development of readability quality measures. Other authors that have treated

the problem of UML layout quality are Wong and Sun, Störrle and Galapov and Nikiforova. As mentioned in Störrle [9, §3], there are four level of design principles:

- General graphical design and visualization principles.
- Gestalt [11] principles have to be respected.
- Careful color use.
- Text readability by tuning the font, style, size, alignment, etc.

These principles are an example of the guides to decide on the necessary requirements of usability criteria.
UML state machines do not follow, by standard, any given layout recommendation. Also, even though there are general design conventions not yet mentioned in this document, Wong and Sun [8], Fuhrmann and von Hanxleden [12] and other authors have recommended layout parameters and investigated about their suitability for drawing's readability improvement.
The prevailing Ericsson's standards define that the state machines must be depicted from top to bottom and from left to right. Similarly, current diagrams tend to have edges with many irregular bends, as it is possible to notice in the figure 1.5.

1.3 Project's scope

This project address the features of the problem that are concerned about the graph quality of the software modeling tools in use. These features relate to how the user perceives the state machine drawings since there is a correlation to the usability of the graph with the cognitive workload when using software modeling tools and overall productivity.
This work explores an alternative graph drawing framework with the final goal to motivate a discussion about the feasibility of migrating the current graph drawing engine into a new one.
This work concludes that it is possible to customize an existing framework to fulfill the usability standards for UML state machine layouts.
Further improvements on the proof of concept are required. Mainly, the geometric information must get involved in realistic scenarios (e.g., real SVG outputs.) This outcome is not included in the final proof of concept because the primary focus was the evaluation of the layout strategies.

1.4 Report's scope boundaries

As a graph related problem, the amount of information conveyed ordinarily on an `.esc` file limits the size of the input. This work does not consider algorithmic problems on general graphs since the data's size for the business case never grows beyond tractable limits (i.e. edge and node set cardinality is seldom greater than fifty elements).
Likewise, the blending of the improved layout geometric data inside the old SVG documents has been left aside since the post processing task was not relevant to the

chief objective. However, the current `Graphviz` engine might become remarkably irrelevant after the completion of such endeavor.
Finally, a quantitative comparison between the data from the original graphs and their improved counterparts is absent because geometric information scraping from the prototype's output is not a priority.

1.5 Outline of this work

The first part of this work will cover simple graph theory notions side to side with providing some context regarding the technical details of the company's business case.
Subsequently, a discussion on the relation between the quantitative and qualitative characteristics in a graph will show the relevancy of the topic as well as introduce the notion of an aesthetic metric.
A short survey on graph drawing techniques, with a particular focus on UML drawing strategies, motivates the technical exposition.
A review of software tools precedes the discussion of the methods to tackle the problem at hand.
Finally, the results are discussed, and some conclusions with additional suggestions close the document.

2

Theory

As Purchase et al. [13] mentions, two factors are encompassing the challenges of graph representation: computational efficiency and conformance to measurable aesthetic criteria; hence, the study of Graph Drawing requires the blending of the Graph Theory and Information Visualization subfields. Graph compliance to minimum aesthetic thresholds subdues the implementation of algorithmic approaches to the possibilities of finding a reasonable computational efficient solution.
The following subsections will provide an overview of the concepts behind graph aesthetics, graph algorithms and how the aesthetic criteria impose efficiency goals to the algorithms. These ideas come mostly from the bibliography and are mentioned for the purpose of assisting the reader in the understanding of the ensuing sections. Subsequently, an itemized summary of tentative frameworks that can be helpful to close the requirement will conclude the subsection.

2.1 Definitions and notation

It is necessary to introduce basic definitions and notation related to graph theory to understand the subsequent discussion. Some of these definitions are standard knowledge in the field; however, graph drawing methodologies literature (e.g., [14, 15]) introduce some items that focus on this particular domain.

2.1.1 Graph Theory required notions

Since UML state machines are complicated geometric drawings involving text, lines, and many other shapes together with the concept of directionality, it becomes a necessity to abstract their basic properties for the sake of simplifying the analysis. This fact validates the introduction of simple vertex-edge undirected graphs as a practical model for further discussion.

Definition 1 (Graph) *A graph G is a pair composed of a set of vertices V and a set of edges E. Each edge $e \in E$ has a set of one or two elements from V known as its endpoints.[16]*

Some theorems regarding planar graphs (to be introduced at definition 8) and other meaningful observations rely on the concept of number of edges incident to a node.

Definition 2 (Node degree) *The degree of a node, $degree(u_i)$, is the number of edges incident on the node u_i.*

As the topic at hand is the display of state machines in a two-dimensional surface, it is necessary to introduce the notion of the drawing belonging to a graph.

Definition 3 (Graph drawing) *A drawing $D(G)$ (also known as an embedding [14]) is a map that assigns $v_i \in V$ into distinct co-ordinate pairs (x_i, y_i). A graph drawing also includes a map of the edges (u, v) for $u, v \in V$ into finite sequences of distinct co-ordinate pairs representing the bends of the polylines for the edges in $D(G)$.*

Notice that the notion of *graph drawing* implies that any given graph has many possible drawings.
Sometimes it becomes necessary to simplify a graph model as much as possible. That motivates the definition of the *auxiliary graph* as it is given by Purchase [15]. Its relevance will become evident in later sections.

Definition 4 (Auxiliary Graph) *A drawing $D'(G)$ is the graph obtained after transforming the bends on each polyline of the drawing $D(G)$ into distinct nodes. Such transformation implies that the edges of $D'(G)$ are straight lines. The name for this transformation is 'bend promotion.'*

An auxiliary graph is a representation whose usefulness is not limited to edge bend counting. Just as edge bents change into nodes inside the auxiliary graph, the process of crossing counting also requires the definition of cross promotion. That process is the reinterpretation of each crossing as a node inside the new auxiliary graph.
Since the business requirement focuses only on state machines, the notion of connected graph limits the scope of the analysis on this document.

Definition 5 (Walk) *In a graph G, a walk from vertex v_0 to vertex v_n is an alternating sequence*

$$W = \langle v_0, e_1, v_1, e_2, \ldots, v_{n-1}, e_n, v_n \rangle$$

of vertices and edges such that the e_i are the shared endpoints of the vertices that precede or follow it in the sequence [16].

Definition 6 (Connected Graph) *A graph is connected if for every pair of vertices u and v there is a walk from u to v*

Planarity is an interesting graph property, mainly because of its relationship with the number of crossings in a drawing since there is a correlation between the planarity of its subgraphs and the overall crossings metric [14].

Definition 7 (Planar drawing) *A planar drawing of a graph is a drawing of the graph in the plane without edge-crossings [16].*

Definition 8 (Planar graph) *A graph is said to be planar if there exist a planar drawing for it[16].*

2.2 Graph aesthetics

Alongside with the minutiae of the algorithms that solve graph theory problems (i.e. finding structures with distinctive characteristics within the graph), the new quantifiable aesthetical parameters require further treatment. The introduction of these parameters, as Di Battista et al. [14] declare, allow to ascribe the subjective qualification of a graph rendering to quantifiable parameters; thus, graph rendering requirements are unequivocally settled.
An obvious requirement for graph drawing is *computational efficiency* as a guarantee of responsiveness; besides this, there are other relevant parameters:

Drawing convention "A *drawing convention* is a basic rule that the drawing *must* satisfy to be admissible." [14, ch. 2] For instance, in the application domain of state machines, the chosen shapes to be rendered as the meaningful representations are those defined by the UML convention (e.g.,s̃olid circles representing initial states, etc.)

Aesthetics "*Aesthetics* specify graphic properties. . . to achieve readability." [14]

Constraints The *constraints* refer to the specific drawing conventions and aesthetical requirements of *subgraphs* and *subdrawings*. [14]

As mentioned previously, representation's usability (e.g.,t̃o render consistent and meaningful graphs) brings about the need to outline further properties about the presented objects. Purchase [17] and Kaufmann and Wagner [18] list a catalog of quantifiable graph properties whose relationship to the subjective perception of quality has been systematically evaluated on users. These features are called "Aesthetic criteria," (*aesthetics*) and encompass measurable characteristics like "crossing minimization," "bend minimization," "display symmetry" and "clustering" etc.
Given that Graph Drawing relies on the possibility of implementing feasible algorithmic solutions to common graph problems, it is clear that optimizing the rendering under the constraints imposed by the potential conflicts between aesthetics is a formidable endeavor [14, sec. 2.2]; for this reason, it is a necessity to set priorities on the constraints and make suitable tradeoffs.
Another possible use for graph metrics is to define the objective function for the families of algorithms that make use of such procedures. An example of this method is the application of similar techniques in simulated annealing algorithms [19–21].

2.2.1 Aesthetical considerations about UML diagrams rendering

Being UML diagram rendering a subset of the generic problem of graph drawing, some authors have made experimental studies about the impact of different aesthetic prioritization and enforcement within the Model-driven software development community.
Purchase et al. [13] elaborate on various criteria mostly relevant for the UML's application domain (e.g., "font type," "layout's width," "orthogonality," etc.) Even

though this study is only related to UML class diagrams and collaboration diagrams, the conclusions are suitable to be extrapolated to other UML diagram types.
On a similar stance, Galapov and Nikiforova [10] advocate for the application of "general layout principles" (e.g.,̃laws of object perceptual organization: the *law of similarity*, the *law of continuation*, the *law of proximity*, etc.) by quoting previous work about those topics from Boff et al. [22]. Among many other authors, they cite[1] Wong and Sun [8] to substantiate their advocacy for UML layout general principles. Their work's value is the restriction of universal aesthetic criteria to the narrower field of UML layout.

2.2.1.1 Graph metrics

Purchase [17] has done comprehensive experimental studies involving real users and measuring the cognitive impact of graph layout. Besides these studies, the outcomes shown in her extensive work[2] are the result of the efforts towards building a methodological framework to relate the quantitative aspects of the aesthetic criteria with the user's qualitative perception. Her work shows that it is possible to correlate and prioritize the criteria based on the statistical analysis by carefully setting up surveys analyzing user's impressions and cognitive improvements.
Defining graph metrics pursues, as it leading goal, the measurability of the geometric characteristics of a graph drawing by disregarding uninteresting information and, in general, any information impeding the comparison between two drawings without consideration for their structural peculiarities.
By taking aside the irrelevant information, it becomes possible to compare the quality of two graphs by disregarding the cardinalities of their vertices and edges sets and any other possible structural characteristics they may possess.
The design of these metrics additionally purports to define the quantities as restricted, dimensionless numbers. That is, every metric is a number in the interval $0 \leq x \leq 1$. Also, by design, values at the interval's rightmost extreme refer to a better aesthetic value.
Besides Purchase approach, there have been innovative efforts toward the simplification of metric gathering by the aggregation of geometric information into simpler figures. Huang et al. [24] encourage the use of *aesthetic aggregation* by designing a metric that gathers the geometric information into an overall score. Although this idea seems compelling, it has not been widely adopted in the field's literature at the time of writing this report and it is out of consideration for its purposes.
The next paragraphs are meant to introduce Purchase's approach to graph metrics. For the purposes of the following discussion, n' will stand for the number of nodes in a drawing and m' be the number of edges as well.

Edge crosses The aesthetic goal for the edge crossings ($\aleph_c$) metric is to reduce it as much as possible. Almost all experts agree on the desirability edge crossing reduction, in consequence, the metric considers the proportionality of the quantity goodness by subtracting the main quotient from 1.

[1] "Requirements set for layout diagram elements" [2] Further reference at Purchase [7], Purchase et al. [13], Purchase [15] and Purchase [23].

The central problem to overcome is the absence of an unambiguous upper bound for the number of crosses that could potentially be part of a drawing.

Definition 9 (Crossings) *A cross promotion is applied to $D(G)$: every cross on the drawing becomes a node producing a drawing $D'(G)$. For the purposes of generating the upper bound, it is assumed that every segment from $D'(G)$ will intersect every other edge. c_{all} fulfills that role:*

$$c_{all} = \frac{m'(m'-1)}{2}$$

In addition, there are impossible crossings given the existence of adjacent edges. $c_{impossible}$ counts the number of such events:

$$c_{impossible} = \frac{1}{2}\sum_{i=1}^{m'} degree(v_i)(degree(w_i)-1)$$

where w_i and v_i are the nodes of the ith edge. This leads to the formulation of the upper bound c_{mx}:

$$c_{mx} = c_{all} - c_{impossible}$$

Finally, the metric is defined as a quotient and reinterpreted in such way that greater values of $\aleph_c$ represent the absence of crossings between edges:

$$\aleph_c = 1 - \begin{cases} \frac{c}{c_{mx}}, & \text{if } c_{mx} > 0 \\ 0, & \text{otherwise} \end{cases}$$

Edge bends Intuitively, edge bends ($\aleph_b$) are the amount of points of the polyline connecting two nodes in the drawing that do not belong to an hypothetical straight line connecting them.
As with the Edge Crosses metric, the problem of scaling appears. This is a calling to attempt the finding of an upper bound as explained in the following definition.

Definition 10 (Bends) *After bent promotion, the number of bends is:*

$$\begin{aligned} b &= n' - n \\ &= m' - m \end{aligned}$$

To avoid the problem of having to compare the number of bents to the possibility of having an infinite number of bends, the scaling is taken from the number of segments in the promoted drawing:

$$b_{avg} = \frac{m'-m}{m'}$$

Finally, the metric is reinterpreted:

$$\aleph_b = 1 - b_{avg}$$

The Figure 2.1 shows the meaning of converting the bends into pseudonodes.

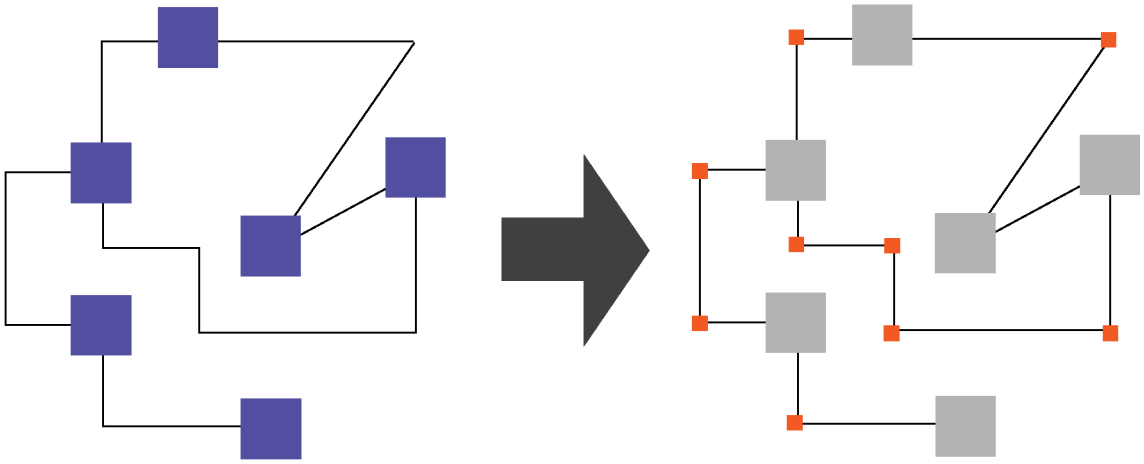


Figure 2.1: Bent promotion

Minimum angle The foundation for this metric ($\aleph_m$) resides on defining the existence of an optimal angle ϑ_i that relates the degree of each singular node to congruent segments of a complete circular arc. Once ϑ_i is defined, the procedure is to measure and normalize the amount of deviation for each of the nodes.

Definition 11 (Minimum angle) *First, define the nature of an optimal angle:*

$$\vartheta_i = \frac{360°}{degree(v_i)}$$

Then, define $\theta_{i\,min}$ as the minimum angle between the incident edges of a node n_i. This is enough to calculate the overall deviation of the edge's angles:

$$d = \frac{1}{n}\sum_{i=1}^{n}\left|\frac{\vartheta_i - \theta_{i\,min}}{\vartheta_i}\right|$$

Finally, the metric is normalized.

$$\aleph_m = 1 - d$$

As an example, suppose the existence of a drawing in which each node has the perfect measure (e.g., $\theta_{i\,min} = \vartheta_i$ for every node n_i). Then, the deviation is $d = 0$ and consequently $\aleph_m = 1$ (i.e. that drawing has the best possible angle setting.) The Figure 2.2 shows on the left the perfect angle distribution for the drawing at hand in comparison to an uneven angle distribution in a different representation of the same graph.

Other metrics The following metrics are defined in Purchase [17]. They are not given relevance in this report either because they had no meaning for its final purpose or because their implementation was outside the scope of this work for the reason of their complexity.

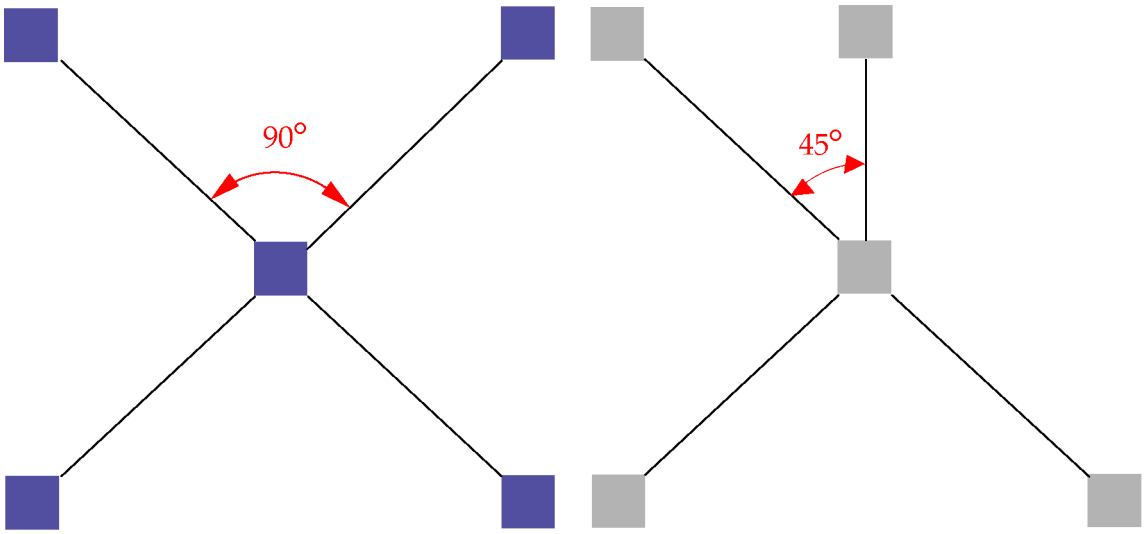


Figure 2.2: Minimum angle

Symmetry The calculation of this metric ($\aleph_s$) may involve comparing the shapes and their possible congruences along the viewport axes. Purchase [17] appeals to a holistic approach that not only cares for a plain contour congruence interpretation based on the its reflection around the classic cartesian axes but also for the rotational symmetry around potential axes.

Purchase [17] suggests an algorithm that takes as an input a drawing. From this input, the algorithm determines new axes (related to the node positions), determining if there is 'enough symmetry' to infer the existence of a subgraph, calculate a symmetry value for each of the subgraphs and finally doing an aggregative operation that normalizes the overall metric to the interval $0 \leq \aleph_s \leq 1$.

Evidently, the motivation supporting the approach is intuitively valid, but its implementation is cumbersome. Computing $\aleph_s$, as Purchase [17] admits, requires the implementation of algorithms having best case execution complexities on the order of $O(n^5)$ and $O(n^7)$ at worst.

Edge orthogonality This metric, $\aleph_{eo}$, deals with the angular deviation between the edges and an imaginary cartesian grid. Since each edge has an angle with relation to the horizontal axis, the metric is calculated as the average deviation of these angles. As an instance, edges that parallel to the horizontal axis do not contribute to the average deviation. As this deviation is subtracted from 1, a theoretical graph in which the edges are collinear with any line parallel to the axis must have the perfect edge orthogonality metric, namely $\aleph_{eo} = 1$.

Node orthogonality The principle of alignment to a grid is also involved in the node orthogonality metric definition $\aleph_{no}$. An assumptive cartesian grid layer is aligned upon the original drawing plane. The size of the grid's cells is tuned to the position of the nodes in the drawing and its bounding box fits the drawing area. Then, the metric's calculation takes quotient between the number of

nodes and the bounding box.

Upward flow The flow metric, $\aleph_f$, measures the edges overall directional consistency. Edges of a drawing containing segments whose direction alternates are penalized; edges that flow evenly do not alter the visual path meant by the edge's arrow. Undirected graphs are not in the consideration for this metric.

2.3 Drawing graphs: the algorithmic perspective

As a problem of algorithm design, graph rendering approaches are classified according to the priority given to the relevant aesthetics[3]. Di Battista et al. [14] categorize the methods using the following convention:

Topology-Shape-Metrics Approach A graph has three properties: the *Topology*, *Shape* and *Metrics*; the processing of each property takes a sequence of definite steps. These properties induce equivalence classes on the possible drawings of a graph. The *Topology* relation considers two drawings as equivalent if there exist a continuous transformation between them. The *Shape* relation is a stronger version of the *Topology* relation enforcing the transformation just on the edge segment length. Finally, the *Metrics* relation is a plain congruence up to translation and rotation [14].

Hierarchical Approach It is another step-wise method that processes graph's vertices by classifying them in layers depending on the direction of the edges. Then, further processing is done aiming to optimize the aesthetics. Good examples of this approach are the *Sugiyama's method* [25] and the *Topological Feature-Based Layout* [26].

Visibility Approach Prioritizes crossing reduction and tries to render edge drawing as a polygon chain.

Augmentation Approach Graph's building method operate by adding one node at a time.

Force Directed Approach Eades [27] and Dwyer [28] give a physical interpretation[4] to the shapes on a model, in consequence, the rendering becomes a problem of solving the equations of a physical system.

The aforementioned techniques rely heavily on the manipulation of custom data structures and the algorithmic knowledge about *graph planarization* and many other algorithms solving classic graph problems.

Another important feature is that the techniques are applied in a stage-wise fashion. This feature, described in more detail at Section 2.4.1.1, is relevant to the implementation of graph drawing frameworks.

[3] "number of bends", "symmetry", etc. See Section 2.2. [4] i.e. One of the classic examples of this approach is called the *Spring Layout Algorithm*.

2.3.1 Layout strategies

As this project deals with strategies to improve UML drawings, the suggestion is to use either the Sugiyama strategy or the Orthogonal strategy. This suggestion has its basis in the recommendations given by the literature surveyed at Section 2.2.1; a brief enumeration of those recommendations, in Galapov and Nikiforova [10] is:

Perceptual organization Location and geometric relation of the perceived objects between each other.

Perceptual segregation Being able to determine the difference between the shapes and their background.

Other guidelines Bend minimization, overlapping avoidance, enforce shape proximity etc.

The Sugiyama strategy and the Orthogonal strategy are natural choices since they enforce some of the recommendations within their algorithmic limitations.
The following subsections intend to introduce both strategies and other related concepts.

2.3.1.1 Sugiyama strategy

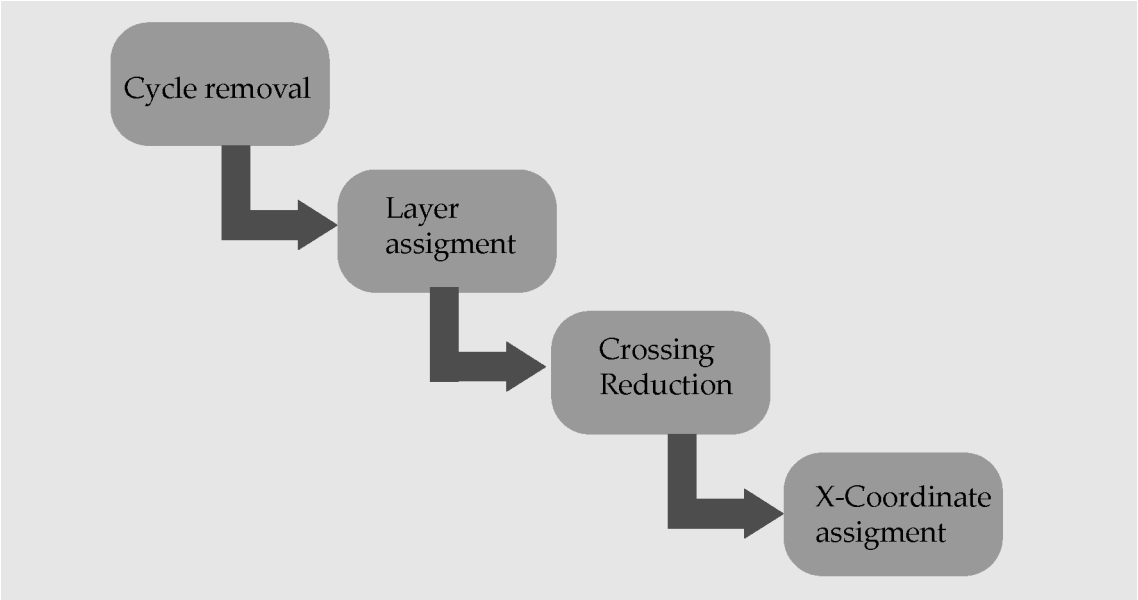


Figure 2.3: Sugiyama strategy process

The Figure 2.3 shows the waterfall structure of the steps involved in the Sugiyama strategy. The Sugiyama strategy prioritizes the following criteria [18, 29]:

- Even node distribution on the viewport.
- Edge flow uniformity.
- Prioritize straight edges.

- Short edge length.
- Minimize edge crossings.

This strategy belongs to a class of layout methods whose main characteristic is to separate the viewport into stripes or layers. The strategy divides the main algorithm goal into the following strictly sequential stages:

1. Cycle removal
2. Layer assignment
3. Crossing reduction
4. X-Coordinate assignment

Cycle removal This step is a preprocessing phase. All the cycles in the graph are removed by reversing as few edges as possible so that the edges point in one direction. The resulting graph directed and free of cycles. Particular cases such as cycles involving just two vertices are noticed at this step and taken care of in later stages.
The phase does not destroy edge directionality information; the cycle removal step preserves it. Later stages will use the stored information to reconstruct the directionality once the pipeline has finished processing the input.
At the completion of this phase, the input for the Layer Assignment step is ready for the consumption of the layer assignment stage.

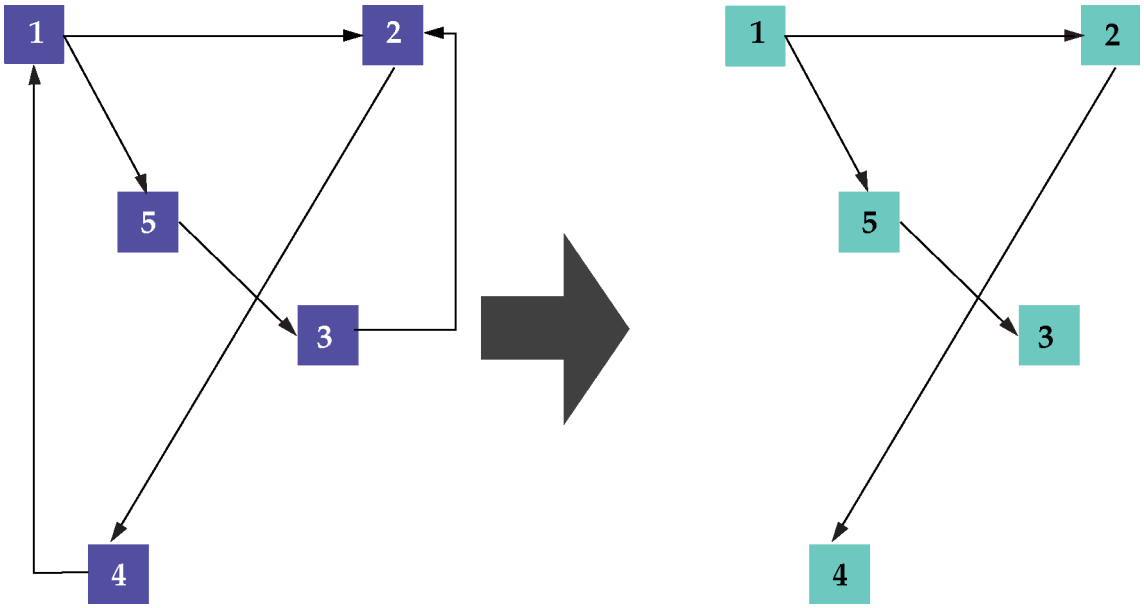


Figure 2.4: Cycle removal stage

Layer assignment A layering process defines the viewport as a collection of adjacent horizontal layers (also known as levels.) Then, the goal is to partition the vertex set by assigning subsets of its elements into layers. The condition inducing the layer partitioning is that *two vertices belonging to the same layer cannot be neighbors.*

The partitioning must also enforce the directionality between the nodes from different layers. Nodes belonging to upper layers must have edges pointing toward nodes in lower layers, causing the nodes to point downwards.
Some algorithms in further stages might require that the edges do not traverse more than one layer. A *proper layering* is a layering in which all edges cross only one layer. Achieving a proper layering may require adding dummy nodes.

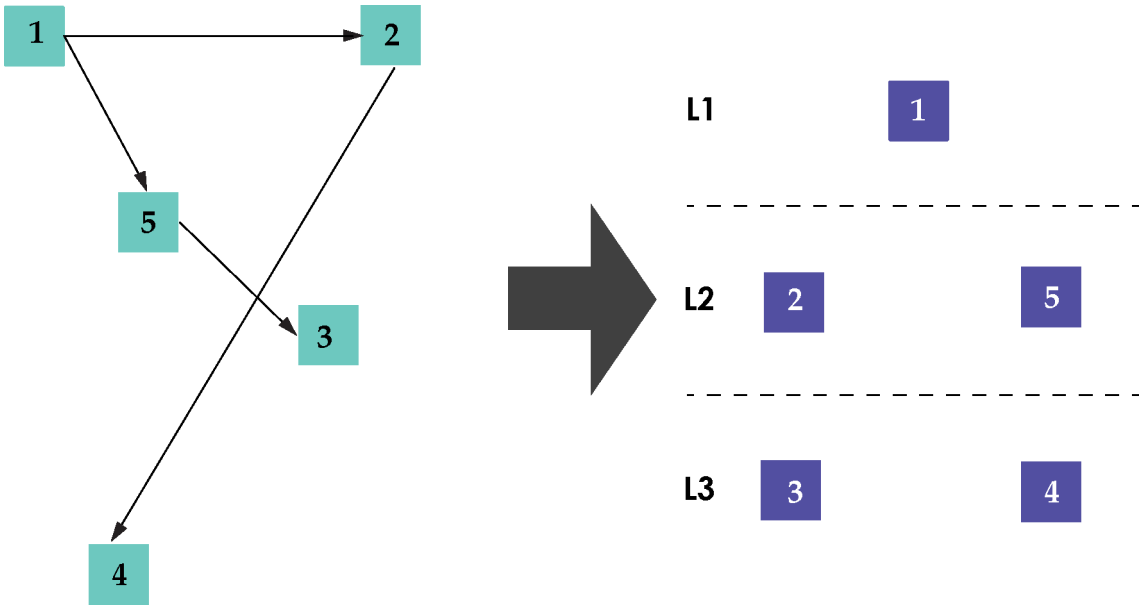


Figure 2.5: Layer assignment stage

Crossing reduction After a successful layer assignment, the vertices within each layer are ordered focusing on reducing the number of edge crossings.
On a first glance, it seems fruitful to sweep each layer hoping to reorder the vertices having in mind the crossing reduction goals. A Layer-by-Layer sweep consists of fixing the ordering of the nodes of a layer while rearranging nodes on other layers until an acceptable threshold for number of crosses if found. This approach, as mentioned by Kaufmann and Wagner [18] is not optimal; hence the technique is to iterate the process many times while picking a random layer on each opportunity.
Notice that the random layer selection technique for each note yields an indeterministic result. OGDF's developers suggest forcing a deterministic behavior by setting up a fixed seed for the random number generator before each call[5].

X-Coordinate assignment Assigning the horizontal coordinates to each vertex must obey the premise of bend minimization. The fake nodes introduced in previous phases appear in this state as bends. A simple heuristic for node positioning is to infer the positions from the information given by the crossing minimization step.

[5] See `http://www.ogdf.net/doku.php/ogdf:faq`

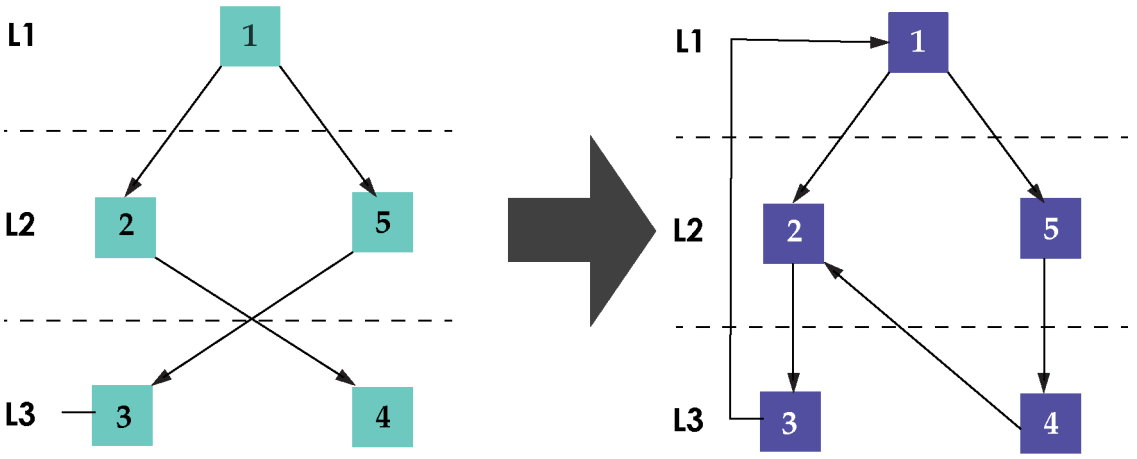


Figure 2.6: Crossing reduction stage

2.3.1.2 Heuristics in cross reduction

Barycenter heuristic Originally proposed by Sugiyama [25], this cross minimization heuristic is based on the assumption of the proportionality between node adjacency and number of crosses (i.e. if two nodes are near, their potential crosses are reduced.) After measuring this metric on each of the vertices of a layer, its corresponding partition is sorting using the barycenter as primary criteria. This method generates drawings free of crossings if the original graph structure permits that possibility.

Median heuristic The innovation from Eades and Wormald [30] resides in suggesting a sort order on the layers that depends on a particular definition of the median x-coordinate of each vertex about the x-coordinates of its neighbors.

2.3.1.3 Orthogonal strategy

A prevalent recommendation for layouts involving UML models (surveyed in detail on Section 1.2.1.2) is to render the shapes into an orthogonal grid. This rendering is called *an embedding* and motivates the following definition:

Definition 12 (Orthogonal Embedding [18]) *An orthogonal grid embedding* Γ *of a graph* $G = (V, E)$ *is a map between* $v \in V$ *and integer grid points* $\Gamma(v)$ *on the plane. The grid embedding also maps edges into sequences of non-overlapping paths on the plane.*

The Figure 2.7 shows an *embedding* (e.g., set up every node in a crossing of an imaginary grid and set the edges along the imaginary lines of such grid) of the edges and nodes into a grid. This is the main idea behind the orthogonal strategy.

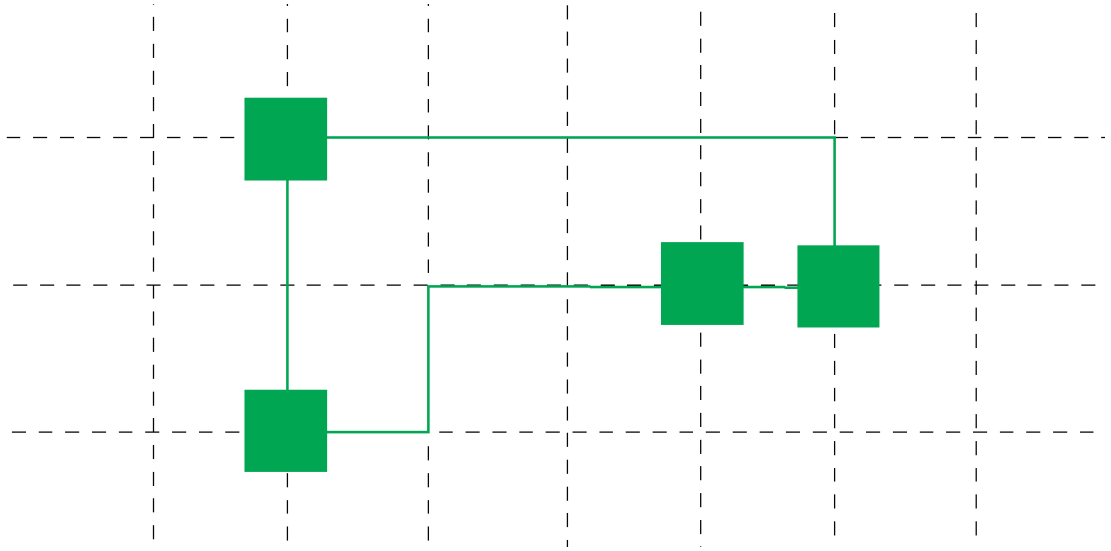


Figure 2.7: Orthogonal shapes distribution

2.4 Graph Drawing software tools in this project

2.4.1 Open Graph Drawing Framework

The OGDF is a library containing reusable data structures and graph algorithms. Its development did not start from scratch on account of Algorithms for Graph Drawing (AGD)'s (Algorithms for Graph Drawing, a project of the Max-Planck Institute [31]) legacy source being its precedent codebase.
The evolution of the graph drawing theory has as an outcome a theoretical framework that characterizes both graphs and algorithms in a taxonomy. Di Battista et al. [14] mentions as a justification for that taxonomy two observations:

- Graphs can be classified (e.g.,ãssigning directions to an undirected graph generates a directed graph); and those classes are subject to a natural hierarchization. Hence, some of the algorithms applicable to graphs of a class can also be useful for similar problems.
- It is common to discretize a graph drawing methodology as a pipeline of steps. A direct consequence is a possibility of analyzing each of the steps independently.

These considerations influenced OGDF's architecture, and are adopted well beyond into the mainstream design trend as it is noticeable in other frameworks.

2.4.1.1 OGDF's Graph drawing functionality, infrastructure and implementation

OGDF's main contributions belong to the following categories [32]:

Basic Data Structures Lists, hash tables, etc. implementations focused on the project's requirements.

Graph representation support The class `Graph`, `CombinatorialEmbedding`, `PlanRep` (for planar representations) etc.

Layout Algorithms Implementations for the `PlanarizationLayout`, `ModularMultilevelMixer` etc.

Modules Planarity-based algorithms require the modules `PlanarSubgraphModule`, `EdgeInsertionModule` among other implementations.

These contributions are implemented in a modular fashion, following the guidelines from the aforementioned observations [29]. As a result, the framework's modularity allows testing new algorithmic approaches by replacing some of the stages in the classical graph drawing pipelines.
As a final remark about OGDF's implementation, a definite design goal is to make the library as self-contained as possible. Except for the linear programming solver (COIN-OR [33]) and the branch-and-cut framework (ABACUS [34]), the library lacks on external dependencies.

2.4.1.2 Other graph software libraries and open formats

It is almost certain that, during the past 20 years, the Graph Drawing community has noticed tacitly the benefits of incorporating all the research experience into reusable software artifacts. This conjecture becomes substantiated by the abundant assortment of libraries and open formats. In fact, some of the libraries have been adopted by the community and grow further along with its necessities.
The main benefit the existence of these libraries provide is that most of the low-level issues have been solved and proven during the last years. Libraries like `TULIP` and OGDF have already got to the point of being platforms allowing experimentation with new layout approaches since they have been designed using a plug-in architecture. Evidently, reaching this level of maturity requires having gone through the development of stable data structures and other well-defined solutions that can be taken as standards for the field.

Graph modeling languages and image formats Side to side with the proliferation of libraries, there is also an extensive variety of languages whose purpose is to standardize graph information interchange.
Libraries can interact using the traditional formats (i.e. Comma Separated Value files) or more specialized formats like Graph Modeling/Markup Language (GML) (one of the many standards supported by `TULIP`), `dot` (`Graphviz`), Graph Exchange XML Format (GEXG)[6] or NET/PAJ (`Pajek`). It is common in these frameworks to allow image information exporting to standards like Portable Network Graphics (PNG), Joint Photographic Experts Group (JPEG), and SVG.
A remarkable example of a successful graph drawing is `TULIP`. This library is a 20-year-old information visualization framework offering techniques and tools to solve domain specific problems. It has a `Python` layer and `C++` Application Programming Interface (API) providing tools for the development of interactive widgets, teaching tasks and other graph related activities [35].

[6] `Gephi`'s format

TULIP has an architecture supporting extensibility by the use of plugins. The plugin architectural design is particularly useful for the free replacement of algorithms by the enforcement of the framework's interfaces; hence, that architecture is a traditional development pattern used among many other libraries. TULIP's modularity is also resourceful to import and to export information using multiple graphs and image formats.

The most well-known libraries given their historical value and their current popularity are "AGD-library" [36] and "Graphviz." [35, 37] Other important libraries worthy of being mentioned are gdLibrary [38, 39] and Pajek. [18]

Along with the open source community, private efforts have grown into patents like the Microsoft Automatic Graph Layout (MSAGL) [40] and Graph Drawing Toolkit (GDT). [18]

There have also been efforts toward drawing UML; in particular, class diagrams. One open sourced popular package for such ends is PlantUML [41]. This software can interpret a text encoding of an UML diagram (understandable by a human) and draw the corresponding graphical representation.

On the adoption of OGDF Considering that this is a project for a for-profit company, it is necessary to choose a library whose code can be freely reused. It is relevant to mention that the software products linking to the library are part of an internal software modeling tool (e.g., it is not for the use of external clients.) Therefore, there is no obstacle to its use concerning its GNU General Public License v2.0 (GPLv2)[7] license.

From the viewpoint of the development, the selected library has to have some of the previously mentioned characteristics regarding maturity[8], and the input/output formats should be as standard as possible.

Originally, the chosen library for the task at hand was TULIP. This library allows to build complete graphical interfaces and provides out-of-the-box functionality covering the requirements for this project. In particular, TULIP allowed the development of shape templates (i.e. generating a reusable primitive mold for a state); hence, the framework can manage the geometric shape data seamlessly. It also provides further functionality to embrace the graph exportation into different formats, a typical business scenario at Ericsson.

Unfortunately, TULIP's development philosophy does not give priority to component self-containment. For instance, compiling the library required references to outdated versions of the Qt framework[9]; that made OGDF a better option even though it does not provide as much out-of-the-box functionality as TULIP does.

[7] From the GPLv2, one of the OGDF licensing schemes (besides GNU General Public License v3.0 (GPLv3)) referenced in the project's code repository: "Activities other than copying, distribution and modification are not covered by this License; they are outside its scope." [8] Most of the classic algorithms and primitive data structures should not be implemented since the scope of this project is not to develop what has been already built and tested. [9] Provided by the Qt Company [42]

2.4.2 SVG object manipulation

The aesthetic metric computation requires knowledge about the geometric characteristics of the shapes inside an SVG; therefore, it is necessary to extract the raw geometric data from its XML structure.
Undertaking the geometric data scraping calls for the use of two libraries focused on SVG specifics:

`svgpathtools` Includes a complete set of functions to operate on SVG path objects, general parsing for SVG shape properties (e.g.,k̃inks along a path), analysis of line differentiability in a real domain, conversion of a form into its equivalent path, path smoothing, intersection detection, path representation and bezier curves among other functionalities Port [43]. This library uses a `Python` package as its main distribution channel.

`svgo` Offers a functionality focused on SVG file structure simplification and optimization. Merging different paths sharing the same endpoints is an example of the possible simplifications an SVG can be subject [44]. The library requires `nodejs` [45] as it is written to be interpreted by a JavaScript runtime.

`NetworkX` As it is necessary to operate on graphs in late stages of layout's analysis, `NetworkX` is the `Python` solution allowing graph representation and some Input/Output related activities. Its description, as summarized at Hagberg [46], is:

> `NetworkX` is a `Python` package for the creation, manipulation, and study of the structure, dynamics, and functions of complex networks.

The specifics regarding SVG manipulation in the library will be reviewed carefully in Section 3.1.1.

3
Methods

Introducing new strategies adds the concern of how to benchmark the present output generated by the toolchain in comparison with the results provided by the new implementation.
For the sake of making evident the changes brought by the observation, it became a necessity to extract geometric information from the original SVG files created by the toolchain.
The following sections explore, in an approximately sequential fashion, the development of the proof of concept. As it is a requirement to extract the geometric data, the chapter starts by motivating and describing Graph Usability Benchmarking Tool (GUBT). Then, YAGT comes naturally as the second step towards generating a new layout.

3.1 Extracting data from the current SVG files

As it was mentioned on Section 1.1.1, after the execution of the `Python` scripts, the toolchain produced SVG files. Given that the SVG standard prescribes rules regarding the geometric disposition of images in a viewport, it was natural to obtain the data to feed the metrics after processing these files as input.
The accomplishment of the data extraction goal suggested the development of single software tool capable metrics gathering: GUBT. Along with this tool, testing new graph layouts was a call for the construction of a different mechanism feeding on the geometric output from GUBT and producing streamlined blueprints encoded as geometric data. YAGT's intention is to embody that goal.

3.1.1 The Graph Usability Benchmarking Tool

The gist of the metric analysis for the SVGs resides in the possibility of extracting geometric data from the SVG content. Its primary task is to retrieve the bounding boxes of every state and, more generally, every `<g/>` group that could be singled out as belonging to the state machine. The second function is to distinguish the paths forming the edges and the relationships connecting the nodes based on the specifics of the SVG structure.
Analyzing each SVG file independently required the implementation of a command-line tool to filter out xml fragments representing the graphic entities. As it can be seen from the fragments describing the entities (e.g., 3.2), the toolchain groups the graphic elements using the `<g/>` tag provided by the SVG specification. This tag

allows to operate on the discrete elements it groups both for the purpose of changing their geometric properties or to allow the identification of the group as a whole. Generating the small details seen on the shapes can be done by breaking the complexity of the path into simpler subpaths that are visually undetected. (e.g., the round corners of the `WaitForResponse` state in figure 3.1.1)

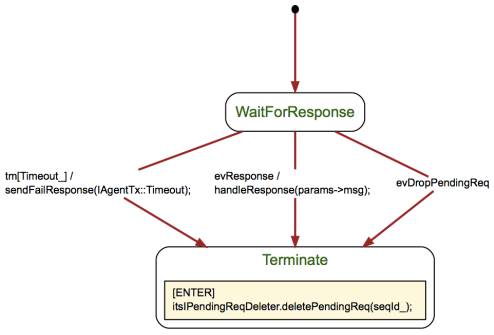


Figure 3.1: Here, the `Start` state (represented as a small dot at the top) has the same XML structure as the other two states. The graph's edges are a composition of two unconnected SVG paths. Label's depiction requires to draw them as nodes whose boundaries are transparent lines.

Listing 3.1: XML description for the edge going from the `Start` state into the `WaitForResponse` state

```
<!-- StartState&#45;&gt;WaitForResponse -->
<g id="edge3" class="edge">
  <title>StartState-&gt;WaitForResponse</title>
  <path fill="none" stroke="firebrick" stroke-width="2"
    d="M271-302.354C271,-291.959 271,-261.433 271,-238.298"/>
  <polygon fill="firebrick" stroke="firebrick"
    points="274.5,-238.286271,-228.286 267.5, -238.286 274.5,-238.286"/>
</g>
```

Listing 3.2: XML description for the shape of the state `WaitForResponse`

```
<!-- WaitForResponse -->
<g id="Agent::PendingReq.WaitForResponse" class="node">
  <title>WaitForResponse</title>
  <a xlink:href="#81:4-18" xlink:title="WaitForResponse">
    <polyline fill="none" stroke="black" points="324,-228 218,-228 "/>
    <path fill="none" stroke="black" d="M218,-228C212,-228 206,-222 206-216"/>
    <polyline fill="none" stroke="black" points="206,-216 206,-204 "/>
    <path fill="none" stroke="black" d="M206,-204C206,-198 212,-192 218-192"/>
    <polyline fill="none" stroke="black" points="218,-192 324,-192 "/>
    <path fill="none" stroke="black" d="M324,-192C330,-192 336,-198 336-204"/>
    <polyline fill="none" stroke="black" points="336,-204 336,-216 "/>
    <path fill="none" stroke="black" d="M336,-216C336,-222 330,-228 324-228"/>
    <text text-anchor="middle" x="271" y="-205.4"
      font-family="Arial"font-size="14.00" fill="darkgreen">WaitForResponse</text>
  </a>
</g>
```

3.1.1.1 Extracting meaningful entities from the SVG

Determining the entities is an XML scraping task involving the parsing of the SVG. Entity data was stored inside dictionaries whose key was the content of the title element inside their groups.

Extracting States There are as many different structures for the string as possible shapes for the states. As it was seen at figure 3.1.1, there are 'start states', 'normal states' etc.

Filtering the SVG for states required to choose the entities that did not contain the substrings `->`, `_To_` or `Sm` and to explicitly take those containing the substring `cluster`.
Usually, the shapes are complex, and they can be classified depending on the visual function they fulfill. Standard states in a machine are a good example of the naming complexity as per the shape is built from two different groups. One of them is the external group of the paths forming the boundary. The other group is the inner box representing the internal content of the shape.
After the shapes are extracted, the next step is to infer their bounding boxes and associate that information to the shape's data.
The groups forming the boundary line are a collection of disjoint paths. Overcoming this obstacle requires to extract the bounding boxes of each path using `svgpathtools` and then finding the most extreme points representing opposite corners of the shape's bounding box.

Extracting Transitions As with the state extraction, filtering out the transitions is challenging. Visually, the transitions between states are shown as paths that are suddenly broken by the transition's label.
The toolchain has a limited functionality regarding transition's labeling. Rendering the transition's label consists of making a customized node with transparent boundaries including other exclusive features and then interpolating it between two paths that connect the label to the real states.

3.1.2 Yet Another Graph Tool

The purpose of YAGT is to enable the experimentation with different strategies upon the data obtained from the layouts extracted by GUBT.

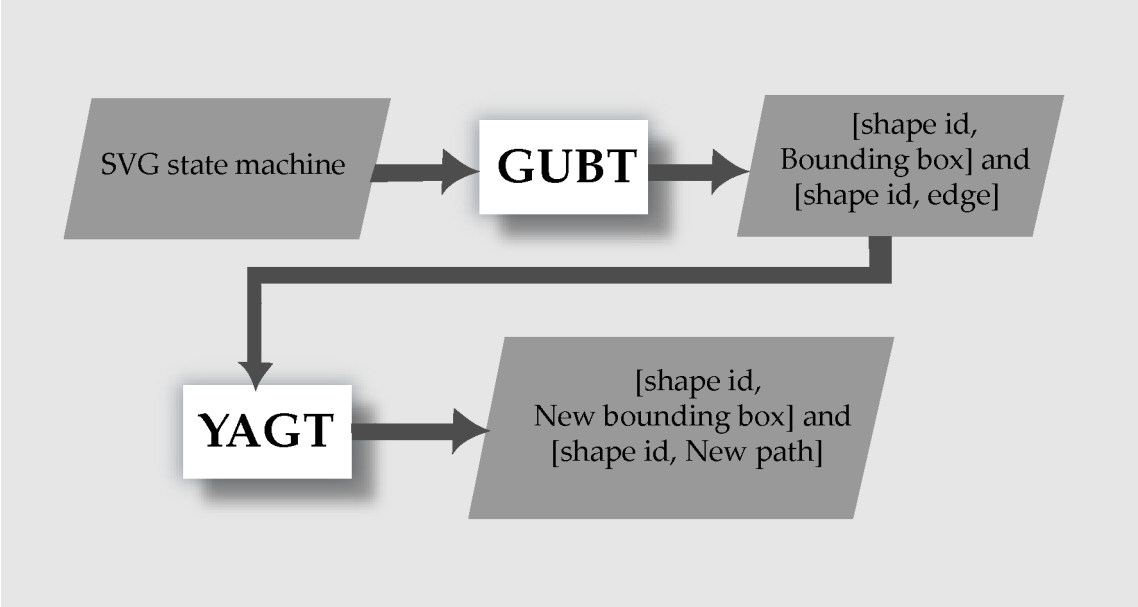


Figure 3.2: YAGT workflow

As shown in figure 3.2, YAGT's construction is a composition of pipelined steps written in `Python` whose purpose is to orchestrate the flow of information coming from GUBT into a layout engine.

3.1.2.1 The pipeline module

Shape extraction The objective of this step is to extract the geometric information of the bounding boxes contained in an SVG. Another output from this step is a data structure containing the graphs' node relationship information (i.e., node predecessors and successors, etc.)
This step makes extensive use of `NetworkX` library, as it builds a graph in its default data structure. `NetworkX`'s graph data structure can generate GML structured text files. As expected, those files will contain identifiers for the nodes, the edges, the edge relationships between the nodes and the state's bounding boxes that can be recognized as such.

Prepare engine's input data The graph data is not yet ready for the consumption of the engine. As the engine depends on OGDF's GML reading capabilities, the data requires to be enriched with extra information.
Also, OGDF's GML writer overlook the node identifier information from the input. This causes shape identifier information loss after the library has done successful reads and writes.
As it is required to be able to recover the identity of each shape after its layout has been modified, preparing the data has the responsibility of marking each GML entity with the identifier of the shape as it comes from the SVG.

Use the engine The engine accepts a GML input together with a switch that allows to choose between the Sugiyama or the Orthogonal strategy. From the input, it generates a GML output as well as an SVG representation of that output.
The GML output contains the same bounding boxes as the input had, but the coordinates are now updated. It also includes the edge relationship enriched with the coordinates of the points belonging the polyline created after the strategy's application.

3.1.2.2 The layout engine module

The layout engine module is a command line program written in `CPP` that feeds from a GML file and, depending on the value of the strategy switch, generates either an orthogonal layout or a Sugiyama arrangement.
The program uses OGDF library extensively. It reads the GML representation into an OGDF data structure and then proceeds to apply the strategies depending on their customized parameters. An undocumented undesirable characteristic of the library's Input/Output system is that it does not have a uniform criteria regarding the information that must come inside the input GML. This implies that using different strategies require the manual tuning of the common data structures even thought there is not real reason supporting the need.

3.2 Strategy implementation

3.2.1 Sugiyama strategy implementation

Applying the Sugiyama layout is easy since it is ready for instantiation just after the source code has a reference to OGDF. Yet, the algorithm's execution is open for further tailoring (and even adding improvements) as the object exposes the following parameters for modification:

Predefined ranking It is possible to assign fixed layers for each node. This comes in handy when the requirement wants to have the control over the shape's vertical locations in the viewport.

Number of cross minimization runs As mentioned on the discussion about cross minimization (2.3.1.1), OGDF makes possible to try different assignments until the amount of crosses reach an optimal bound. The tradeoff for having this characteristic is that the layout becomes nondeterministic.

Cross minimization heuristics As per modular design, the heuristics can be instantiated and assigned to the strategy's object prior to the layout generation

Layout setup The subclasses of `HierarchyLayoutModule`[1] are suitable options to set up the strategy layout.

3.2.2 Orthogonal strategy implementation

The orthogonal strategy implementation involves setting up a planarization layout and then allows to customize the node embedding strategy.
The details concerning the geometric distribution of the objects in the grid are established by setting up the parameters of the Orthogonal Layout object. This object is then assigned as the chosen layout mechanism on the Orthogonal Strategy object.

[1] e.g., `OptimalHierarchyLayout` or `FastHierarchyLayout`

4

Results

For the purposes of evaluating the results, this chapter presents an example of YAGT on a simple state machine: the `PdsClient`.

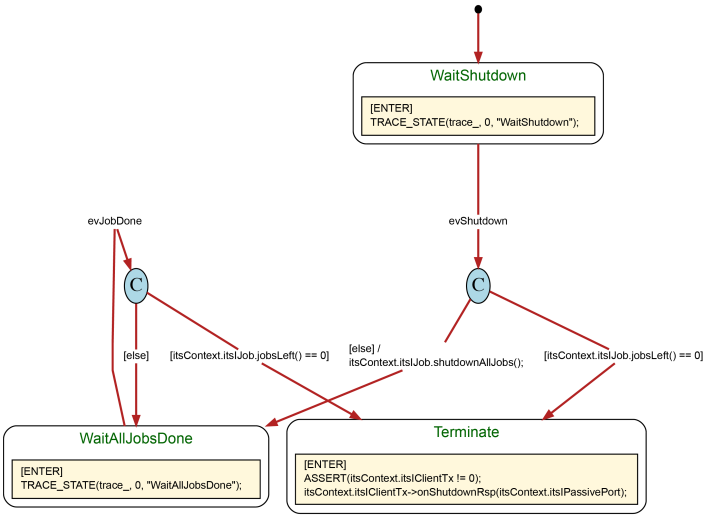


Figure 4.1: PdsClient state machine.

Each one of the following experiments tries different parameters for each given strategy. Notice that the strategies also produce a placing for the edges. Although labeling is also a problem contributing to the overall graph usability, this work assigns a higher priority for the shapes and edges distribution.

4.1 Sugiyama strategy outcomes

OGDF's Sugiyama strategy implementation has the interesting property of allowing to set up the node ranking. As it sequentially processes the nodes, it assigns user predefined layers to them. This property requires further investigation since it is not clear from OGDF's documentation if the node sequence is nondeterministic for sequential executions of the algorithm.
The experiments will show the application of two heuristics for cross minimization on an unranked input. The layer distance is fixed but slightly modified given because the lack of edge orthogonality may force a edge segment to be rendered below an state.

4.1.1 Experiment 1: Median heuristic

This result avoids the crossings. It is noticeable that the ordering of the nodes does not respect the intention conveyed in the original drawing.

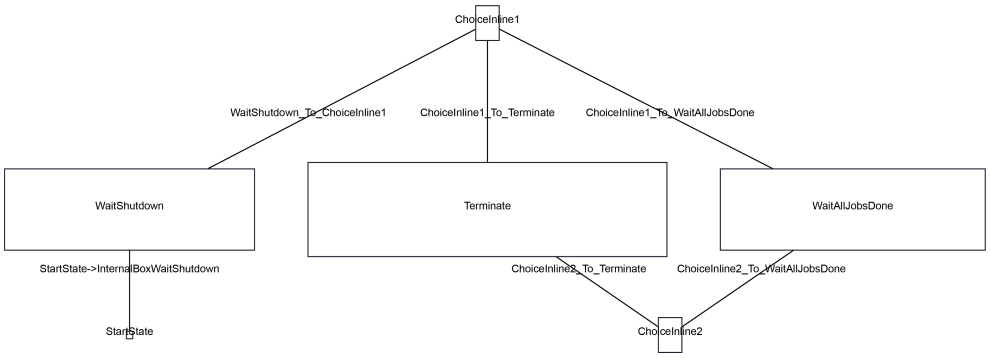


Figure 4.2: Experiment 1 - Median Heuristic

4.1.2 Experiment 2: Barycenter Heuristic

Again, the result avoids the crossings. The ordering, however, breaks the intended direction flow.

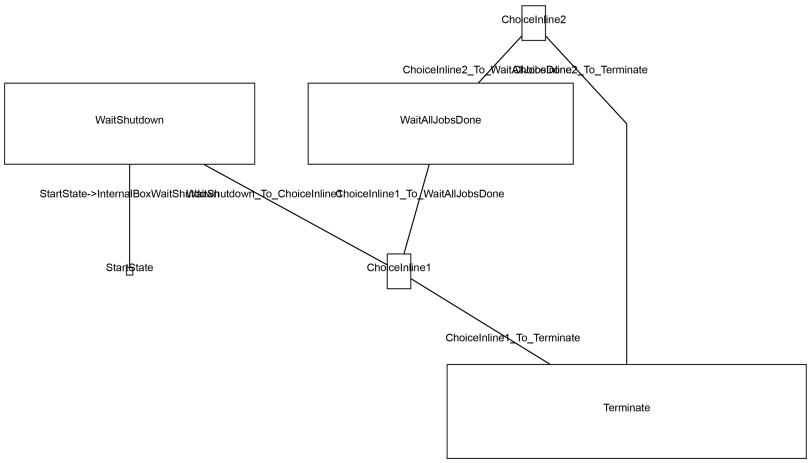


Figure 4.3: Experiment 2 - Barycenter Heuristic

4.1.3 Experiment 3: Barycenter heuristic plus node ranking

Layer distance has been slightly increased.

Ranking option 1 (fig. 4.4)
The sequential definition of the ranking is $[0, 0, 1, 1, 2, 2]$. This ranking defines three layers and two nodes per layer.

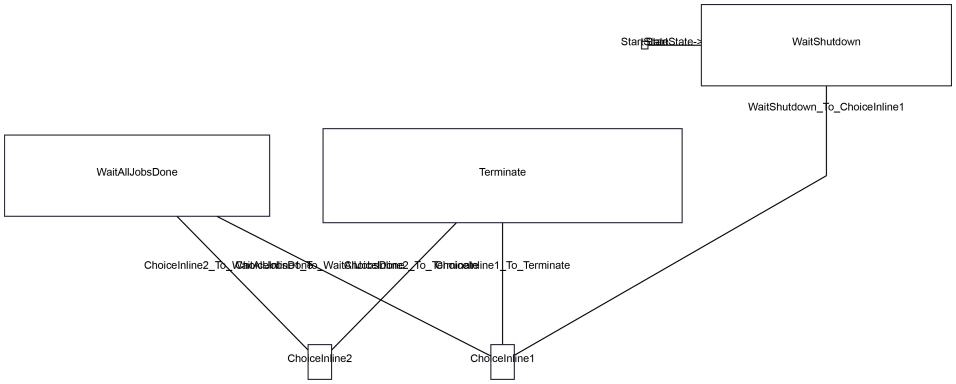


Figure 4.4: Experiment 3 - First ranking

Ranking option 2 (fig. 4.5)
The sequential definition of the ranking is $[2, 2, 1, 1, 0, 0]$. This ranking defines three layers and two nodes per layer but gives a different arrangement for each layer.

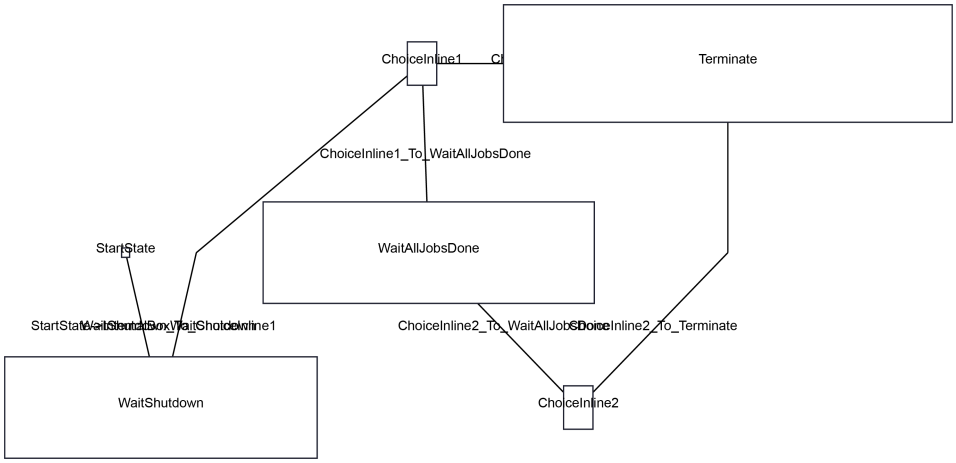


Figure 4.5: Experiment 3 - Second ranking

Ranking option 3 (fig. 4.6) The sequential definition of the ranking is $[1, 2, 1, 2, 1, 2]$. This ranking defines two layers and three nodes per layer.

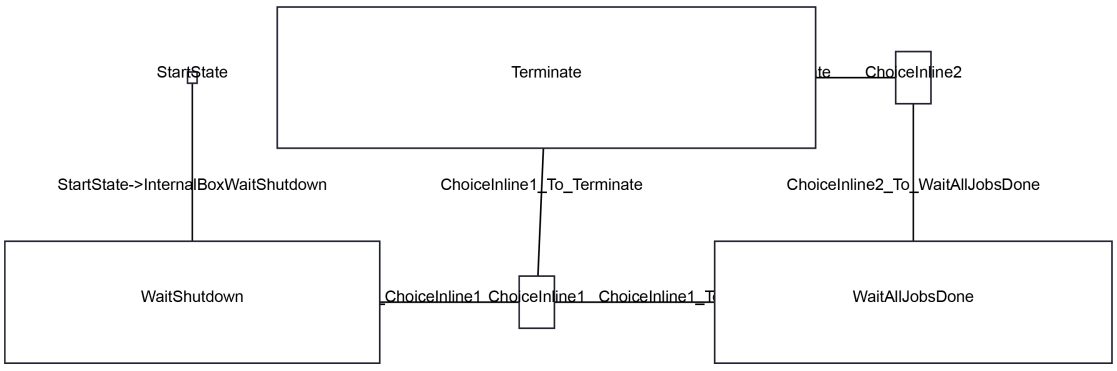


Figure 4.6: Experiment 3 - Third ranking

Comments The graph usability improves greatly after forcing a ranking order on the nodes.

4.2 Orthogonal strategy outcome

For the orthogonal layout results the parameters are:

- Maximum bound on admitted bents: 10.
- Margin: 10px.
- The edges are not allowed to scale (change their length.)
- The separation between shapes is set to 50px.

This approach is promising as it respects the directionality of the graph. The lack of control on node ordering might affect negatively larger graph inputs.

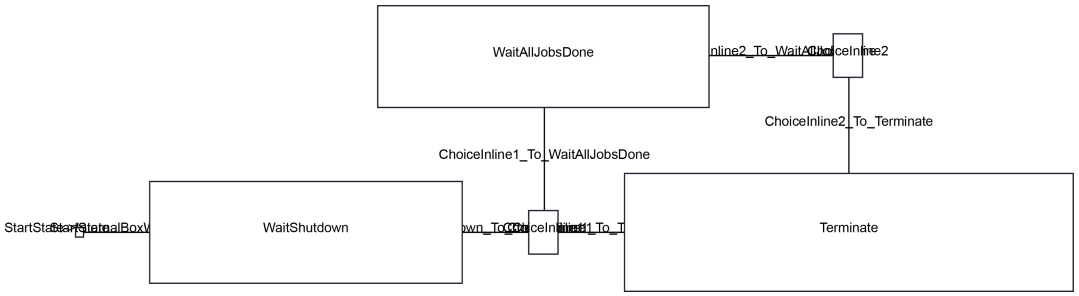


Figure 4.7: Experiment 3 - Orthogonal example

4.3 Failed experiments

The library made acceptable outputs for small input graphs like the one on the figure 4.1. However, larger state machines made evident unexpected instabilities in OGDF's implementation of the strategies.
Mainly, the failed outcomes can be classified as:

1. The algorithm spends ever increasing amounts of memory.
2. The output is worst than the input, or it is completely scrambled.
3. The algorithm does not stop in a reasonable execution time.

The first and third failures are related to unchecked stack overflows. It could be also the case that the algorithm implementation do not take into account some edge cases that might be present on the input. It might be also possible that the input, the GML file, has not been correctly extracted.
An experimental setup must be created to systematically debug the library.

5
Discussion

This report covered basic concepts of graph theory, and, upon them, it surveyed notions of graph aesthetics concerning UML diagrams. The report, following along the primary goals, showed the driving ideas behind the implementation to address the intended goal. Finally, this section expects to make obvious remarkable issues, suggest guidelines and propose possible new relevant tasks.

5.1 Remarks

As an overall observation, the methods are promising. However, designing a new strategy, or even tuning an existing one, requires careful experimentation. Also, OGDF is a promising framework, but it is necessary to take into estimation the time and effort that must be invested to make it production ready or at least to limit the bound of critical undetected bugs.
Open frameworks ease most of the already mastered knowledge in the graph drawing field as well as the SVG geometric manipulation. That maturity is an asset whose utility is more than optimal for solving specific problems like the one addressed in this work.

5.2 Suggestions

The main suggestion is to replace the `Graphviz` engine for an in-house solution. This work shows that OGDF is mature enough to be embedded in the toolchain provided that its architecture does not generate undesired couplings. Also, being OGDF an open source library complying with minimum code readability standards, makes it a good candidate to be further improved.
Another recommendation is to avoid delegating the layout responsibilities too early in the process of converting the ESC files into SVGs. Instead of using the `dot` language to feed `Graphviz`, the geometric data coming from the primitive (and pre-computed) SVG shapes in the input can be feed to an OGDF layout engine. After the engine produces the output, the final step is to merge the geometric data and the pre-computed information directly on the final SVG output.

5.3 Future work

The possibilities lying ahead will probably relate to the following problems:

Merging the results As mentioned on the suggestions, external geometric layout providers should be integrated in the solution either by pushing their contents in the final SVGs or by providing the layouts during the SVG generation process.

Experiment with other strategies Most of the requirements for UML layout readability are not taken care of by default by the libraries mentioned in the theory. Therefore, a careful tailoring from the functionalities into the UML readability guidelines is the best option to improve readability

Edge labeling Edge labeling is a problem as complex as the graph layout problem. As YAGT shows, the output of a layout engine is suitable to further geometric enhancements. A proposal for edge labeling is, perhaps, to process the polylines from the output of YAGT and then decide where the labels should go. A theoretical pipeline would then add the pre-computed labels into the final SVG drawing.

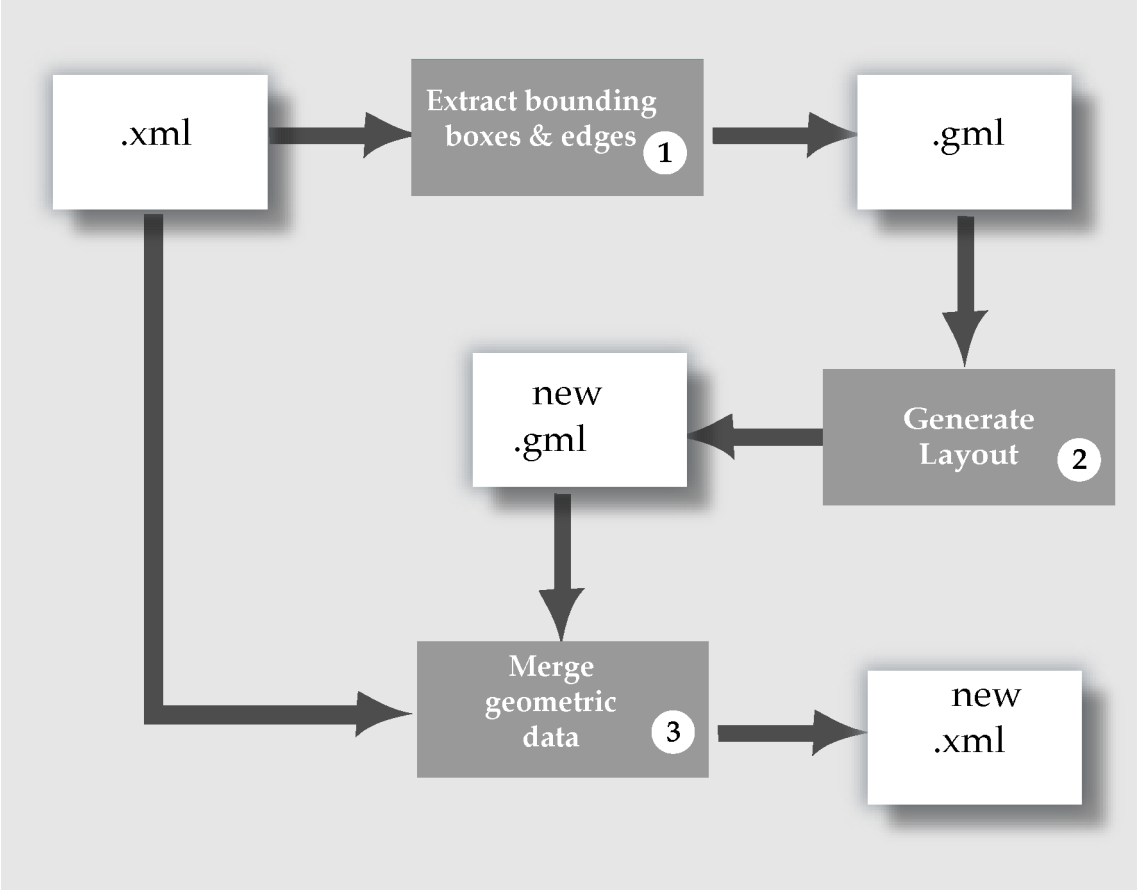


Figure 5.1: Complete pipeline

A possible complete pipeline could be implemented following the guidelines of figure 5.1.

Bibliography

[1] H. Sörensson and V. Mazetti, "Visualisation of state machines using the Sugiyama framework Master of Science Thesis in Computer Science," no. June, 2012.

[2] B. B. Bederson and B. Shneiderman, *The craft of information visualization: readings and reflections*. Morgan Kaufmann, 2003.

[3] R. Mazza, *Introduction to Information Visualization*. London: Springer London, 2009, vol. 1, no. 978-1-84800-218-0. [Online]. Available: http://link.springer.com/10.1007/978-1-84800-219-7

[4] I. Herman, G. Melancon, and M. Marshall, "Graph visualization and navigation in information visualization: A survey," *IEEE Transactions on Visualization and Computer Graphics*, vol. 6, no. 1, pp. 24–43, 2000. [Online]. Available: http://ieeexplore.ieee.org/document/841119/

[5] Ericsson, "The Easy StateChart Language," 2009.

[6] G. Booch, I. Jacobson, and J. Rumbaugh, "The Unified Modeling Language Reference Manual," 1999.

[7] H. C. Purchase, *Experimental Human-Computer Interaction*. Cambridge: Cambridge University Press, 2012. [Online]. Available: http://ebooks.cambridge.org/ref/id/CBO9780511844522

[8] K. Wong and D. Sun, "On evaluating the layout of UML diagrams for program comprehension," *Software Quality Journal*, vol. 14, no. 3, pp. 233–259, 2006.

[9] H. Störrle, "On the impact of layout quality to understanding UML diagrams: size matters," in *International Conference on Model Driven Engineering Languages and Systems*. Springer, 2014, pp. 518–534.

[10] A. Galapov and O. Nikiforova, "UML Diagram Layouting: the State of the Art," *Scientific Journal of Riga Technical University. Computer Sciences*, vol. 44, no. 1, jan 2011. [Online]. Available: http://www.degruyter.com/view/j/acss.2011.44.issue--1/v10143-011-0027-0/v10143-011-0027-0.xml

[11] K. Koffka, *Principles of Gestalt psychology*. Routledge, 2013, vol. 44.

[12] H. Fuhrmann and R. von Hanxleden, *On the pragmatics of Model-Based Design*, 1998. [Online]. Available: www.informatik.uni-kiel.de/rtsys/

[13] H. C. Purchase, J.-A. Allder, and D. Carrington, "User Preference of Graph Layout Aesthetics: A UML Study," in *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, 2001, pp. 5–18. [Online]. Available: http://link.springer.com/10.1007/3-540-44541-2{_}2

[14] G. Di Battista, P. Eades, I. G. Tollis, and R. Tamassia, *Graph drawing: algorithms for the visualization of graphs*, 1999.

[15] H. C. Purchase, "Metrics for graph drawing aesthetics," *Journal of Visual Languages & Computing*, vol. 13, no. 5, pp. 501–516, 2002.

[16] J. L. Gross and J. Yellen, *Graph theory and its applications*. CRC press, 2005.

[17] H. C. Purchase, "Which aesthetic has the greatest effect on human understanding?" *Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics)*, vol. 1353, pp. 248–261, 1997.

[18] M. Kaufmann and D. Wagner, *Drawing graphs: methods and models*. Springer, 2003, vol. 2025.

[19] M. Jünger and P. Mutzel, Eds., *Graph Drawing Software*, ser. Mathematics and Visualization. Berlin, Heidelberg: Springer Berlin Heidelberg, 2004. [Online]. Available: http://link.springer.com/10.1007/978-3-642-18638-7

[20] D. M. Spönemann, *Graph Layout Support for Model-Driven Engineering*, 2015.

[21] G. D. Battista, P. Eades, R. Tamassia, and I. G. Tollis, "Algorithms for drawing graphs: an annotated bibliography," *Computational Geometry*, vol. 4, no. 5, pp. 235–282, 1994.

[22] K. R. Boff, L. Kaufman, and J. P. Thomas, *Handbook of perception and human performance*. Wiley New York, 1986, vol. 2.

[23] H. C. Purchase, "A healthy critical attitude: Revisiting the results of a graph drawing study," *Journal of Graph Algorithms and Applications*, vol. 18, no. 2, pp. 281–311, 2014. [Online]. Available: http://jgaa.info/getPaper?id=323

[24] W. Huang, M. L. Huang, and C.-C. Lin, "Evaluating overall quality of graph visualizations based on aesthetics aggregation," *Information Sciences*, vol. 330, pp. 444 – 454, 2016, sI Visual Info Communication. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0020025515003874

[25] K. Sugiyama, S. Tagawa, and M. Toda, "Methods for visual understanding of hierarchical system structures," *IEEE Transactions on Systems, Man, and Cybernetics*, vol. 11, no. 2, pp. 109–125, 1981.

[26] D. Archambault, T. Munzner, and D. Auber, "Topolayout: Multilevel graph layout by topological features," *IEEE transactions on visualization and computer graphics*, vol. 13, no. 2, 2007.

[27] P. Eades, "A Heuristics for Graph Drawing," *Congressus Numerantium*, vol. 42, pp. 146–160, 1984. [Online]. Available: http://ci.nii.ac.jp/naid/10000075358/en/

[28] T. Dwyer, "Three dimensional UML using force directed layout," in *Proceedings of the 2001 Asia-Pacific symposium on Information visualisation-Volume 9*. Australian Computer Society, Inc., 2001, pp. 77–85.

[29] R. Tamassia, *Handbook of graph drawing and visualization*. CRC press, 2013.

[30] P. Eades and N. C. Wormald, “Edge crossings in drawings of bipartite graphs,” *Algorithmica*, vol. 11, no. 4, pp. 379–403, 1994.

[31] M. Chimani, C. Gutwenger, M. Jünger, G. W. Klau, K. Klein, and P. Mutzel, “The Open Graph Drawing Framework (OGDF).” *Handbook of Graph Drawing and Visualization*, vol. 2011, pp. 543–569, 2013.

[32] M. Chimani, “OGDF Official Website.” [Online]. Available: http://www.ogdf.net/

[33] K. Martin, “Tutorial: Coin-or: Software for the or community,” *Interfaces*, vol. 40, no. 6, pp. 465–476, 2010.

[34] M. Jünger and S. Thienel, “The abacus system for branch-and-cut-and-price algorithms in integer programming and combinatorial optimization,” *Softw., Pract. Exper.*, vol. 30, no. 11, pp. 1325–1352, 2000.

[35] D. Auber, R. Bourqui, M. Delest, A. Lambert, P. Mary, G. Melançon, B. Pinaud, B. Renoust, and J. Vallet, “TULIP 4,” LaBRI - Laboratoire Bordelais de Recherche en Informatique, Research Report, sep 2016. [Online]. Available: https://hal.archives-ouvertes.fr/hal-01359308

[36] M. Jünger, G. W. Klau, P. Mutzel, and R. Weiskircher, “AGD — A Library of Algorithms for Graph Drawing,” 2004, pp. 149–172. [Online]. Available: http://link.springer.com/10.1007/978-3-642-18638-7{_}7

[37] J. Ellson, E. Gansner, L. Koutsofios, S. C. North, and G. Woodhull, “Graphviz— Open Source Graph Drawing Tools,” 2002, pp. 483–484. [Online]. Available: http://link.springer.com/10.1007/3-540-45848-4{_}57

[38] T. Boutell, “GD graphics library,” 2002. [Online]. Available: www.boutell.com/gd

[39] P. Joye, “gdLibrary Official Website.” [Online]. Available: https://libgd.github.io/

[40] L. Nachmanson, G. Robertson, and B. Lee, “Layered graph layouts with a given aspect ratio,” 2011. [Online]. Available: http://www.google.com/patents/US7932907

[41] A. Roques, “PlantUML.” [Online]. Available: http://plantuml.com

[42] Qt, “Qt - UI design.” [Online]. Available: https://www.qt.io/ui/

[43] A. Port, “svgpathtools - A collection of tools for manipulating and analyzing SVG Path objects and Bezier curves.” [Online]. Available: https://github.com/mathandy/svgpathtools/

[44] K. Belevich, “Nodejs-based tool for optimizing SVG vector graphics files.” [Online]. Available: https://github.com/svg/svgo

[45] N. Foundation, “Node.js.” [Online]. Available: https://nodejs.org/en/

[46] A. Hagberg, "NetworkX: Python software for complex networks." [Online]. Available: https://networkx.github.io