MASTER

A layout algorithm for workflow nets

Teurlings, M.A.

Award date:
2012

Link to publication
A layout algorithm for Workflow nets

Maik Teurlings

Project supervisors:
Huub van de Wetering and Eric Verbeek

February 2012
Abstract

In this thesis we present a layout algorithm for Workflow nets, which are a specific type of Petri net. While much research has been performed on layout algorithms for graphs, there is little known research on layout algorithms for WF-nets. Our layout algorithm is an extended version of the Sugiyama algorithm combined with the Murata reduction rules. We formulate a number of aesthetics that define a good layout. We introduce new contraction rules to contract the WF-net. On this contracted WF-net, a layout is created, while taking the aesthetic requirements into account as much as possible. Via expansion of the contraction nodes, we get a layout for the complete input WF-net. To our knowledge, the layout algorithm proposed in this thesis is currently the only one specifically designed for WF-nets. In a small use case, we demonstrate the benefits of our algorithm.
Acknowledgements

I would like to thank Huub van de Wetering and Eric Verbeek for supervising my graduation project. Thanks for all the helpful discussions, stimulating feedback and the inspiring ideas. Especially thanks for the amount of time you made available for me and for always believing in me.

I would also like to thank Michel Westenberg for supervision during the absence of Huub van de Wetering and for joining the assessment committee.

Furthermore, I would like to thank Marloes Timmermans of the Student Service Center, for the helpful talks and all her help during the finishing of the project.

And last, but certainly not least, I also would like to thank my parents, my brother, and all my friends for helping me with everything and believing in me throughout my entire study.

Maik Teurlings
February 2012
Contents

Abstract i

Acknowledgements ii

1 Introduction 1
   1.1 Problem definition ............................................. 1
      1.1.1 Aesthetics .................................................. 2
      1.1.2 Assumptions ................................................ 2
   1.2 Related work ................................................... 3
      1.2.1 Current situation .......................................... 4

2 Preliminaries 7
   2.1 Graphs .......................................................... 7
   2.2 Petri nets ....................................................... 7
      2.2.1 Murata rules ............................................... 9
   2.3 Layered drawing of digraphs .................................. 10
      2.3.1 Cycle removal .............................................. 11
      2.3.2 Layer assignment .......................................... 14
      2.3.3 Crossing reduction ....................................... 15

3 Petri net layout algorithm 19
   3.1 Layering ........................................................ 20
      3.1.1 Create the layers ......................................... 21
      3.1.2 Cycle removal ............................................. 22
      3.1.3 Unique source property ................................... 25
   3.2 Contraction ..................................................... 27
      3.2.1 Contracted Petri net ..................................... 28
      3.2.2 Contraction driver algorithm ............................ 30
      3.2.3 Contraction rule algorithm ............................... 32
   3.3 Lay-out top level Petri net .................................. 35
      3.3.1 Important arc layers ...................................... 36
      3.3.2 Create important layering ................................ 39
      3.3.3 Leveled barycentric crossing reduction ................. 41
      3.3.4 Positioning nodes ......................................... 44
   3.4 Expansion ....................................................... 47
      3.4.1 Expansion driver .......................................... 47
      3.4.2 Crossing reduction ....................................... 48
      3.4.3 Node placement ........................................... 53
4 Contraction Rules
  4.1 Rules ........................................ 55
     4.1.1 Contraction of series nodes .................. 55
     4.1.2 Contraction of parallel nodes ................. 56
     4.1.3 Other contraction rules ...................... 58
  4.2 The execution of the rules ........................ 59
     4.2.1 The order on the rules ...................... 59
     4.2.2 The place on the Petri net ................. 61
  4.3 Create dummy-nodes ................................ 61
     4.3.1 Create dummy-nodes for parallel contraction .. 62
     4.3.2 Create dummy-nodes for series contraction .. 63
  4.4 Expansion ...................................... 65
     4.4.1 Expansion for parallel contraction .......... 66
     4.4.2 Expansion for series contraction .......... 66

5 The case study .................................. 71
  5.1 The case study .................................. 71
     5.1.1 Setup of the case study .................... 71
     5.1.2 Results of the case study ................. 72
  5.2 The user study ................................... 75
     5.2.1 Participant ............................... 75
     5.2.2 Questionnaire ............................ 76
     5.2.3 Outcome ................................. 76

6 Conclusions ...................................... 77
  6.1 Conclusion ..................................... 77
  6.2 Future work .................................... 77
     6.2.1 Assumptions ............................... 78
     6.2.2 Parameters ............................... 78
     6.2.3 Limitations ............................... 79

A Contraction rules ................................ 81
  A.1 Series contraction rules ......................... 81
  A.2 Parallel contraction rules ....................... 84
  A.3 Other contraction rules ......................... 91
Chapter 1

Introduction

Nowadays there are several mathematical modeling languages. One of these are the Petri nets. They were introduced by Carl Adam Petri in the beginning of the 60’s, and since then much research has been done on the theory of Petri nets. Figure 2.1 shows an example of a Petri net. It is a directed graph with two types of nodes.

![Figure 1.1: An example of a Petri net.](image)

Less research has been done on the layout of those Petri nets, but on the other hand much research has been done on layout algorithms for graphs. The goal of this project is to create a layout algorithm for Petri nets.

1.1 Problem definition

As said we want to create a layout for Petri nets. The problem is that there aren’t graph layout algorithms that are specially created for Petri nets. In our case we concentrate on a special subclass of Petri nets, namely the Workflow nets (WF-nets). One property of a workflow net is that it has only one start and one end node. So the goal of the project is: Create a layout algorithm for WF-nets.
To achieve this we define some aesthetics the algorithm should take into account. These are a couple of standard graph layout aesthetics, but also some aesthetics specially applicable on Petri nets. All the aesthetics are described in the next section.

### 1.1.1 Aesthetics

In this section we give aesthetics for the layout of the Petri nets. Aesthetics are (measurable) properties, which we would apply as much as possible to increase the readability.

We defined the following basic aesthetics:

- **CROSSING** Minimize of the total number of crossings between the arcs.
- **AREA** Minimize the drawing area.
- **ARC_LENGTH** Minimize the variance of edge lengths.
- **BEND** Minimize the number of bends.
- **FLOW** Maximize the number of edges respecting the process flow, that is from the start node to the end node. This is an important aesthetic, because we’re dealing with process models. An exception on this aesthetic is in the cycles, where arcs may point in the other direction.
- **SYMMETRY** Display the symmetries of the Petri nets.
- **OVERLAP** Avoid overlapping nodes.

Except for the OVERLAP aesthetics, which is a hard requirement, we want to apply these aesthetics as much as possible, but we cannot enforce them. The first reason is that there can be situations where it is impossible. Take for example the non-planar graph in Figure 1.2a, this graph cannot be drawn without crossings. Another reason is because they contradict each other, take for example two representations in Figure 1.2b. In the left Petri net there is a crossing, which has disappeared in the right Petri net, at the cost of two longer arcs and 2 extra bending points.

### 1.1.2 Assumptions

To reduce the scope of our research, we made some assumptions to the Petri nets that we want to lay out. These assumptions help us to ease the creation of the layouts. As future work there can be an investigation whether or not we can relax these assumptions.
Workflow nets (WF-nets) There are different subclasses of Petri nets, one of them is a Workflow net. More information about WF-nets can be read in section 2.2. We only deal with these WF-nets.

Small The Petri nets we’re dealing with are not enormous, the number of transitions and places together is less than 500.

Size of the nodes The nodes have an initial size, which is the same for all the nodes of the same kind. Also no investigation is needed for the labeling of the nodes.

1.2 Related work

Much research on the layout of all kinds of graphs has been performed, an overview is described in [14]. Also a lot of research has been done on visualizing large graphs, an overview can be found in [16]. We have to deal with digraphs, and our input Petri nets aren’t large, therefore we focus us on the layout of digraphs.

The most important one is the layered drawing of digraphs. This method was first described by Sugiyama et al. [13] and further explored in the literature [2, 14]. This approach consists of four steps. The first step is the transformation of a possible cyclic digraph to an a-cyclic digraph by reversing edges of cycles. The second step is the placement of cycles on horizontal layers, which determines the Y-coordinate. The third step is the crossing reduction, this determines the order of the nodes on each layer. In the fourth step, the X-coordinates of the nodes are determined. Figure 1.3 gives an example of the last three steps.

There are also some domain-specific layout algorithms for other modeling languages, such as for the Business Process Execution Language (BPEL) [1]. BPEL is a is a verbose and hierarchical work flow language. This approach is based on the layered drawing of
The most important extension is the addition of Path ways. Path ways are sets of nodes that should be vertically aligned. Therefore, these nodes should have special attention during the steps of the layered graph drawings, that they are placed near each other and can be aligned vertically. Figure 1.4 shows an example of a layout created with the algorithm.

1.2.1 Current situation

As said current algorithms have some shortcomings, because they are not created to fit the Petri net domain. In this section we provide some examples of the current layout algorithm of ProM. ProM is a generic open-source framework for implementing process mining tools in a standard environment [9]. It uses JGraph to create the layouts for the Petri net [6]. We discuss the drawbacks of those examples and we use them to show the improvements of our proposed algorithm.
**Petri net 1**
In Figure 1.5 we see a Petri net with one cycle involving $t_{10}$. If we place transition $t_{10}$ between the two places, we can draw this graph planar, now we see 5 crossings (There is one point where 3 lines cross each other, these are 3 crossings). Another issue is the number of bending points, there are 32 of them.

![Figure 1.5: Petri net 1.](image)

**Petri net 2**
The layout of the Petri net in Figure 1.6 can be much more symmetric. The names of the transitions correspond with the cardinal directions, and say something about the place. Here they are placed more or less on a line. It has also 14 bending points and contains two crossings.

**Petri net 3**
The layout of the Petri net in Figure 1.7 is very odd. We have many long arcs, with many bending points. It seems like the nodes on the right can be placed more to the left. This Petri net uses 23 layers.
Figure 1.6: Petri net 2.

Figure 1.7: Petri net 3.
Chapter 2

Preliminaries

In this chapter we discuss some concepts needed later on. First we discuss the Petri net mathematical modeling language. We show what they look like, and give a brief overview of the Murata reduction rules, which form the inspiration of our layout algorithm. Secondly, we discuss the layered drawings of directed graphs.

2.1 Graphs

A graph $G = (V, E)$ is a set of nodes (vertices) $V$ and a set of edges $E \subseteq V \times V$. An edge connects a pair of nodes. A directed graph or digraph is a graph with a direction on the edges, i.e. the edges are arcs. If $(u, v) \in E$, then there is an arc from node $u$ to node $v$, and also $(u, v) \neq (v, u)$. A bipartite graph is a graph in which the nodes can be divided into two disjoint subsets $V_1 \cup V_2 = V$ such that all the arcs are connected between one vertex in $V_1$ and one in $V_2$, i.e. $E \subseteq (V_1 \times V_2) \cup (V_2 \times V_1)$.

2.2 Petri nets

Petri nets are a mathematical modeling language to model processes. Petri nets were founded by Carl Adam Petri, since then much research is done on the theory of Petri nets. Figure 2.1 shows an example of a Petri net. It is a bipartite directed graph with two different kinds of nodes, namely places and transitions. Places are represented by circles and transitions are represented by rectangles. If we recall the definition of a bipartite graph, we can project the places to $V_1$ and the transitions to $V_2$.

Definition 2.1 shows the formal definition of a Petri net.
Definition 2.1 (Petri net). A Petri net is a 3-tuple, $PN = (P, T, F)$, where:

$P$ is a finite set of places,
$T$ is a finite set of transitions, and
$F \subseteq (P \times T) \cup (T \times P)$ is a set of arcs (flow relation),
such that $P \cap T = \emptyset$ and $P \cup T \neq \emptyset$.

A node $v$ is called an input node of another node $u$ if and only if there is an arc $(v, u) \in F$. Likewise, a node $v$ is called output node of another node $u$ if and only if there is an arc $(u, v) \in F$. In our example Petri net place $p4$ is an input place of transition $t7$ and transition $t10$ is an output transition of place $p8$. The set of all input nodes of a node $v$ is called the preset, denoted $\bullet v$, and the set of all output nodes of a node $v$ is called the postset, denoted $v \bullet$. Some examples of presets and postsets of our example Petri net are: $\bullet t2 = \{p1\}$, $p3 \bullet = \{t5, t6\}$, $\bullet t6 = \{p3, p4\}$ and $t6 \bullet = \{p4, p8\}$. But also $\bullet p1 = \emptyset$ and $p11 \bullet = \emptyset$ holds, as $p1$ has no input transitions and $p11$ no output transitions. In these cases $p1$ is called a source node and $p11$ is called a sink node. We can also define presets and postsets on a set of nodes, these are the union of all presets or postset of all nodes from that set, see definition 2.2.

Definition 2.2 (Pre- and Postset of sets). Let $V$ be a set of nodes.

The postset of this set of nodes $V \bullet$ is the union of all postsets of each node of $V$, i.e.
$V \bullet = \bigcup_{v \in V} v \bullet$.

The preset of this set of nodes $\bullet V$ is the union of all presets of each node of $V$, i.e.
$\bullet V = \bigcup_{v \in V} \bullet v$.

A Workflow net (WF-net) is a special kind of Petri net, introduced by Van der Aalst [15]. A Petri net is called a WF-net if it satisfies the following requirements:

1. It has exactly one source place, which is called the begin place, i.e. $\exists ! p \in P : \bullet p = \emptyset$.
2. It has exactly one sink place, which is called the end place, i.e. $\exists ! p \in P : p \bullet = \emptyset$. 

\[ \text{Figure 2.1: An example of a Petri net.} \]
3. Every node is on at least one path from the begin place to the end place.

In our example place $p1$ is the only place with an empty preset (the begin place), place $p11$ is the only place with an empty postset (end place) and all other nodes are on at least one path from $p1$ to $p11$, hence it is a WF-net.

2.2.1 Murata rules

Murata presented [8] six rules to reduce Petri nets into smaller Petri nets, while preserving boundedness and liveness. Boundedness and liveness together correspond to soundness, which is a desirable property for a WF-net (see [15]).

The first two rules reduce one transition and two places to a single place, or one place and two transitions to a single transition. These rules are called, respectively, the Fusion of Series Places (FSP) and Fusion of Series Transition (FST), and are depicted in Figure 2.2.

FSP has two preconditions. The first one is that the transition has only one input, and only one output place, i.e. $•t = \{p1\}$ and $t• = \{p2\}$. The second one is that the input place $p1$, do not have more than one output transition (no other output transition than the one that is being reduced), i.e. $p1• = \{t\}$. For the FST rule the same hold for the place that’s being reduced. It has only one input and one output place, i.e. $•p = \{t1\}$ and $p• = \{t2\}$. For the input and output transitions the preconditions are mirrored. Here the output transition should have only the place as input place, i.e. $•t2 = \{p\}$.

The next two rules reduce two parallel places or two parallel transitions to one. These rules are called the Fusion of Parallel Places (FPP) and Fusion of Parallel Transition (FPT), and are depicted in Figure 2.3. The preconditions of FPP is that the two places have the same preset and postset, i.e. $•p1 = •p2$ and $p1• = p2•$. Similarly for the FPT rule hold: $•t1 = •t2$ and $t1• = t2•$. Notice that in Figure 2.3 only one input node and only one output node are depicted, but there may be more.
The last two rules reduce self-loops. They are called the Elimination of Self-loop Places (ESP) and the Elimination of Self-loop Transitions (EST), and are depicted in Figure 2.4. The precondition of ESP \(^1\) is that place \(p\) may only contain one input transition and one output transition, which is the same transition as the input transitions, i.e. \(p \cdot = \cdot p = \{t\}\). For the EST a similar precondition hold. The transition may only contain one input place and one output place, which is the same place as the input place, i.e. \(t \cdot = \cdot t = \{p\}\).

2.3 Layered drawing of digraphs

In this section we discuss the layered drawings of digraphs. This method was first described by Sugiyama et al. [13] and further explored in the literature [2, 14]. This method is divided into 4 steps:

**Step 1.** Cycle removal

Due to the fact that the next step of the method is only applicable on a-cyclic digraphs, the cycles of a cyclic digraph should be removed. One edge of each cycle is reversed, to break the cycles. After the layout steps, the reversed cycles will be inverted, to retrieve the original graph.

**Step 2.** Layer Assignment

Every node will be assigned to a layer, which determines the X-coordinate of the nodes in the final layout.

\(^1\)ESP has also one behavioral precondition, but that is out of the scope for this thesis
Step 3. Crossing Minimization
The nodes on the layers will be ordered such that the number of crossings in the graph is reduced.

Step 4. Computations of Coordinates
The Y-coordinate of the nodes will be calculated.

We now give an overview of the different steps, as they are important for the understanding of our algorithm. For more details we refer to the literature [2, 13, 14]. One main difference between the overview and the literature is the orientation of the graph. In the literature the graphs are described in the direction from top to bottom, we’ll describe the method in the orientation from left to right, as we will use it for WF-layouts.

2.3.1 Cycle removal

To perform the second step of the algorithm, the digraph needs to be a-cyclic. Therefore the cyclic digraph will be transformed into an a-cyclic digraph by reversing arcs, which are part of the cycles. This NP-hard problem is equivalent with the feedback arc set problem: find a minimal set $E_f \subseteq E$, such that the graph $(V, E \setminus E_f)$ contains no cycles.

One approach for this problem is the Greedy Cycle Removal [14]. This approach contains two steps:

Step 1. Sequentialize
The nodes are placed in a sequential order from left to right.

Step 2. Reverse leftward arcs
The arcs that point left (leftward arcs) are part of a cycle, and are reversed.

Step 1 Algorithm 1 is used to place the nodes into a sequential order. The goal of this algorithm is to place the nodes such that the number of leftward arcs is approximately the smallest. The algorithm removes nodes from the graph $G$ and adds them to one of the two lists $S_l$ or $S_r$. When all nodes are removed from $G$ the output sequential order will be the concatenation of the two lists $(S_l \cup S_r)$. Figure 2.5 shows a scheme of this approach and Figure 2.6 a running example.

First the sink nodes are prepended to $S_r$ (line 4) with sub-algorithm 2. They have only input arcs, so they won’t cause leftward arcs when they are placed on the right hand side in the sequential order. After a removal of a sink, other sinks can appear, which can also be prepended in turn to $S_r$. In Figure 2.6a node $v11$ is a sink and will be prepended
to $S_r$, after the removal of $v_{11}$, node $v_{10}$ becomes also a sink and will also be prepended to $S_r$, see Figure 2.6b.

Second the sources are appended to $S_l$ (line 5) with sub-algorithm 3. They have only output arcs, so they won’t cause leftward arc when they are placed on the left hand side in the sequential order. After a removal of a source, other sources can appear, which can also be appended in turn to $S_l$. In Figure 2.6b node $v_1$ is a sink node, and will be appended to $S_l$, see Figure 2.6c.

If all sinks and sources are removed, but there are still nodes in $G$, a node $v$ is chosen for which $|v \bullet | - |\bullet v|$ is maximum and added to $S_l$ (line 7) with sub-algorithm 4. This choice ensures that the number of rightward arcs is relatively the highest. In Figure 2.6c, there are no sinks and sources, so a node is to be chosen. In this case node $v_6$ is the only node which has maximum number of $|v_6 \bullet | - |\bullet v_6| = 2 - 1 = 1$, so $v_6$ is chosen and added to $S_l$, see Figure 2.6d.

After the removal of this node, sinks and sources may arise again, so the algorithm starts again by removing those nodes, and repeats these steps until $G$ is empty. The returned sequential order is the concatenation of $S_l$ and $S_r$. In Figure 2.6d the repeatedly removal of the arising sinks is enough to make $G$ empty, and the sequential order is built, see Figure 2.6e.

**Algorithm 1: Serialize($G$)**

**Input:** Graph $G = (V,E)$

**Output:** Sequential list $S = [v_1, ..., v_k] : \{v_1...v_k\} = V \land k = |V|

1. $S_l \leftarrow []$
2. $S_r \leftarrow []$
3. **while** $G \neq \emptyset$ **do**
   4. $S_r \leftarrow \text{RemoveSinks}(G) \; + \; S_r$
   5. $S_l \leftarrow S_l \; + \; \text{RemoveSources}(G)$
   6. **if** $G \neq \emptyset$ **then**
      7. $\{\text{We have at least 1 cycle}\}$
      8. $S_l \leftarrow S_l \; + \; \text{ChooseNodes}(G)$
4. **return** $S_l \; + \; S_r$
Step 2
Algorithm 5 takes the sequential order and creates a reversion set $E_{rev}^S$ with all leftward arcs. This is trivially done by walking through the sequential order, and checking the postset for every node. If a node of the postset has already been visited in the sequential order, the arc between those nodes is a leftward arc and will be added to the reversion set $E_{rev}^S$. If a node of the postset has not been visited the arc is a rightward arc. In Figure 2.6f the arcs of $E_{rev}^S$ are reversed and depicted with dotted lines.
Algorithm 2: RemoveSinks($G$)

**Input**: Graph $G = (V, E)$

**Output**: Sequential list $S = [n_1, \ldots, n_k] : (n_1, \ldots, n_k \in V)$ with the (arised) sink nodes

**Result**: The nodes of $S$ are removed from $G$

1. $S \leftarrow []$
2. while ($\exists v \in V : v \bullet = \emptyset$) do
3. \hspace{1em} $S \leftarrow [v] + S$
4. \hspace{1em} $G \leftarrow (V \setminus \{v\}, E)$
5. return $S$

Algorithm 3: RemoveSources($G$)

**Input**: Graph $G = (V, E)$

**Output**: Sequential list $S = [n_1, \ldots, n_k] : (n_1, \ldots, n_k \in V)$ with the (arised) source nodes

**Result**: The nodes of $S$ are removed from $G$

1. $S \leftarrow []$
2. while ($\exists v \in V : \bullet v = \emptyset$) do
3. \hspace{1em} $S \leftarrow S + [v]$
4. \hspace{1em} $G \leftarrow (V \setminus \{v\}, E)$
5. return $S$

Algorithm 4: ChooseNodes($G$)

**Input**: Graph $G = (V, E)$

**Output**: Sequential list $S = [n] : (n \in V)$ with the chosen node

**Result**: The node of $S$ is removed from $G$

1. $v \leftarrow x \in V : |x \bullet | - |\bullet x| = (\uparrow |u \bullet | - |\bullet u| : u \in V)$
2. $S \leftarrow [v]$
3. $G \leftarrow (V \setminus \{v\}, E)$
4. return $S$

2.3.2 Layer assignment

In this step the nodes will be assigned to layers, which also determines the X-coordinates of the nodes. A *layering* of a graph $G$ is a partition of the nodes $V$ into subsets $L_1, L_2, \ldots, L_h$. The nodes are assigned to the layers such that the arcs of the a-cyclic graph, all point to the same direction, i.e. if $(u, v) \in E$ and $u \in L_i$ and $v \in L_j$ then $i > j$. The *span* of an edge $(u, v)$ with $u \in L_i$ and $v \in L_j$ is $i - j$. The layering is *proper* if it doesn’t have edges on more than one layer, i.e. no edge has a span larger than one. Proper layering is necessary for crossing reduction.

For all edges for which this doesn’t hold, dummy nodes will be added on the path on the skipped layers. For example in Figure 2.7 we have $(v3, v7) \in E$ and $v3 \in L_4$ and $v7 \in L_1$, this edge is split up into $(v3, d_1)$, $(d_1, d_2)$ and $(d_2, v7)$ and the two dummy nodes $d_1$ and
Algorithm 5: CREATEREVERSIONARCSET($S$)

Input: The sequential order $S$ of a digraph $G = (V, E)$

Result: A set of arcs $E_{rev} \subset E$ for which holds that $(V, E) \setminus E_{rev}$ is cycle free

1. $E_{rev} \leftarrow \emptyset$
2. $V_{seen} \leftarrow \emptyset$
3. while $S \neq [\ ]$ do
4.   $v \leftarrow \text{head}[S]$
5.   $E_{rev} \leftarrow E_{rev} \cup \{(v, n) \in E : n \in V_{seen}\}$
6.   $V_{seen} \leftarrow V_{seen} \cup \{v\}$
7.   $S \leftarrow \text{tail}[S]$
8. end
9. return $E_{rev}$

d$_2$ are placed on layers $L_3$ and $L_2$. These dummy nodes will be the bending points of the arcs.

There are several strategies to place the nodes on the layers; one of them is the so-called Longest Path Layering. All sinks are placed on layer $L_1$, each remaining node is placed on layer $L_{p+1}$ where $p$ is the length of the longest path from $v$ to a sink. The height of the layering is the number of layers $h$, so the nodes are divided into the subsets $L_1, L_2, ..., L_h$. Note that the longest path to a sink only exists in the absence of loops.

As shown in Figure 2.7 nodes $v7$ and $v8$ are sink nodes and placed on layer $L_1$. Node $v3$ can reach a sink in only one step ($v3, v7$), but the longest path contains 3 steps ($v3, v4$, $(v4, v6)$ and $(v6, v7)$ or ($v3, v4$), $(v4, v6)$ and $(v6, v8)$, hence node $v3$ is placed on layer $L_{3+1} = L_4$. The height $h$ of this example is 6.

![Figure 2.7: The layer assignment of a graph. The grey dots are the dummy nodes.](image)

2.3.3 Crossing reduction

In this step the number of crossings will be reduced, by (vertically) ordering the nodes on the layers. Most heuristics (it is a NP-complete problem) use the layer-by-layer sweep method. A vertex ordering of one layer is chosen and held fixed. The ordering of the
next layer is permuted such that the crossings between the two layers is reduced. In the next iteration, the permuted layer is held fixed, and the next layer is permuted. This is done though the whole graph. There are different directions on which these iterations can be performed. For example going from the first to the last layer or from the last to the first layer.

There are different methods to choose a permutation of a layer. One of them is the barycentric method, introduced by Sugiyama et al. [13]. In this method for each node on the permuted layer a barycenter (average) of the relative y-coordinates of the connected nodes on the fixed layer will be calculated. The nodes on the permuted layer will be ordered on those barycenters. If nodes have equal barycenter, the order will be preserved. The sortation on the barycenters is called Phase-1.

In Phase-2 the nodes on a layer with the same barycenters will be reversed in each set. After a reversion on a layer, Phase-1 is executed again, because the barycenters on the next layer can be changed, so phase 2 uses phase 1.

Each phase contains 2 different procedures, for each direction one. Phase-1 (left) and phase-2 (left) walks layer 2 (layer 1 is then fixed) through layer \( h \) (layer \( h - 1 \) is then fixed). Phase-1 (right) and phase-2 (right) walks layer \( h - 1 \) (layer \( h \) is then fixed) through layer 1 (layer 2 is then fixed). When the other phase is executed after one phase, it will be in the same direction.

After each execution of phase-1, the number of crossings will be checked if it is the best result so far. If it is, the number of crossings with the orderings will be stored. The number of crossings is calculated by checking for each arc, the number of crossings with other arcs. Because each crossing is counted twice, by the investigation of each arc once, the total number divided by 2 is the number of crossings. Take for example the graph of Figure 2.8a, here the number of crossings for the arcs are \((v_1, v_5) = 2\) (crossings with arcs \((v_2, v_4)\) and \((v_3, v_4)\)), \((v_2, v_4) = 1\) (crossing with \((v_1, v_5)\)), \((v_3, v_4) = 1\), \((v_6, v_8) = 1\), \((v_5, v_9) = 1\), \((v_7, v_12) = 2\), \((v_8, v_10) = 1\) and \((v_8, v_11) = 1\), which makes the total number 10. So the total number of crossings is 10/2 = 5.

The algorithm stops if the number of crossings is 0 or a maximum number of iterations is performed, and returns the ordering on the nodes, which is stored as the best one.

Formula 2.1 calculates the barycenter of a node \( v \) regarding the connected nodes on layer \( i \). Here \( V^i_v \) denotes the nodes on layer \( i \) connected to the node \( v \), i.e. \( V^i_v = L_i \cap (v \cup L_i) \). As one can see, there is a chance for a division by zero (if \( |V^i_v| = 0 \)) in that case, the node will be omitted in the sortation.
\[ B^i(v) = \frac{1}{|V_i^v|} \sum_{w \in V_i^v} w.y \]  

(2.1)

Take for example the graph on Figure 2.8a, in the first step of the Phase-1-RIGHT procedure, we calculate and order the barycenter numbers for the nodes \( v_4, v_5 \) and \( v_6 \). For the relative \( y \)-coordinates we use the ordering number, that is 1 for \( v_1 \), 2 for \( v_2 \) and 3 for \( v_3 \). The barycenters for \( v_3 \) and \( v_4 \) are \( B_1^1(v_4) = \frac{1}{2} \cdot (2 + 3) = 2, 5 \), \( B_1^1(v_5) = \frac{1}{1} \cdot 1 = 1 \), no barycenter can be calculated for \( v_6 \). Figure 2.8b shows the graph after the sortation of the nodes on those barycenters (\( v_5, v_4, v_6 \)). After this sortation, the barycenters of the next layers will be calculated, and sorted. Figure 2.8c shows the graph after the whole execution of Phase-1-RIGHT. The number of crossings is now 1.

In Phase-2-RIGHT the vertices (\( v_{10}, v_{11} \)), with the same barycenters are reversed, see Figure 2.8d. Note that the barycenters in this Figure are now calculated to the previous layers, for the LEFT direction. In Phase-2-LEFT the vertices (\( v_4, v_5 \)), with the same barycenters are reversed, see Figure 2.8e. In Phase-1-LEFT, nodes \( v_1, v_2 \) and \( v_3 \) are reordered, Figure 2.8f. The number of crossings is now 1. Finally in Phase-1-RIGHT the nodes \( v_9 \) and \( v_7 \) are ordered, see Figure 2.8g for the barycenters and 2.8h for the resulting graph, the number of crossings is now 0.
Figure 2.8: A running example of the crossing reduction method.
Chapter 3

Petri net layout algorithm

The algorithm we present is the layered drawings of digraph method combined with contraction rules inspired on the Murata reduction rules. The idea behind this approach is to contract nodes that are locally easy to layout, which makes the remainder of the graph easier to layout. Figure 3.1 shows the scheme of our algorithm against the scheme of the layered drawings of digraph method. When we talk about Petri nets, we mean WF-nets.

![Layout algorithm for Workflow nets](image)

**Figure 3.1:** The schematic representation of the our algorithm step against the steps of the layered drawings of digraph method

Our first step, the layering step, has the same goal as the second step of the layered drawings of digraph method: the nodes will be subdivided into different layers. This is necessary to apply the FLOW aesthetic. Our layering method differs from the layering method of the layered drawings of digraph. In our layering step we use the WF-net property of the one start place, which layers the nodes closer to each other. Another difference is that our layering is not proper at this time, as we aren’t interested in that yet.
To achieve this, the input Petri net should be a-cyclic, therefor a preprocessing step to transform a cyclic Petri net into an acyclic Petri net is needed, like the first step of the layered drawings of digraph method. We also have a cycle removal step, but we combine it into the layering step, because it’s only for the layering of the Petri net. The difference between our cycle removal method and current cycle removal methods is that we’re not interested in breaking the cycles by reversing the least number of arcs, but in breaking the cycles in such way they get a compacter layout. Our layering step is described in section 3.1.

Our second step is a completely different step than the layered drawings of digraph method. In this step we want to reduce the Petri net, such that the remainder is easier to layout. This is done by replacing multiple nodes with a single node by predefined contraction rules. We still want to have the information of the original nodes, therefore we create our own extended Petri net structure, namely the contracted Petri net (CPN). One extension to it is the parent relationship, which indicates which nodes are replaced with the new node. More about this structure is described in subsection 3.2.1. In this chapter we describe the application of the contraction rules in general. A more detailed description is given in Chapter 4 and Appendix A contains an overview of the rules.

We perform the third and fourth step of the layered drawings of digraph method together in the third step. We have contracted the petri net to a smaller net. For this net we create a layout, which we assume to be easier as it contains far fewer nodes.

In the fourth step we again perform the step of the layered drawings of digraph method together. This time we iteratively place the multiple nodes back for each node that replaced them, in the already created layout. This way the task of laying out the original petri net is divided into smaller tasks, which is assumed to be easier.

When all nodes are placed back, we have the original Petri net again, with its lay-out.

### 3.1 Layering

This part of the algorithm will create a layering on the WF-net, see Figure 3.2 for a schematic overview of this part of the algorithm. The input of this step is the original WF-net $WF$. To create a layering for the WF-net we first make the WF-net acyclic in the cycle removal part. In this step a set of arcs is created, which makes the Petri net cycle free if they are reversed. A copy of the original Petri net $WF$ but with the arc reversed is the input of the layering part. The output of this part is only the partitions of the nodes in the layers $\{L_1, \ldots, L_h\}$. These subsets together with the original Petri net is the output of this step.
We discuss the layering of the Petri net in Subsection 3.1.1, which is before the cycle removal part of Section 3.1.2, to make the cycle removal part more understandable.

### 3.1.1 Create the layers

In this step we want to find a layering for a Petri net. We demand two properties of the Petri net, the first one is that it must be an acyclic Petri net otherwise it cannot be layered in such way that all the arcs are pointing in the same direction. The second one is that the Petri net contains only one source node. Initially a WF-net has only one sink and only one source node, but due to the cycle removal extra sinks can be introduced, one restriction we make here is that no extra sources may be introduced.

One strategy is the Longest Path Layering. This strategy places all sink nodes in layer $L_1$, and all other nodes in the layer $L_{p+1}$, where $p$ is the longest path, counted in the number of arcs, from the node to a sink node. However this may introduce unnecessary long arcs, due to the fact that there can be more sinks, which are all placed in layer $L_1$ (see Figure 3.3a).

![Figure 3.3](image)

**Figure 3.3:** Petri net a is layered from the sinks, Petri net b is layered from the sources. The dotted arcs are the reversed arcs.
Due to the restriction that there is only one source node, an alternative for this strategy is to start the layering from that unique source node (see Algorithm 6). The source node is placed in layer \( L_1 \) and all other nodes \( v \in P \cup T \) are placed in layer \( L_{p+1} \), where \( p \) is the longest path counted in the number of arcs from the source node to the node \( v \). This way all other nodes will be placed as close as possible to the source node, so no arc is longer than necessary (see Figure 3.3b).

**Algorithm 6: LayerTheNet(PN)**

**Input:** Acyclic Petri net \( PN = (P, T, F) \) with exactly one source node

**Result:** \( \{L_1, ..., L_h\} : \bigcup_{i=1}^h L_i = P \cup T \land (\forall(u,v) \in F \land u \in L_i \land v \in L_j : i < j) \)

1. \( v_s \leftarrow v \in P : \bullet v = 0 \)
2. forall \( v \in (P \cup T) \) do
   3. \( p \leftarrow \# \text{arcs on the longest path from} \ v_s \text{ to} \ v \)
   4. \( L_{p+1} \leftarrow L_{p+1} \cup \{v\} \)
5. end
6. return \( \{L_1, ..., L_h\} \)

We place the resulting subsets into the Petri net definition, which makes the output of this step a Layered Petri net: \( LPN = (P, T, F, \{L_1, ..., L_h\}) \).

### 3.1.2 Cycle removal

In this step we make a Petri net acyclic, by reversing some arcs that are part of a cycle. To achieve this we use an approach similar to the greedy cycle removal algorithm (see Algorithm 1). That is, we first create a sequential ordering of the nodes \( P \cup T \) of the Petri net: \( S = (v_1, v_2, ..., v_n) \). Secondly an arc set \( F_{rev}^S \) that contains all leftward arcs of sequence \( S \) is created. Reversing the arcs in this set makes the Petri net acyclic. We use a different strategy for the creation of the sequential order, the creation of the leftward arc-set is still the same.

**Create sequential order**

Algorithm 7 creates a sequential order on the nodes. This algorithm is similar to the greedy cycle removal algorithm (see Algorithm 1), but we use a different CHOOSENODES sub-algorithm (line 8).

The greedy cycle removal algorithm wants to have a small \( F_{rev}^S \), therefore it chooses the node for which \( |v \bullet| - |\bullet v| \) is maximized. See for example the cyclic part of the Petri net of Figure 3.4a. If a smallest set \( F_{rev}^S \) is reversed, which is a single arc, in the layering step the nodes will be placed all in a consecutive order (see Figure 3.4b).
Algorithm 7: Serialize($PN$)

Input: Petri Net $PN = (P, T, F)$
Output: Sequential list $S = [n_1, ..., n_k] : \{n_1...n_k\} = P \cup T \land k = |P \cup T|

1. $S_l \leftarrow []$
2. $S_r \leftarrow []$
   
   \{Create a copy of the Petri net\}
3. $PN' \leftarrow (P, T, F)$
4. while $PN' \neq \emptyset$
5.   $S_r \leftarrow $ RemoveSinks($PN'$) + $S_r$
6.   $S_l \leftarrow S_l + $ RemoveSources($PN'$)
7. if $PN' \neq \emptyset$
8.   \{We have at least 1 cycle\}
9.   $S_l \leftarrow S_l + $ ChooseNodes($PN'$)
10. end
11. return $S_l + S_r$

---

Figure 3.4: Petri net $a$ is a part of a Petri net with a small cycle. Petri net $b$ shows the result of the layering step, when the arc $(t10, p3)$ is reversed. Petri net $c$ shows the result of the layering step, when four arcs are reversed. The reversed arcs are depicted with dotted lines.

We want the representation of cycles to be more compact, therefore we possibly need to reverse more than one arc of a cycle, so the layering step can place nodes more parallel (see Figure 3.4c). To achieve this we choose a special node and from that node we build a sequence that transforms more arcs into leftward arcs (see Algorithm 8).

We choose an entry node as that special node. A feature of WF-nets is that all nodes of the net are situated on a path from the unique start place to the unique end place. Therefore cycles have at least one entry node and one exit node. A node is called an entry node if it is part of a cycle and it has an input arc that is not part of that cycle (see Definition 3.1). A node is called an exit node when it is part of a cycle and it has an output arc that is not part of that cycle (see Definition 3.2). Notice that a node can be both an entry and an exit node.

Definition 3.1 (Entry node). Let $PN = (P, T, F)$ be a Petri net and $c = (n_1, ..., n_k)$
**Algorithm 8:** `CHOOSENODES(PN', PN)`

**Input:** Petri Net $PN' = (P, T, F)$ from which nodes are removed
**Input:** Original Petri Net $PN = (P, T, F)$ to retrieve the entry and exit nodes
**Output:** Sequential list $S = [n_1, ..., n_k] : (n_1, ..., n_k \in P \cup T)$
**Result:** The nodes of $S$ are removed from $PN$

1. $v \leftarrow x \in P \cup T : PN.IsEntryNode(x)$
2. $S \leftarrow [v]$
3. Remove $v$ from $PN$

   {Repeatedly remove sinks, which are not exit nodes, and append them to $S$}
4. **while** $(\exists v \in P \cup T : v^\bullet = \emptyset \land \neg PN.IsExitNode(v))$ **do**
5.   $S \leftarrow S + [v]$
6.   Remove $v$ from $PN$
7. **end**
8. **return** $S$

be a cycle, where $n_1, ..., n_k \in P \cup T$. A node $n \in c$ is an entry node for cycle $c$ if and only if $n^\bullet$ has nodes not in $c$, i.e. $n^\bullet \setminus c \neq \emptyset$.

**Definition 3.2** (Exit node). Let $PN = (P, T, F)$ be a Petri net and $c = (n_1, ..., n_k)$ be a cycle, where $n_1, ..., n_k \in P \cup T$. A node $n \in c$ is an exit node for cycle $c$ if and only if $n^\bullet$ has nodes not in $c$, i.e. $n^\bullet \setminus c \neq \emptyset$.

We see an entry node as a start of a cycle, and an exit node as an end of a cycle. We prefer that all nodes are layered between two such nodes, therefore we want to reverse as many arcs as possible on the path from the exit node to the entry node. We achieve this by backtracking paths that are going to the entry node. Note that a cycle may have multiple entry and multiple exit nodes, so we just choose one entry node in this algorithm (line 1). After the removal of the entry node, the ends of these paths become sink nodes. So continuously removing the arising sinks and appending them to the end of the left list, transforms the arcs of those paths into leftward arcs (lines 4 ... 6). We don’t remove and place sink nodes that are also exit nodes, as they correspond to the beginning of such paths. When there isn’t a sink (anymore), it means that a part of such a path is also contained in another cycle which we do not want to interfere with at this moment.

**Creation of reversion arc set**
For the creation of the reversion arc set $F_{rev}^S$, we can use the normal algorithm 5.

**Running example**
Figure 3.5 depicts a running example of this algorithm, where the Petri net of Figure
Petri net layout algorithm

3.5a is the input Petri net. First we remove the sinks and sources and add them to the lists (lines 5 and 6) in the same way as the greedy cycle removal algorithm, so with the same sub-algorithms RemoveSinks 2 and RemoveSources 5. The removed sinks are put at the front of the right list (see Figure 3.5b). The sources are put at the end of the left list (see Figure 3.5c).

Then we choose the entry place $p_2$ and place it at the end of the left list. After this transition $t_{13}$ becomes a sink and is removed and placed at the end of the left list. The same is done for place $p_{12}$, which is then a sink, and for $t_{11}$, which becomes a sink after the removal of $p_{12}$. Place $p_6$ is a sink node, but also an exit node, so this part of the algorithm stops here and the Serialize algorithm continues at line 5. Figure 3.5d shows the result of this part of the algorithm.

Again sinks and sources are removed. So place $p_6$, which is a sink, will now be removed and placed at the front of the right list (see Figure 3.5e). When there are other cycles, the algorithm again comes at line 8 and chooses again a node as source node and places the sinks at the end of the left list (see Figure 3.5f).

This is repeated until no nodes are left. Then all nodes are either in the left list or right list. The concatenation of the lists results into the sequential order (see Figure 3.5f).

Algorithm 5 takes the sequential order and creates a reversion set $F^S_{rev}$ with all leftward arcs. In Figure 3.5g the arcs of $F^S_{rev}$ are reversed and depicted with dotted lines.

3.1.3 Unique source property

As stated in section 3.1.1 the net should have only one source place. A WF-net has initially only one source and only one sink place, but as can be seen in the running example the cycle removal can introduce sink nodes.

So we must ensure that no extra source places will be introduced. This is the case for our cycle removal method: The only way that reversion of leftward arcs in the sequential ordering transforms a node into a source is when all incoming arcs are leftward arcs, so it doesn’t have an incoming rightward arc. This situation cannot occur with our serialize method. There are three different ways we add nodes into the sequential order:

I. We place all sink nodes on the right list, which are the only nodes we place on the right list that will be the end of the sequential order. Therefore we know that the input nodes of a sink will be placed before the sink, so those sinks have incoming rightward arcs.
II. We place source nodes at the end of the left list. The first source is the start place of the WF-net, which is the only source node of the original net. All other source nodes became a source node when all input nodes were removed and placed on the sequential order. These input nodes cannot be sink nodes, so they are placed on
the left list. The generated source node will be placed after its input nodes, hence it has at least one incoming rightward arc.

III. When we place an entry node on the list, it has an already placed input node on the left list otherwise it wasn’t an entry node, so those nodes have also at least one incoming rightward arc. All the sink nodes, placed after the entry node, have outgoing leftward arcs that will be reversed into incoming arcs, so those nodes also won’t be transformed into source nodes.

### 3.2 Contraction

In this part of the algorithm we decrease the number of nodes and arcs of the Petri net by contracting nodes and arcs, see Figure 3.6 for a schematic overview of this part of the algorithm. The input of this step is a layered Petri net. The goal is to ease the layout of the remainder of the graph. The contraction rules are inspired on the Murata reduction rules. In this section we discuss the global application of the contraction rules, Chapter 4 contains the elaboration of these rules. We discuss the driver algorithm, which deals with the recurrent execution of the rules on the Petri net in Subsection 3.2.2. The global application of a rule is described in subsection 3.2.3.

When we contract a set of nodes, the so called **contracted nodes**, we replace these nodes with a new node, the so called **contraction node**. We want to keep the information of the contracted nodes, therefore we create our Contracted Petri net definition, which is a layered Petri net with extra information, see subsection 3.2.1. The output of this step is a contracted Petri net **CPN**, created on the input layered Petri net **LPN**, after the execution of the contraction rules. So on this contracted Petri net **CPN** no rule can be applied anymore.
3.2.1 Contracted Petri net

Definition 3.3 is the definition of a Contracted Petri Net. This Petri net contains one extra kind of node, the dummy nodes $D$, and it has a parent relationship ($p$) defined on the nodes and arcs.

**Definition 3.3** (Contracted Petri Net).

A contracted Petri net $CPN = (P, T, D, F, p, \{L_1, \ldots, L_h\})$ is a Layered Petri net $LPN = (P, T, F, \{L_1, \ldots, L_h\})$ with an additional node set $D$ and an additional parent relation $p$.

- $D \subseteq (F \times L_n)$ is a finite set of Dummy nodes
- $p \subseteq ((P \cup T) \to (P \cup T)) \cup (D \to (P \cup T)) \cup (F \to (P \cup T)) \cup (F \to F)$ is the parent relationship.

**Parent relationship**

When we replace the contraction nodes with the contracted node, we want to keep the information of the contracted nodes, therefore they won’t be removed from the Petri net. To indicate that those nodes are replaced with the contraction node, we use the parent relationship. The parent of the contracted nodes is the contraction node. Arcs can have either contraction nodes or other arcs as parent. When we apply rules on the Petri net we use only the top-most Petri net, which is the net with only the nodes and arcs that do not have a parent.

With the parent relationship, we can define some useful functions, like the ancestors function, which returns all ancestors of a node (definition 3.4).

**Definition 3.4** (Ancestor relations). Let $CPN = (P, T, D, F, p, \{L_1, \ldots, L_h\})$ be a contracted Petri net with $v \in P \cup T \cup D \cup F$, the ancestors function $p^*$ is defined by $p^*(v) = p(v) \cup \{p^*(p(v)) : p(v) \neq \emptyset\}$

We can of course retrieve nodes in the other direction of the parent relationship, like for the child function, definition 3.5.

**Definition 3.5** (Child relations). Let $CPN = (P, T, D, F, p, \{L_1, \ldots, L_h\})$ be a contracted Petri net with $v \in P \cup T \cup D \cup F$, the child function $p^\dagger$ that returns all nodes that have $v$ as parent node is defined by $p^\dagger(v) = \{x : x \in P \cup T \cup D \cup F : p(x) = v\}$

And we can also define a descendant function, which returns all descendants of a node, definition 3.6.
Definition 3.6 (Descendants relations). Let $CPN = (P, T, D, F, p, \{L_1, ..., L_h\})$ be a contracted Petri net with $v \in P \cup T \cup D \cup F$, the descendant function $p^*$ is defined by $p^*(v) = p^1(v) \cup \{p^*(x) : x \in p^1(v)\}$

Dummy nodes

Dummy nodes are placed on arcs to route them around the nodes. A dummy node is a combination of the arc and a layer in which it will be placed. In the layout the dummy nodes will be the bending points of the arcs, therefore the dummy nodes are only placed on original arcs. An original arc is an arc of the initial Petri net, so those arcs which do not have children (definition 3.7). So, $\forall (f, l) \in D : p^1(f) = \emptyset$.

Definition 3.7 (Original Arc). Let $CPN = (P, T, D, F, p, \{L_1, ..., L_h\})$ be a contracted Petri net. An arc $(u, v) \in F$ is called an original arc, if it has an empty child relation, i.e. $p^1((u, v)) = \emptyset$

Definition 3.8 gives two recursive functions to retrieve the original arcs, one for a set of arcs as input, the other for a single arc as input.

Definition 3.8 (getOriginalArcs). Let $CPN = (P, T, D, F, p, \{L_1, ..., L_h\})$ be a contracted Petri net, $F'$ be a set of arcs with $F' \subset F$ and $f \in F$ be an arc. Then:

$CPN.getOriginalArcs(F') = \bigcup\{CPN.getOriginalArcs(f') : f' \in F'\}$

$CPN.getOriginalArcs(f) = \begin{cases} \{f\}, & \text{if } p^1(f) = \emptyset \\ CPN.getOriginalArcs(p^1(f)), & \text{otherwise} \end{cases}$

Algorithm 9 creates dummy nodes. It creates a node on the arc $f$ in layer with number $l$ and adds the node into the Petri net and the layering.

Algorithm 9: CREATE DUMMY NODE $\langle CPN, f, l \rangle$

Input: Contracted Petri Net $CPN = (P, T, D, F, p, \{L_1, ..., L_h\})$

Input: The arc $f$ that the dummy node belongs to

Input: The layer number $l$ in which the dummy node should be placed

Output: The created dummy node $d$

1. $d \leftarrow (f, L_l)$
2. $D \leftarrow D \cup \{d\}$
3. $L_l \leftarrow L_l \cup \{d\}$
4. return $d$

Figure 3.7 shows an example of the extra information of the Petri net, before and after a contraction.
Petri net layout algorithm

Definition 3.9 (getMaxLayer and getMinLayer) Let $CPN = (P, T, D, F, p, \{L_1, \ldots, L_h\})$ be a Contracted Petri Net, and $V$ be a set of nodes, with $V \subseteq P \cup T \cup D$, $v$ a node, with \( v \in P \cup T \cup D \) and $(u, w) \in F$ an original arc. Then:

$CPN.getMaxLayer(v) = (\uparrow i : v \in L_i : i)$.  
$CPN.getMaxLayer(V) = (\uparrow v \in V : CPN.getMaxLayer(v))$.  
$CPN.getMaxLayer((u, w)) = (CPN.getMaxLayer(u) \uparrow CPN.getMaxLayer(w))$.  
$CPN.getMinLayer(v) = (\downarrow i : v \in L_i : i)$.  
$CPN.getMinLayer(V) = (\downarrow v \in V : CPN.getMinLayer(v))$.  
$CPN.getMinLayer((u, w)) = (CPN.getMinLayer(u) \downarrow CPN.getMinLayer(w))$.  

### 3.2.2 Contraction driver algorithm

To perform the contraction rules we use the driver Algorithm 10, which deals with the recurrent execution of the rules on the Petri net and calls the rule-algorithm 11, which executes the rule itself.
Algorithm 10: Contraction-Driver($PN, R$)

Input: Contracted Petri Net $CPN = (P, T, D, F, p, \{L_1, ..., L_h\})$

Input: List of Reduction rules $R$

Result: Contracted Petri net $PN$

1. $orderNumber \leftarrow 0$
2. $ContractionRuleApplied \leftarrow true$
3. while $ContractionRuleApplied$ do
    4. $V \leftarrow DefineNodeOrder(CPN)$ // $V$ contains only the top level nodes
    5. $ContractionRuleApplied \leftarrow false$
    6. foreach $rule \in R$ do
        7. foreach $v \in V$ do
            8. $ContractionRuleApplied \leftarrow ApplyContractionRule(CPN, rule, v, orderNumber)$
            9. if $ContractionRuleApplied$ then
                10. $orderNumber \leftarrow orderNumber + 1$
                11. break // For Each of line 7
            end
        end if $ContractionRuleApplied$ then
            13. break // For Each of line 6
        end
    end
4. end

The contraction driver gets the order of the contraction rules $R$ in which they should be applied. Important rules will be applied before less important rules. Different orderings of the rules give different results. More about the order is described in Chapter 4. First the algorithm determines an order of the nodes, in which they are investigated by the rule-algorithm (line 4). This order also determines which part of the Petri net will be investigated first. Chapter 4 contains more information about this order.

The algorithm takes the first rule and walks though the nodes in the defined order, and tries to apply the rule on a node. Only the top level nodes, i.e. $\{v \in P \cup T : p(v) = \emptyset\}$ are investigated. If a rule is applied, the algorithm starts again by creating an order of the nodes. This is necessary, since some new nodes are created and some nodes are no longer topmost nodes. It also starts by investigating the first rule again, because the application of a rule can enable a more important rule.

The algorithm also keeps track of an order number in which the nodes will be contracted, this is necessary for the expansion.

If all the rules are investigated and no rules can be applied anymore, the algorithm stops and the output is the Petri net contracted to the minimum number of nodes. The Petri net is then said to be a contracted Petri net. Since all the rules decrease the number of nodes and arcs, this algorithm will always stop.
### 3.2.3 Contraction rule algorithm

Algorithm 11 is the algorithm the driver uses to apply a rule.

**Algorithm 11: Contraction-Rule(CPN, rule, v, orderNumber)**

**Input:** Contracted Petri Net $CPN = (P, T, D, F, p, \{L_1, ..., L_h\})$

**Input:** Contraction rule $rule$

**Input:** Node $v$

**Output:** A boolean value that indicates whether or not the rule is performed

**Result:** If the rule is applicable; nodes from $PN$ are contracted; otherwise $PN$ is unchanged

1. $V_N \leftarrow \{v, v\bullet, v\bullet\bullet\bullet v, v\bullet\bullet\bullet (v\bullet), (v\bullet), (v\bullet)\}$
2. if $rule.pre(v, V_N)$ then
   3. $V_R \leftarrow rule.getNodes(v, V_N)$ // Get only the contractible nodes
   4. Create a new contraction node $v_c$
   5. $v_c.orderNumber \leftarrow orderNumber$
   6. Add $v_c$ to $CPN$
   7. PUTNODEINTOLAYERS($CPN, V_R, v_c$)
   8. CREATEDUMMYNODES($CPN, V_R, v_c$)
   9. CALCULATESIZE($CPN, V_R, v_c$)
10. CONTRACT($CPN, V_R, v_c$)
11. return true
12. else
13. return false

First it checks whether or not the rule is applicable on the input node, and the neighborhood of the input node (line 2). If it is not the algorithm returns false. If it is applicable we retrieve those nodes (line 3) and contract them.

First we make a new node, which will be the contraction node $v_c$. For this node we set the order number, this is necessary for the expansion. We place this node into the right layers (line 7). There is a possibility that the contraction causes crossings with the arcs, therefore we create dummy nodes for those arcs (line 8). After we calculate the right size of the contraction node (line 9) we contract the nodes, by updating the parent relationship (line 10). Finally, we return the value true, to indicate that a contraction has been performed.

**Put the contraction node into the layers**

Algorithm 12 puts the contraction node $v_c$ into the right layers.

We first retrieve the minimum and maximum layer numbers of the nodes $V_R$ that are being contracted and place the contraction node $v_c$ in all the layers between and including these numbers.
Algorithm 12: PutNodeIntoLayers(\(CPN, V_R, v_c\))

**Input:** Contracted Petri Net \(CPN = (P, T, F, D, p, \{L_1, ..., L_h\})\)

**Input:** Set of nodes \(V_R \subseteq P \cup T\) that will be contracted

**Input:** Contraction node \(v_c\)

**Result:** Contraction node \(v_c\) is added to the layers \(\{L_1, ..., L_h\}\) of \(CPN\)

1. \(min \leftarrow CPN.getMinLayer(V_R)\)
2. \(max \leftarrow CPN.getMaxLayer(V_R)\)
3. for \(i \in [min, max]\) do
4. \(L_i \leftarrow L_i \cup \{v_c\}\)
5. end

Create dummy nodes

When we contract nodes we should also add dummy nodes for some arcs to prevent them from crossing. To understand this part better, one should have a better view on the rules, therefore this part of the algorithm is described in section 4.3 of the rules chapter.

Contraction of the nodes

Now that we have calculated all the information we can connect the contraction node into the Petri net. First we create new arcs that will replace the arcs between the contracted nodes \(V_R\) and the other nodes of the net, with the contraction node and the other nodes of the net (lines 1 3). Then we create the new parent relationships which replace the old ones.

First the contracted nodes will get the contraction node as parent (line 6). The other nodes will preserve their own relationship. Second the arcs will get new parent relationships (line 8). If the arc is connected between two contracted nodes, the parent will be the contraction node. If the arc is connected between a contracted node and another node, the arc will get the corresponding arc as parent, which we created earlier. Other arcs will preserve their old parent.

Calculate Size

Algorithm 14 calculates the size of a node. This size is important because we should be able to place the contracted nodes into the region of the contraction such that they do not overlap.

We can calculate the width of the node, via the layer numbers. We subtract the minimum layer from the maximum layer and multiply this with the series offset (line 3). The series offset is the predefined distance between two layers (see Figure 3.8). The nodes are aligned on the layers, therefore we should add half the original node size for the first and last layer.
Algorithm 13: Contract($CPN, V_R, v_c$)

**Input:** Contracted Petri Net $CPN = (P, T, D, F, p, \{L_1, ..., L_h\})$

**Input:** Set of nodes $V_R \subseteq P \cup T$ that will be contracted

**Input:** Contraction node $v_c$ that will replace the nodes of $V_R$

**Result:** The Petri Net $PN$, with the contraction node $v_c$ connected, and the contracted nodes removed

1. $F \leftarrow F \cup \{(v_c, v)\}$
2. $F \leftarrow F \cup \{(u, v)\}$
3. $F \leftarrow F \cup \{(u, v)\}$

\{Create the new parent relations for the nodes\}

4. $\forall v \in P \cup T \cup D$
5. $p'(v) \leftarrow \begin{cases} v_c, & \text{if } v \in V_R \\ p(v), & \text{otherwise} \end{cases}$

\{Create the new parent relations for the arcs\}

6. $\forall (u, v) \in F$
7. $p'((u, v)) \leftarrow \begin{cases} v_c, & \text{if } (u, v) \in V_R \times V_R \\ (v_c, v), & \text{if } (u, v) \in V_R \times (V \setminus V_R) \\ (u, v), & \text{if } (u, v) \in (V \setminus V_R) \times V_R \\ p((u, v)), & \text{otherwise} \end{cases}$

\{Add the new parent relations to the Petri Net\}

8. $p \leftarrow p'$

To retrieve the height we calculate the height of the contracted nodes on each layer, and take the maximum of it. We assume that the original nodes have a height defined in $v.height$. The dummy node has also a defined height, which is the height of an original node. The height on one layer is calculated by adding the height of all the child nodes from that layer together with the needed number of parallel offset(s). The parallel offset is the predefined minimum distance between two nodes of the same layer (see Figure 3.8).

We also calculate a maximum level number, see line 10. This is the maximum number of nodes on one layer. We use this number later on.

Note that the series offset is from center to center and the parallel offset from top to bottom. This is because the widths of the nodes are determined by the layering, which are static values and the parallel offset is determined by the nodes, which can have different sizes.
Algorithm 14: CalculateSize($CPN, v$)

**Input**: Contracted Petri Net $CPN = (P, T, D, F, p, \{L_1, ..., L_h\})$

**Input**: Node $v$ from which the size will be updated.

**Result**: Node $v$ has the right size

1. $maxLayer \leftarrow CPN.getMaxLayer(v)$
2. $minLayer \leftarrow CPN.getMinLayer(v)$
3. $v.width \leftarrow (maxLayer - minLayer) \cdot \text{SeriesOffset} + \text{NodeWidth}$
4. $maxHeight \leftarrow 0$
5. $maxLevel \leftarrow 0$
6. for $i \leftarrow minLayer$ to $maxLayer$ do
   7. $V_i \leftarrow \{x \in p^1(v) : x \in L_i\}$
   8. $height \leftarrow \Sigma(x \in V_i : x.height)$
   9. $height \leftarrow height + (((\#x \in V_i) - 1) \cdot \text{ParallelOffset})$
   10. $maxLevel \leftarrow maxLevel \uparrow (\#x \in V_i)$
   11. $maxHeight \leftarrow maxHeight \uparrow height$
7. end
8. $v.height \leftarrow maxHeight$

![Figure 3.8](image.png)

Figure 3.8: The different offsets uses to calculate the sizes of the nodes.

### 3.3 Lay-out top level Petri net

The input of this step is a contracted petri net $CPN$, on which no rule can be applied anymore. For this Petri net we create a lay-out for top level Petri net. Figure 3.9 shows a schematic overview of this step of the algorithm. To create such layout we use the top level nodes, and the original arcs of the top level arcs. Those original arcs are important, because they contain the precise layer information, and top level arcs can contain more original arcs, which all should be placed. In this step we also create dummy nodes for those arcs, and as known, dummy nodes may only be placed on original arcs.

![Figure 3.9](image.png)

Figure 3.9: The schematic representation of the layout creation of the top level Petri net.
The layout for the top level Petri net is performed with the same steps as for the layered drawings of digraphs method. First of all we should create a layering. For this we can use the layer numbers we already defined. Due to the fact we have nodes laying in more than one layer, we can omit certain layers. Therefor we use only the important layers. Before we can introduce those important layers some more knowledge is needed on the important layers of the arcs, this is described in Subsection 3.3.1.

When we have those important layers, we should make the layering proper, therefore we also should add dummy nodes to the arcs. This layering approach is described in section 3.3.2.

The second step is the crossing reduction. We use an adapted version of the barycentric crossing reduction method, because we can have nodes that are laying on more than one layer, more about this adapted crossing reduction in subsection 3.3.2.

Finally we can place the nodes, by giving them the absolute X and Y coordinates, which is described in subsection 3.3.4.

### 3.3.1 Important arc layers

Before we start with the important layering, some more understanding about the arcs is needed. Due to the fact that we have contraction nodes, we can have arcs which are connected to that node, but not to the bound of the node. Take for example the arc \((t_{12}, p_{22})\) in Figure 3.10. Here the arc is connected to the node \(p_{22}\), which is contracted into the contraction node \(p_{c5}\). Node \(p_{22}\) is situated in layer 13, which is the right bound of the node, but that’s not the correct bound for that arc. The arc is going to the left from the node, so the correct bound should be the left bound. So there should be a dummy node on layer 12 to route the arc around the node. This layer for that arc is called the node exit route layer.

Another thing we should take into account is the set of dummy nodes we added to the arcs by the contraction. (See line 8 of the CONTRACTION-RULE algorithm 11 and the description in section 4.3). In our example the arc has a dummy node on layer 8, in node \(t_{12}\). So the layer the arc is leaving the node is layer 8 and not the minimal layer of the arc. This layer for that arc is called the node exit layer. And the corresponding node exit route layer number should be 9.

The important layers for an arc are the node exit layers and the node exit route layers, which only exist when the arc has a span bigger than 1 and the node exit layers isn’t the correct bound of the nodes.
We defined two different functions to get those layer numbers, one for the left node and one for the right node. We only give the functions for the left node, the functions for the right can be trivially obtained due to the symmetry.

Definition 3.10 returns the node exit route layers and node exit layers for the left node. This function uses 3 subfunctions.

**Definition 3.10 (getImportantArcLayersLeft).** Let \( CPN = (P, T, D, F, p, \{L_1, \ldots, L_h\}) \) be a contracted Petri net and \( f \in F \) be an original arc. And we retrieve:

1. \( v_l \leftarrow CPN.getLeftNode(f) \) Get left original node.
2. \( v_{tl} \leftarrow CPN.getTopLevelNode(v_l) \) Get left top node.
3. \( i_{el} \leftarrow CPN.getNodeExitNumberLeft(f, v_{tl}) \) Get the exit layer number.

Then:

\[
CPN.getImportantArcLayersLeft(f) = \begin{cases} 
    \{i_{el}, i_{el} + 1\}, & \text{if } i_{el} < CPN.getMaxLayer(v_l) \\
    \{i_{el}\}, & \text{otherwise}
\end{cases}
\]

**Sub function 1: Get left original node** First we should retrieve the left node of the arc. This is trivially done by checking the layer numbers, see definition 3.11

**Definition 3.11 (getLeftNode).** Let \( CPN = (P, T, D, F, p, \{L_1, \ldots, L_h\}) \) be a contracted Petri net and \( (u, v) \in F \) be an original arc. Then:

\[
CPN.getLeftNode((u, v)) = \begin{cases} 
    u, & \text{if } CPN.getMaxLayer(u) < CPN.getMaxLayer(v) \\
    v, & \text{otherwise}
\end{cases}
\]
In our example we have $\text{CPN.getMaxLayer}(t_{12}) < \text{CPN.getMaxLayer}(p_{22}) = 6 < 13$, which is true, so the left node is node $t_{12}$.

**Sub function 2: Get left top node** From that node, we should retrieve the top level node. This is done by taking recursively the parent of the node, until a node has no parent, this node then is the top level node. The $\text{getTopLevelNode}$ function is defined in definition 3.13.

**Definition 3.12** (getTopLevelNode). Let $\text{CPN} = (P, T, D, F, p, \{L_1, \ldots, L_h\})$ be a contracted Petri net and $v \in P \cup T$ be a node of $\text{CPN}$. Then $\text{CPN.getTopLevelNode}(v) =$ \begin{align*} &\begin{cases} v, & \text{if } p(v) = \emptyset \\ \text{CPN.getTopLevelNode}(p(v)), & \text{otherwise} \end{cases} \end{align*}

In our example we get $\text{CPN.getTopLevelNode}(t_{12}) = tc_6$.

**Sub function 3: Get the exit layer number** Now we can retrieve the layer number in which arc $f$ leaves the top level node $v_{tl}$. First we retrieve the set of dummy nodes of $f$, which are situated under the top level node $v_{tl}$, which is $D_f = \{(f, L_i) \in p^*(v_{tl})\}$. If $D_f$ is not empty (there are dummy nodes) we take the maximum layer number of those dummy nodes. If there aren’t dummy nodes we take the minimum layer number of arc $f$.

**Definition 3.13** (getNodeExitNumberLeft). Let $\text{CPN} = (P, T, D, F, p, \{L_1, \ldots, L_h\})$ be a contracted Petri net and $f \in F \cup T$ be an original arc, with $v \in P \cup T$ the top level left node.

We define $D_f = \{(f, L_i) \in p^*(v)\}$ as the set of dummy nodes of original arc $f$ under $v$, then:

$\text{CPN.getNodeExitLayerLeft}(f, v) =$ \begin{align*} &\begin{cases} (\uparrow j : (f, L_j) \in D_f), & \text{if } D_f \neq \emptyset \\ \text{CPN.getMinLayer}(f), & \text{otherwise} \end{cases} \end{align*}

In our example we get for $D_f = \{((t_{12}, p_{22}), 8)\}$, so $\text{CPN.getNodeExitLayerLeft}((t_{12}, p_{22}), tc_6) = 8$.

**Retrieving the layer numbers** Now that we have the node exit layer $i_{el}$ we can check if this is on the boundary or not, by checking it with the maximum layer number of the top level node $v_{tl}$. If $i_{el}$ is smaller, then we need a node exit route layer number, which is $i_{el} + 1$, and return both values. Otherwise, there is no need for a node exit route layer number and we return only the node exit layer.

In our example we get $\text{CPN.getImportantArcLayersLeft}((t_{12}, p_{22})) = \{8, 9\}$. 

**Important arc layers**  By combining the functions, we create one function that gives all the important layers of a certain arc back, see definition 3.14. If the difference between the minimum and maximum layers of the arc is 1, then only the minimum and maximum layers of the arc are the important layers, otherwise the node exit layer numbers and the possible node exit route layer numbers.

**Definition 3.14** (getImportantArcLayers). Let $CPN = (P, T, D, F, p, \{L_1, \ldots, L_h\})$ be a contracted Petri net and $f \in F$ be an original arc. And we retrieve:

- $\text{min}_f \leftarrow CPN.\text{getMinLayer}(v_l)$ Get minimum layer number of arc $f$.
- $\text{max}_f \leftarrow CPN.\text{getMaxLayer}(v_l)$ Get maximum layer number of arc $f$.
- $\text{I}_{\text{ill}} \leftarrow CPN.\text{getImportantArcLayersLeft}(f)$ Get important layers left.
- $\text{I}_{\text{ilr}} \leftarrow CPN.\text{getImportantArcLayersRight}(f)$ Get important layers right.

Then:

$$\text{CPN.importantArcLayers}(f) = \begin{cases} \text{I}_{\text{ill}} \cup \text{I}_{\text{ilr}}, & \text{if } \text{max}_f - \text{min}_f > 1 \\ \{\text{min}_f, \text{max}_f\}, & \text{otherwise} \end{cases}$$

### 3.3.2 Create important layering

Before we can do the crossing reduction we should make the layering proper. At this moment we have arcs and nodes crossing more than one layer. In the next section we describe the way to make the layering proper regarding the nodes. To make the layering proper regarding the arcs we add dummy vertices on the layers they cross. Due to the fact that we have contraction nodes, we can have layers, which do not differ from their adjacent layers. By omitting these layers, the crossing reduction is performed a smaller graph and we add less dummy nodes. Therefore we introduce the important layers, definition 3.15.

**Definition 3.15** (Important layer). Let $CPN = (P, T, D, F, p, \{L_1, \ldots, L_h\})$ be a contracted Petri net. A layer $L_i$ is an important layer if:

1. A top level node starts or ends in that layer, i.e. $(\exists v \in P \cup T : p(v) = \emptyset \land (CPN.\text{minLayer}(v) = i \lor CPN.\text{maxLayer}(v) = i))$

2. The layer is an important layer for an original arc, i.e. $(\exists f \in CPN.\text{getOriginalArcs}(F) : i \in CPN.\text{getImportantArcLayer}(f))$

Take for example the part of the Petri net of Figure 3.11, here the important layers are: 6, 7, 10, 11, 14, 15, 16, 17 and 18, so layers 8, 9, 12 and 13 can be omitted. Algorithm 15 creates the important layers, and adds dummy nodes for each arc, which crosses more than one important layer.
Algorithm 15: CreateImportantLayering(CPN)

Input: Contracted Petri Net CPN = (P, T, D, p, {L₁, ..., L_h})

Output: A set \{Iₘ, ..., Iₙ\} ⊆ \{L₁, ..., L_h\} containing only the important layers

Result: All original arcs of the top level arcs has, if needed, dummy nodes on the important layers

1. \( N_{top} \leftarrow \{v : v \in P \cup T : p(v) = \emptyset \} \)
2. \( F_{org} \leftarrow \bigcup \{CPN.getOriginalArcs(f) : f \in F : p(f) = \emptyset : \} \)
   \{Create a set with the important layer numbers\}
3. \( Im \leftarrow \bigcup \{x \in N_{top} \cup F_{org} : \{CPN.getMinLayer(x)\} \cup \{CPN.getMaxLayer(x)\} \} \)
   \{Create dummy nodes, on the arcs in the important layers\}
4. \( Im \leftarrow Im \cup \{f \in F_{org} : CPN.getImportantArcLayers(f) \} \)
5. foreach \( f \in F_{org} \) do
6.     \( \min_f \leftarrow CPN.getMinLayer(f) \quad \max_f \leftarrow CPN.getMaxLayer(f) \)
7.     foreach \( i \in (\min_f, \max_f) \cap Im \) do
8.         \( d \leftarrow \text{CreateDummyNode}(CPN, f, i) \)
9. return \{\( L_i : L_i \in \{L₁, ..., L_h\} : i \in Im \}\}

It first retrieves the important layers, therefore we create a set \( N_{top} \) with the top level nodes (line 1), and a set \( F_{org} \) with the original arcs of the top level arcs (line 1). From these sets we built a set with the important layer numbering. For each node of \( N_{top} \), we retrieve the minimum and maximum layer number (line 3), and for each arc from \( F_{org} \) we retrieve the important arc layer numbers, as described in the previous section (line 4).

Now we add for each arc on each important layer between and excluding the minimum and maximum layers of those arcs a dummy node (lines 5..7).

The output of the algorithm is the set of the important layers.

Figure 3.12 is the Petri net of Figure 3.11 after the execution of algorithm 15.
3.3.3 Leveled barycentric crossing reduction

For the crossing reduction, we created our own *leveled barycentric crossing reduction*, which is an adapted version of the barycentric crossing reduction method of Sugiyama et al. [13]. First of all, we have to deal with nodes lying on more than one layer, which is not allowed for the crossing reduction. The solution for this is to (imaginary) split up the nodes placed on more layers into *temporary nodes* for each layer, with a connection between them. We use a higher value for a connection between a contraction node of two layers than for other connections. This is the so called *cost* function: \( \text{cost}: F \rightarrow \mathbb{N} \). This ensure that the cost of crossing a contraction node is higher than other crossings. Therefore we must add the cost to the calculation of crossings, this is done by multiplying the cost of both arcs by each crossing.

Take for example the part of the contracted Petri net of Figure 3.13. In the top there is a representation with only one crossing, but this is a crossing with the contraction node \( t_5 \). If \( t_5 \) contains a lot of arcs between layer 5 and 6, the number of crossings on the expanded Petri net is more than 1. In the bottom the same part of the Petri net is shown, but now with two crossings, but only with arcs. The number of crossings will stay 2 in the expanded Petri net, assuming the contraction nodes don’t contain internal crossings.

When we add the cost of 10 to the connections of the temporary nodes, we get the following number of crossings for the top net: \( (p_{19}, t_{17}) \) with \( (t_{15}, t_{17}) = 1 \times 10 \), and for \( (t_{15}, t_{17}) \) with \( (p_{19}, t_{17}) = 10 \times 1 \), which makes the total of 20, so the number of crossings with the cost is 20/2 = 10. And for the bottom net the number of crossings with the cost is 2. So, in the crossing reduction the bottom situation is stored as better result than the top situation. An investigation can be done to get the right number for the cost, see also section 6.2.2.
The other adaption is the introduction of level numbers. The temporary nodes of the contraction nodes should be placed after each other on the same Y coordinate, because after the crossing reduction the whole node should be placed back. Also no two nodes may be crossing each other. Therefore we add a level number to the crossing reduction. Each layer gets the same number of levels, on which nodes can be placed. This level number is determined by the layer with the highest number of nodes, i.e. \( \text{levelNumber} = \left( \uparrow : I \in \{I_m, \ldots, I_n\} : |I| \right) \). Each node of a layer is assigned to a level number. For the levels with no nodes we add an empty node to it, this is just a node with no connection to the graph.

The calculation of the barycenters should also use the cost of the connections. Formula 3.1 is the barycenter formula including the cost.

\[
B^i(v) = \frac{\sum_{w \in V_i} w.y \cdot \text{cost}(w, v)}{\sum_{w \in V_i} \text{cost}(w, v)} \quad (3.1)
\]

To the existing phases of the algorithm we add some extra restrictions and we introduce one extra phase.

**Phase 1**

The ordering on the barycenters is performed only on dummy nodes and the nodes
starting at that layer. Contraction nodes from that layer that start on other layers, will be omitted from the sortation and they stay on their level number. For example, in Figure 3.14 we have four levels, with on level 3 an unmovable contraction node. The corresponding barycenters are (2, 1, 3, 1.5), by the sortation on those barycenters where level 3 is omitted we get the following order (1, 1.5, 3, 2), see the right hand side.

<table>
<thead>
<tr>
<th>level number</th>
<th>barycenters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>1.5</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>level number</th>
<th>barycenters</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>1.5</td>
</tr>
<tr>
<td>3</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>2</td>
</tr>
</tbody>
</table>

Figure 3.14: An example of the sortation on barycenters, with fixed levels.

Phase 2
The reversion on the barycenters is also performed only on dummy nodes and the nodes starting at that layer. Contraction nodes from that layer that start on other layers, will be omitted from the reversion and they stay on their level number. With the reversion we also take the empty nodes into account, which will be reversed with the adjacent nodes.

Phase 3: Leveling
All the contraction nodes should be placed on the same level, therefore we introduce a leveling phase that places the temporary nodes of a contraction node to the same level. It takes the temporary nodes and checks the level number with the level number of the temporary node on previous (or next, determined from the direction) layer. If this node on this layer has another level number, it trivially switches the node with the node on the correct level number.

Phase 3 is performed at the start of the crossing reduction, because the nodes can initially lay on different level numbers. And it’s executed right after each ordering or reversing step of phases 1 and 2, if a contraction node is involved (that contraction node gets an different level).
Running example

Figures 3.15 and 3.16 show an running example of the leveled barycentric crossing reduction.

We start with an initial graph:

On this graph we start with the level the nodes (Start of phase-3-right). The temporary nodes of B will be placed on level 2, the nodes of D on level 3

After the leveling, we start with phase-1-right. On layer 5, the dummy node and node D will be ordered:

Node D is a contraction node, so phase-3-right will be executed now:

3.3.4 Positioning nodes

The positioning of the nodes is trivial done on a grid, by algorithm 16.
After phase-3-right, the barycenters are all ordered, so phase-1-right is ended. We start with phase-2-right, and on layer 3 node B and the dummy node will be reversed.

Node B is a contraction node, so phase-3-right will be executed now:

After phase-3-right, phase-1-right (part of phase-2) will be executed, and it will order node C and the dummy node on layer 3:

After this node D and the dummy node on layer 4, will be reordered, and phase-3-right is performed:

The other nodes are ordered on the barycenters, so phase-1-right will end. The number of crossings now is 0, so the algorithm ends.

We already have the X coordinates of the nodes, which are determined by the layers. The relative Y coordinate is also determined by the level numbers of the crossing reduction. We should now calculate the absolute y-coordinates, this is done by retrieving for each level number the highest height (line 7). By placing these heights together, with the
Algorithm 16: PositioningNodes(CPN)

Input: Contracted Petri Net CPN = (P, T, D, F, p, {L_1, ..., L_h})
Result: The top level nodes of CPN has its position

1 \text{maxLevelNumber} \leftarrow (\uparrow: L \in \{L_1, ..., L_h\} : |\{v \in L : p(v) = \emptyset\}|)
2 LevelHeights[] \leftarrow \text{new array} (\text{maxLevelNumber})
3 \text{foreach} L \in \{L_1, ..., L_h\} \text{ do}
4 \quad \text{foreach} v \in L : p(v) = \emptyset \text{ do}
5 \quad \quad \text{LevelHeights}[v.\text{level}] \leftarrow \text{LevelHeights}[v.\text{level}] \uparrow v.\text{height}
6 LevelYPositions[] \leftarrow \text{new array} (\text{maxLevelNumber})
7 LevelYPositions[1] \leftarrow 0
8 \text{foreach} i \in (1, \text{maxLevelNumber}] \text{ do}
9 \quad \text{foreach} v \in L : p(v) = \emptyset \text{ do}
10 \quad \quad LevelYPositions[i] \leftarrow \text{LevelYPositions}[i - 1] +
11 \quad \quad \text{LevelHeights}[i - 1] +
12 \quad \quad ((i - 1) \cdot \text{ParallelOffset})
13 \quad \text{foreach} v \in P \cup T \cup D : p(v) = \emptyset \text{ do}
14 \quad \quad \text{min_v} \leftarrow \text{CPN.getMinLayer}(v)
15 \quad \quad v.x \leftarrow ((\text{min_v} - 1) \cdot \text{SeriesOffset}) - (\text{NodeWidth}/2)
16 \quad \quad v.y \leftarrow \text{LevelYPositions}[v.\text{level}] +
17 \quad \quad ((\text{LevelHeights}[v.\text{level}] - v.\text{height})/2)

right amount of parallel offsets, we retrieve the absolute Y value of the start of each level (line 16). Note that the Y direction is from top to bottom.

Now we can place each node. First we calculate the X coordinate. This is the X coordinate of the minimum layer number of the node minus half the node width of an original node, because the nodes are aligned on the layer. The X coordinate of a layer is trivially retrieved by multiplying the layer number minus one with the series offset (line 19). For example the node \( t \) of Figure 3.17, the X coordinate is \((7 - 1) \cdot 50) - (25/2) = 287.5.

Second we calculate the absolute Y coordinate, due to the fact that the level height is determined on the highest node on that level, we align the nodes in the middle of the level. Therefore we calculate the remaining space of the level, by taking the height of the level minus the height of the node. Half the remaining space should be placed above the node, so we add this to the Y coordinate of the level and we have the absolute Y coordinate (line 20). In the example node \( t \) of Figure 3.17, the Y coordinate is \( 50 + ((75 - 25)/2) = 75 \).
3.4 Expansion

The input of this step is a contracted Petri net $CPN$ for which the top level net has a lay-out. In this final step we create the layout for the bottom level petri net, which is the original net. Figure 3.18 shows a schematic overview of this step of the algorithm. On the top level layout we iteratively replace contraction nodes, by its contracted nodes. To perform this we first do a crossing reduction on the contracted nodes an their connected neighborhood, secondly we place the nodes in the area of the contraction node.

3.4.1 Expansion driver

First we describe the driver algorithm 17, which deals with the recurrent expansion of a contraction node of the Petri net. If there are still contraction nodes in the net, the algorithm takes the node with the highest order number (line 2). Its important for the expansion to expand the nodes in the opposite direction of the contractions. Otherwise we can loose connections while expanding nodes. For example if we have an arc $(t1, p2)$ and the node $t1$ is first contracted in node $A$, and $p2$ is contracted later in $B$, we have the arcs $(t1, p2)$, $(A, p2)$ and $(A, B)$, with $p((t1, p2)) = (A, p2)$, $p((A, p2)) = (A, B)$ and $p((A, B)) = \emptyset$. If we first expand $A$, we remove node $A$ and the arc $(A, B)$, and we have no arc $(t1, B)$. 
We also take the arcs connected to the contraction node and the rest of the Petri net (line 3). These arcs are used for the crossing reduction.

When we have all the info of the contraction node, we perform a crossing reduction on the child nodes (5). After this we have the relative places of the child nodes, so we can place them absolute, in the area of the contraction node (6). Finally, we remove the contraction node, and with that also the arcs connected to that node, and the parent relations to that node.

Algorithm 17: Expansion($CPN$)

| Input: Contractions Petri Net $CPN = (P, T, D, F, p, \{L_1, ..., L_h\})$ |
| Input: Contraction node $v_c$ |
| Result: The Petri Net $CPN$, with the contraction node $v_c$ expanded |

1 while $\exists v \in P \cup T : p(v) = \emptyset \land p^{-1}(v) \neq \emptyset$ do |
2 $v_c \leftarrow v \in P \cup T : v.orderNumber = (\uparrow x \in P \cup T : x.orderNumber)$ |
3 $F_c \leftarrow \{(v_c, u) \in F\} \cup \{(u, v_c) \in F\}$ |
4 $SubCPN \leftarrow CREATESubCPN(p^{-1}(v_c), F_c)$ |
5 $PERFORMCROSSINGREDUCTION(SubCPN)$ |
6 $PLACENODES(p^{-1}(v_c))$ |
7 end

In the next sections we discuss the global execution of the crossing reduction and the placement of the nodes. There are some specific properties we use for these methods, so a precise description is better to understand if one has a better view on the rules. These precise descriptions are situated in section 4.4 of the rules chapter.

3.4.2 Crossing reduction

To minimize the crossing, we perform the same leveled crossing reduction as described in section 3.3.3, with one adaption to it. We are now performing a crossing reduction, with nodes connected to other nodes in the Petri net, the neighborhood. We won’t place these neighborhood nodes, because they are already placed, or will be placed with another expansion. But these connections should be taken into account. So these connections are present in the sub-net we’ll perform crossing reduction on, but they are fixed in the ordering, reversing and leveling. This is trivially done, because these nodes are situated around the contracted nodes, and never in between them. Therefore we can simply order the nodes in the way we do for the leveled crossing reduction crossing.
Take for example Figure 3.19a. Here we have an expansion with two levels and 5 connections to the neighborhood, represented by the stars. First we level it, so we get the sub-net of Figure 3.19b. We start now with phase-1-right. The start of node $p_6$ on layer $L_2$ is also connected with the two nodes of the neighborhood, so these nodes are also used to calculate the barycenters, which makes the barycenter $B^2(p_6) = 2$. It is the start of node $p_6$, so node $p_6$ and the dummy node can be reordered. After the reordering we should level the nodes again. Then we get the outcome of Figure 3.19c. This is a situation with one crossing, and it will be the best situation, so after the crossing reduction this situation will be the result.

Now we see also an example of contradicting aesthetics. In this case the **CROSSING** and **AREA** aesthetics are contradicting. Figure 3.20 shows two layouts of the sub-net of Figure 3.19. In the left one there is less area needed than in the right one, but it costs 1 extra crossing. In the right layouts there aren’t crossings, but the needed area is bigger.

**Create the sub net**

To perform crossing reduction on a part of a petri net we should create a sub net. For this we also need a place to save the fixed points, therefore we create a new Petri net structure: the **sub contracted Petri net** (**SubCPN**) (definition 3.16).

**Definition 3.16** (Sub Contracted Petri net). A sub contracted Petri net $\text{SubCPN} = (P, T, D, F, Z, \{L_1, \ldots, L_h\})$ is a Contracted Petri net $\text{CPN} = (P, T, D, F, p, \{L_1, \ldots, L_h\})$ without the parent relationship, but with an additional node set $Z$, containing the fixed points:

$Z \subset (P \cup T \cup D \times L_n)$ is a finite set of fixed nodes.

The parent relationship is not necessary in this structure, because it contains only the information the crossing reduction uses. There is a new kind of node set $Z$ added, containing the fixed point, in the figures represented with the black stars. This node is a cartesian product of a node with a layer. Algorithm 18 creates such a sub net.

First a sub contracted Petri net is made with only the contracted nodes and arcs. This is trivially done by taking the intersection of the sets of the contracted Petri net with the child nodes of the contraction node (lines 1 ... 7). Then we create the fixed points and the connection between the contracted nodes and those points. These points are calculated from the original arcs (line 8), because they contain the precise layer information, and parent arcs can contain more original arcs, which all should be taken into account.

First we check whether or not the arc is moving from the left to the node i.e. the contraction node $v_c$ is the right node of the arc. If it is calculate and place the fixed
point on the previous layer (lines 11 ... 13). Otherwise on the next layer (lines 15 ... 17). We only discuss the calculation and creation of the fixed points on the previous layer number, the calculation and creation of the fixed points on the next layer number can be trivially obtained due to the symmetry.

We retrieve the contracted node $v_r$, where the original arc is connected to, by taking the intersection of the child nodes of the contraction nodes with the set of ancestors of the original node of the original arc (line 11). This node is used to create the connection. We retrieve the layer number on which the arc is connected to the contraction node (line...
12), which is necessary to create the fixed point. We use algorithm 19 to create the fixed point \( z \) (line 17). Then we create a connection between the contracted node and the fixed point. The direction of the connection is not important for the crossing reduction.

**Algorithm 18:** \textsc{CreateSubCPN}(CPN, VR, vc)

\textbf{Input:} Contracted Petri Net \( CPN = (P, T, D, F, p_0, \{L_1, ..., L_h\}) \)

\textbf{Input:} Set of arc \( F_c \) that contains the arcs connected with the contraction node and the other node of the net.

\textbf{Input:} Contraction node \( v_c \) that will be expanded

\textbf{Result:} The contraction node \( v_c \) has dummy-nodes

\begin{verbatim}
1 \( P^* \leftarrow P \cap p^1(v_c) \)
2 \( T^* \leftarrow T \cap p^1(v_c) \)
3 \( D^* \leftarrow D \cap p^1(v_c) \)
4 \( Z^* \leftarrow \{O\} \)
5 \( F^* \leftarrow F \cap p^1(v_c) \)
6 \( \{L_1^* \cap p^1(v_c), ..., L_h^* \cap p^1(v_c)\} \)
7 \text{SubPN} \leftarrow (P^*, T^*, D^*, Z^*, F^*, \{L_1^*, ..., L_h^*\})
8 \text{Fo} \leftarrow CPN.getOriginalArcs(F_c)
9 \text{foreach} \ f \in \text{Fo} \text{ do}
10 \quad \text{if} \ CPN.exitingLeft(f, v_c) \text{ then}
11 \quad \quad \text{vr} \leftarrow x \in p^1(v_c) \cap p^*(CPN.getRightNode(f))
12 \quad \quad \text{cl} \leftarrow CPN.getNodeExitNumberRight(f, v_c)
13 \quad \quad \text{z} \leftarrow CreateFixedPointPrevious(CPN, SubCPN, f, v_c, cl)
14 \quad \text{else}
15 \quad \quad \text{vr} \leftarrow x \in p^1(v_c) \cap p^*(CPN.getLeftNode(f))
16 \quad \quad \text{cl} \leftarrow CPN.getNodeExitNumberLeft(f, v_c)
17 \quad \quad \text{z} \leftarrow CreateFixedPointNext(CPN, SubCPN, f, v_c, cl)
18 \quad \text{F}\^* \leftarrow \text{F}^* \cup (\text{vr}, \text{z})
19 \text{return} \text{SubPN}
\end{verbatim}
Create fixed points

Algorithm 19 is used to create the fixed points. It calculates the point on the arc on the layer $cl - 1$, because on layer $cl$ the arc is connected to the contraction node, and the arc is situated to the left of the dummy node. To calculate this point we first calculate two known points of the arc: one point where the arc is connected to the contraction node $(x_{vc}, y_{vc})$ and one point where the arc is connected to another node of the net $(x_n, y_n)$. Then we can calculate the fixed point on the line of the two points on layer $cl - 1$.

First calculate the point $(x_{vc}, y_{vc})$ of the contraction node (lines 1, 2). For the X coordinate we use the layer number $cl$ on which the arc leaves the node. For the Y coordinate we calculate the middle point of the contraction node.

Then we should retrieve a point on the arc closest to the point of the contraction node. This can either be a dummy node that is already placed, or another node. Only dummy nodes without a parent are the placed dummy nodes. These are the dummy nodes created by the creation of the top level layout, or the nodes placed by other expansion, then the contraction node, with the parent relations is removed and these nodes have also no parents. We are calculating nodes on the left side, so the contraction node is connected to the maximum layer of the arc, so we take the node with the maximum layer number, that is the dummy node closest to the contraction node. We can take the X and Y coordinate of this dummy node as $(x_n, y_n)$ (lines 6,7).

If it hasn’t a placed dummy node, we take the node connected to the arc to calculate $(x_n, y_n)$. For the X coordinate we take the layer number on which the arc left the node, and for the Y coordinate we take, like for the contraction node, the middle point of the node (lines 17,18).

Then we check whether or not a fixed point is defined for that (dummy) node on that layer. If it is, we take that node, otherwise we create one. This is important because for the crossing reduction we should connect multiple arcs with the same point to that point and not to each own point.

Then we calculate the X and Y coordinates of the points, the X coordinate is calculated via the layer number. Note that we subtract two from $cl$, one because we calculate the point on a previous layer, and one for the calculation of the coordinate of the layer via the series offset. We use this X coordinate to calculate the Y coordinate. Notice that in almost all cases the X and Y coordinates are the same as $(x_n, y_n)$, because the adjacent layer of the layer on which an arc leaves the node is indicated as an important layer.
3.4.3 Node placement

The placement of the nodes will be like for the placement of the nodes of the top level net (algorithm 16). Except the nodes we use are now the contracted nodes, and they are now placed in the area of the contracted node $v_c$. Therefore in line 7, we should make the initial Y coordinate the Y coordinate of the contraction node. So line 7 should be:

$$LevelY\ Positions[1] \leftarrow v_c.y.$$ 

There is one exception for a couple of contraction rules, to understand this one should have a better view on the rules, therefore this is described in section 4.4.2 of the rules chapter.
Algorithm 19: CreateFixedPointPrevious($CPN, SubCPN, f, v_c, cl$)

Input: Contracted Petri net $CPN = (P, T, F, p, \{L_1, ..., L_h\})$
Input: Sub contracted Petri net $SubCPN = (P^s, T^s, D^s, Z^s, F^s, \{L^s_1, ..., L^s_h\})$, in which the fixed point will be added
Input: The original arc $f$ for which a fixed point is made
Input: Contraction node $v_c$ that is being expanded
Input: The layer number $cl$ on which the original arc $f$ leaves the contraction node $v_c$

Result: The sub contracted Petri net $SubCPN$ contains a fixed point for arc $f$
Output: The created fixed point $z$

1. $x_{vc} \leftarrow (cl - 1) \cdot \text{SeriesOffset}$
2. $y_{vc} \leftarrow v_c.y + (v_c.height/2)$
3. $D_f \leftarrow \{(f, L_i) \in D : p((f, L_i)) = \emptyset\}$
4. if $D_f \neq \emptyset$ then
   5. $d \leftarrow (f, L_i) \in D_f : i = (\uparrow j : (f, L_j) \in D)$
   6. $x_n \leftarrow d.x$
   7. $y_n \leftarrow d.y$
5. if $\exists (d, L_{cl-1}) \in Z$ then
   8. $z \leftarrow (d, L_{cl-1}) \in Z$
   9. else
      10. $z \leftarrow (d, L_{cl-1})$
      11. $Z \leftarrow Z \cup \{z\}$
      12. $L_{cl-1} \leftarrow L_{cl-1} \cup \{z\}$
   13. else
      14. $v_n \leftarrow CPN.getTopLevelNode(CPN.getLeftNode(f))$
      15. $el \leftarrow CPN.getNodeExitNumberLeft(f, v_n)$
      16. $x_n \leftarrow (el - 1) \cdot \text{SeriesOffset}$
      17. $y_n \leftarrow v_n.y + (v_n.height/2)$
5. if $\exists (v_n, L_{cl-1}) \in Z$ then
   8. $z \leftarrow (v_n, L_{cl-1}) \in Z$
   9. else
      10. $z \leftarrow (v_n, L_{cl-1})$
      11. $Z \leftarrow Z \cup \{z\}$
      12. $L_{cl-1} \leftarrow L_{cl-1} \cup \{z\}$
   13. else
      14. $z.x \leftarrow (cl - 2) \cdot \text{SeriesOffset}$
5. $z.y \leftarrow \frac{(y_{vc} - y_n)(z.x - x_n)}{x_{vc} - x_n} + y_n$
Chapter 4

Contraction Rules

The core of our algorithm is the contraction of the nodes, in such a way that the contracted nodes can trivially be laid out. So we defined rules on which the nodes may be contracted, the so-called contraction rules. Murata [8] defined six rules to reduce Petri nets, while some properties still remain. These rules were the inspiration of our contraction rules.

In this chapter our contraction rules are described. There is not a unique way to apply these rules: different order of rule application or different places in the network to apply these rules gives different results.

4.1 Rules

There are two main groups of the contraction rules, one for the series contractions and another one for the parallel contraction. The series contraction contracts nodes which have a sequential order. They contract three nodes, two of one kind and one of the other. The parallel contraction contract nodes of the same kind, which have some connections in common. Beside the rules in those two groups there are two other rules.

An overview of the rules is given in Appendix A. Here we discuss only the rules for the transitions, all the transitions rules have a similar place rule, in those rules the places are transitions and vice versa.

4.1.1 Contraction of series nodes

These rules contract a series of a transition, a place, and a transition to a single transition, see Figure 4.1. There are two preconditions, which must hold for all the four
different variants of these rules. The first one is that the place should have only one input and one output transition (precondition p1). The second one is that the transitions should not overlap each other (precondition p2). Overlapping is defined in definition 4.1.

\[ p_1 \cdot p = \{ t_1 \} \land p \cdot = \{ t_1 \} \]

\[ p_2. \quad \neg \text{overlap}(t_1, t_2). \]

**Definition 4.1** (overlap). Let \( CPN = (P, T, D, F, p, \{ L_1, ..., L_h \}) \) and \( v_1, v_2 \in P \cup T \) be nodes. Then \( CPN.\text{overlap}(v_1, v_2) = \exists L_i : \{ v_1, v_2 \} \in L_i \)

The strong variant of the rules (CONTRACTIONS OF SERIES TRANSITION STRONG) has two more preconditions. The input transition should have only the place as output place (precondition p3) and the output transition should have only the place as input place (precondition p4).

\[ p_3. \quad t_1 \bullet = \{ p \} \]

\[ p_4. \quad \bullet t_2 = \{ p \} \]

Then there are two variants for which only p1, p2 and p3 holds (CONTRACTION OF SERIES TRANSITIONS IN WEAK) or only p1, p2 and p4 (CONTRACTION SERIES OF TRANSITIONS OUT WEAK). The weakest, and fourth, variant of this rule (CONTRACTION OF SERIES TRANSITIONS WEAK) has only the preconditions p1 and p2.

\[ \forall t \in t^* : (\forall p \in \bullet t_1 \cup t_1 \bullet : \neg \text{overlap}(t, p)), \text{ where } t^* \text{ contains all the transition, which are contracted.} \]
Then there are some different preconditions the pre- and postsets of the transition should hold:

**p2.1.** \( \forall t \in t*: \bullet t_1 = \bullet t \)

**p2.2.** \( \forall t \in t*: t \bullet = t \bullet \)

**p2.3.** \( \forall t \in t*: \bullet t_1 \cup t \bullet = \bullet t \cup t \bullet \)

**p3.1.** \( \bullet t_2 \subseteq \bullet t \)

**p3.2.** \( t \bullet \subseteq \bullet t \)

The Murata look-a-like rule (CONTRACTION OF PARALLEL TRANSITIONS) has as extra preconditions that the transition should have the same preset (precondition p2.1) and postset (precondition p2.2). There is also a rule (CONTRACTION OF PARALLEL TRANSITIONS NO DIRECTION), which do not take the directions of the arcs into account, so the nodes of the presets and postsets together should be the same (precondition p2.3).

Then there are three rules, which only contract two transitions, but the equivalence on the pre- and post sets may be weaker. In one variant (CONTRACTION OF PARALLEL TRANSITIONS IN WEAK) the preset of one transition may be bigger, so the preset of one transition may be the subset of the other. The postset still should be the same. So this rule has preconditions p3.2 and p2.1 as extra preconditions. Of course there is a mirrored version of this rule (CONTRACTION OF PARALLEL TRANSITIONS OUT WEAK), this rule has the extra preconditions p3.1 and p2.2. And we have also the weakest variant of this rule (CONTRACTION OF PARALLEL TRANSITIONS WEEK) with the extra preconditions p3.1 and p3.2.

Finally, there are two rules, which only take the connected nodes before or after the contracted nodes into account. So this rule only takes the nodes situated on the layers before or after the node into account:
p4.1. \( \forall t \in t^* : \text{preLayerSet}(t_1) = \text{preLayerSet}(t) \)

p4.2. \( \forall t \in t^* : \text{postLayerSet}(t_1) = \text{postLayerSet}(t) \)

The definitions of \( \text{preLayerSet} \) and \( \text{preLayerSet} \) are given in 4.2. Note that we also use the precondition p1 for these rules, so we have no situation in which nodes are situated partly before of after the transitions, because overlap is not allowed.

**Definition 4.2** (\( \text{preLayerSet} \) and \( \text{postLayerSet} \)). Let \( \text{CPN} = (P, T, D, F, p, \{ L_1, ..., L_k \}) \) and \( v \in P \cup T \) be a node, with \( \text{max}_v = \text{CPN.getMaxLayer}(v), \text{min}_v = \text{CPN.getMinLayer}(v) \) then:

\[
\text{CPN.preLayerSet}(v) = \{ x : x \in v \cup v^\bullet : \text{CPN.getMaxLayer}(x) < \text{min}_v \}
\]
\[
\text{CPN.postLayerSet}(v) = \{ x : x \in v \cup v^\bullet : \text{CPN.getMinLayer}(x) > \text{max}_v \}
\]

The first variant of this rule (Contraction of Parallel Transitions Previous Layer) checks the nodes on the layers before the transitions and has the preconditions p1 and p4.1. The second variant (Contraction of Parallel Transitions Next Layer) checks the nodes on the layers after the transitions and has the preconditions p1 and p4.2.

### 4.1.3 Other contraction rules

There are two other contraction rules. We have one rule that contracts some arcs together and two rules that contract self-loops together.

**Contraction Self Loop**

This rule contracts a transition and a place to a single transition, see Figure 4.3. There is only one precondition to this rule, that is that the preset and postset of \( p \) is the single transition \( t \).

\[
p1. \; \bullet p = \{ t \} \land p\bullet = \{ t \}
\]

\[\text{Figure 4.3: A graphical representation of a contraction of a self loop place.}\]
Contraction Arcs
This rule contracts two arcs connected to the same nodes into one arc, see Figure 4.4.
This can only occur when the two arcs are connected in the other direction, otherwise the arcs are already contracted to one arc.

\[ \exists (p, t) \in F \land \exists (t, p) \]

For the contraction arc, we should define a direction, this direction will be from the node with the minimal layer number on which an original arc is connected, to the other node.

First we get the original arcs \( F_o \leftarrow \text{CNP.getOriginalArc}((p, t)) \cup \text{CNP.getOriginalArc}((t, p)) \).
Then we get the minimal layer number of those arcs \( \text{min}_o \leftarrow (\downarrow f \in F_o : \text{CPN.getMinLayer}(f) \downarrow) \).
Then we get the left node of the arc with the minimal layer number:
\[ v_{ln} \leftarrow (\text{CNP.getTopLevelNode}((\text{CPN.GetLeftNode}(f)) : f \in F_o : \text{CPN.getMinLayer}(f) = \text{min}_o) \]
Then we place the other arc underneath the arc with the determined direction. When we expand nodes, we always take the bottom level arcs, so there is no need for an expansion rule for this contraction rule.

4.2 The execution of the rules
In this section we discuss the execution of the rules. First of all we can use different orders of the application of the rules. Different orders gives different results. Second we can change the order of the nodes, which are investigated first. As future work these different orders can be explored more, for example in a user case study. We now give initial orders.

4.2.1 The order on the rules
As said different orders on the rules give different results. We have defined a basic order, which is created by our intuition and the strongness and weaknesses on the rules.

First we perform the Contraction of Self Loops and the Contraction of dual Arcs. The Contraction of Self Loops rule doesn’t interfere with other rules, so
can be applied at all time. The **Contraction of dual Arcs** rule can enable other rules, therefore it should be performed first, but it disables the **Contraction of Self Loops** rule, so this rule is performed before the **Contraction of Self Loops** rule.

The next rule is the **Contraction of Series Transition Strong**. This rule does not disable other rules, but it makes the net smaller, so easier to investigate.

We apply the parallel rules before the other contraction of series transition, because this brings great contribution to the symmetric requirements. Take for example Figure 4.5a. When a parallel contraction is performed, the transitions will be aligned at the middle of the places. When first a series contraction is performed, the two transitions will be aligned beside one place and the other place will be placed under (or above) the three other nodes (see Figure 4.5b).

![Figure 4.5: Two different contraction rules applied of the same nodes.](image)

We start from the strongest to the weaker ones (except for the weakest one). The **Contraction of Parallel Transitions** rule comes before the **Contraction of Parallel Transitions No Direction** rule. In this case the transitions of the same flow will be grouped, which intuitively gives better results. See for example Figure 4.6a. In a we perform first the **Contraction of Parallel Transitions** two times before the **Contraction of Parallel Transitions No Direction** rule. The transitions $t1$ and $t3$ which have the same flow (p1 as input place and p2 as output place) are grouped, as are the transitions $t2$ and $t4$, which have also the same flow. If we perform the **Contraction of Parallel Transitions No Direction** rule before the **Contraction of Parallel Transitions** rule, the latter rule will be redundant, because the first one also covers the cases of the second one. See for example Figure 4.6b.

The next two rules are the **Contraction of Parallel Transitions In Weak**, **Contraction of Parallel Transitions Out Weak**. Because these nodes are weaker than the previous parallel rules. We chose to take the **in weak** before the **out weak**, because it’s more likely that incoming arcs are connected to a previous layer, due to the Flow aesthetic, and placed closer to the transition, due to our layering method.

Then we apply the series rules **Contraction of Series Transitions In Weak** and **Contraction Series of Transitions Out Weak**. For the same reasons as for the
parallel contraction, we chose here also the IN WEAK variant before the OUT WEAK variant.

Next two rules are the Contraction of Parallel Transitions Previous Layer and Contraction of Parallel Transitions Next Layer rules. These rules are also weaker as the previous previous parallel rules. And these rules can block the previous series contraction rules.

Finally we investigate the weakest rules, namely the Contraction of Parallel Transitions Weak rule and the Contraction of Series Transitions Weak. These rules cover also the stronger ones, so when we perform these rules more in the beginning the stronger ones are redundant.

4.2.2 The place on the Petri net

Also the investigation order on the nodes gives different results. As future work an investigation on this order can be done. We use an order on the nodes which is determined by the layer numbers. Our intuition says that this is a good way since the nodes are placed as close as possible to layer 1.

4.3 Create dummy-nodes

When we contract nodes, we should investigate the arcs, whether or not they should be routed around nodes. Each contraction group has its own method to create those dummy nodes. For the parallel contraction it’s important that the arcs can be connected to the boundaries of the contraction node. For the series contraction, the placement of the nodes is similar to the placement of the nodes in the creation of the lay-out of the contracted Petri net. For Contraction Self Loop rule, we use the same method as for the series contraction rule.
Algorithm 20 controls the creation on the dummy nodes. First it checks whether or not
the contraction is a parallel contraction or a series/self loop contraction.

Algorithm 20: CREATE DummyNodes\((CPN, V_R, v_c)\)

Input: Contracted Petri Net \(CPN = (P, T, D, F, p, \{L_1, ..., L_h\})\)
Input: Set of nodes \(V_R \subseteq P \cup T\) that will be contracted
Input: Contraction node \(v_c\) that will replace the nodes of \(V_R\)
Result: The contraction node \(v_c\) has dummy-nodes

1. if \(V_R \subseteq P \lor V_R \subseteq T\) then
2. \(\text{CREATE DummyNodesParallel}(CPN, V_R, v_c)\)
3. else
4. \(\text{CREATE DummyNodesSeries}(PN, V_R, v_c)\)

If the set of contracted nodes \(V_R\) is a subset of the places \(P\) or the transitions \(T\), i.e. it
contains only nodes of the same kind, the current contraction is a parallel contraction
otherwise it’s a series contraction. After this check the right algorithm will be called.

4.3.1 Create dummy-nodes for parallel contraction

If a parallel contraction is performed we want all the arcs to be connected to the left
or right bound of the contraction node, otherwise the arcs can cross other nodes of the
contraction. See for example Figure 4.7a with three transitions that will be contracted
parallel. Without dummy nodes the arc \((t_2, p_5)\), which is connected to a node situated
on the left of the transition \(t_2\) and the arc \((t_4, p_5)\) both crosses the node \(t_3\). With the
dummy nodes we can draw the arcs such that they won’t cross any node, Figure 4.7b.

![Figure 4.7: An example a situation where arcs cross nodes a) and where they don’t,
due to dummy nodes b).](image)

Algorithm 21 creates the dummy nodes for the parallel contraction. It investigates for
each node \(v_r\) the original arc \(f\) connected to it, and adds where necessary dummy nodes
on it.

It creates a set \(Im\) with the important layer. First of all the boundary of the contraction
node, and the contracted nodes are important (line 11). Second its takes the important
layers of all the arcs connected to the node (line 12). By taking the intersection of the following sets and lists, we get the layers where the dummy nodes should be placed 16:

- The interval of the layer numbers of the contraction node \( v_c \). Because the dummy nodes should be placed in the range of this contraction node.
- The interval of the layer numbers of the arc \( f \), excluding the minimal and maximal layer number. Because the dummy nodes should be placed on the arc, between the minimal and maximal number.
- The set \( Im \) of important layers. Because we want to place as few nodes as possible, but on these layers it’s necessary to route around objects.

Take for example the original arc \((t_{15}, p_{18})\) of Figure 4.8 here the dummy nodes will be placed on the layers: \( [\min_{vc}, \max_{vc}] \cap (\min_f, \max_f) \cap Im \equiv [4,8] \cap (6,9) \cap \{4,8,7\} = \{7,8\} \)

![Figure 4.8: A parallel contraction with two nodes, left without dummy nodes, right with dummy nodes.](image)

The dummy nodes are created by the earlier described \texttt{CREATEDUMMYNODE} algorithm 9.

We want to have these dummy nodes for each contracted node separated, therefore we create for each contracted node a contraction node where we put in the contracted node, with the dummy nodes. This contraction node is made the same way as other contraction nodes. First we create and add such node (lines 4 and 5, then we add the contracted node (line 6) and the created dummy nodes 18 to the parent relationship. At last we put the node into the layers, calculate the size, and contract the contracted node with the dummy nodes (lines 21...23).

4.3.2 Create dummy-nodes for series contraction

If we contract series nodes, we are only interested in placing dummy nodes on the arcs which are contracted, i.e. those arcs which are connected between the contracted nodes.
Algorithm 21: CREATEDUMMYNODESPARALLEL($CPN, V_R, v_c$)

\textbf{Input:} Contracted Petri Net $CPN = (P, T, D, F, p, \{L_1, ..., L_h\})$
\textbf{Input:} Set of nodes $V_R \subseteq P \cup T$ that will be contracted
\textbf{Input:} Contraction node $v_c$ that will replace the nodes of $V_R$
\textbf{Result:} The contracted nodes of $V_R$ has dummy-nodes

\begin{algorithmic}[1]
\State $\min_{vc} \leftarrow CPN.getMinLayer(v_c)$
\State $\max_{vc} \leftarrow CPN.getMaxLayer(v_c)$
\For {$v_r \in V_R$}
  \State Create a new contraction node $v_{rc}$
  \State $p(v_r) \leftarrow v_{rc}$
  \State $\min_{vr} \leftarrow CPN.getMinLayer(v_r)$
  \State $\max_{vr} \leftarrow CPN.getMaxLayer(v_r)$
  \State $F_{vr} \leftarrow \{(u, v) \in F : u = v_r \lor v = v_r\}$
  \State $F_o \leftarrow CPN.getOriginalArcs(F_{vr})$
  \State $Im \leftarrow \{\min_{vc}, \max_{vc}, \min_{vr}, \max_{vr}\}$
  \State $Im \leftarrow Im \cup \{f \in F_o : CPN.getImportantArcLayers(f)\}$
  \For {$f \in F_o$}
    \State $\min_f \leftarrow CPN.getMinLayer(f)$
    \State $\max_f \leftarrow CPN.getMaxLayer(f)$
    \For {$i \in [\min_{vc}, \max_{vc}] \cap (\min_f, \max_f) \cap Im$}
      \State $d \leftarrow CREATEDUMMYNODECPN, f, i)$
      \State $p(d) \leftarrow v_{rc}$
    \End
  \End
  \State PutNODEINTOLAYERS($CPN, p^1(v_{rc}), v_{rc}$)
  \State CALCULATESIZE($CPN, p^1(v_{rc}), v_{rc}$)
  \State CONTRACT($CPN, p^1(v_{rc}), v_{rc}$)
\End
\end{algorithmic}

The dummy nodes for arcs connected to the contraction node will be investigated later on. Algorithm 22 creates the dummy nodes for the internal arcs.

First it creates a set $Im$ with the important layers. These are the start and end layers of the contracted nodes (line 3), and the important layers of contracted arcs. But also the important layers of the arcs connected between the contracted node and other nodes of the net are important, because we need these layers for the crossing reduction on the contraction node (line 4).

For every contracted arc $f$, the layers on which the dummy nodes should be added is taken, by the intersection of the interval of the layer numbers of arc $f$, with the important layers set $8$. The interval of the arc is excluding the minimal and maximal layer number, because the dummy nodes should be placed on the arc, between the minimal and maximal number. Unlike the parallel contraction we aren’t interested in
the interval of layers of the contraction node, because we’re only interested in contracted arcs, which are automatically between the intervals of the contraction node.

Take for example the original arc \((p_6, t_{11})\) of Figure 4.9 here the dummy nodes will be placed on the layers: \((\text{min}_f, \text{max}_f) \cap \text{Im} \equiv (4, 9) \cap \{3, 4, 6, 7, 8, 9\} = \{7, 8\}\)

![Figure 4.9: A series contraction, left without dummy nodes, right with dummy nodes.](image)

The dummy nodes are created by the earlier described CREATEDummyNode algorithm 9. The parent of these dummy nodes is the contraction node.

Algorithm 22: CREATEDummyNodeSeries(PN, VR, vc)

Input: Contracted Petri Net \(CPN = (P, T, D, F, p, \{L_1, ..., L_h\})\)
Input: Set of nodes \(VR \subseteq P \cup T\) that will be contracted
Input: Contraction node \(vc\) that will replace the nodes of \(VR\)
Result: The contraction node \(vc\) has dummy-nodes

1. \(F_{org} \leftarrow \bigcup \{(f \in (VR \times VR) \cap F : CPN.getOriginalArcs(f))\}\)
2. \(F_{vr} \leftarrow \{(u, v) \in F : (u \in VR \land v \notin VR) \lor (u \notin VR \land v \in VR) : CPN.getOriginalArcs((u, v))\}\)
3. \(Im \leftarrow \bigcup \{v \in VR : \{CPN.getMinLayer(v)\} \cup \{CPN.getMaxLayer(v)\}\}\)
4. \(Im \leftarrow Im \bigcup \{f \in F_{org} \cup F_{vr} : CPN.getImportantArcLayers(f)\}\)
5. foreach \(f \in F_{org}\) do
6.      \(\text{min}_f \leftarrow CPN.getMinLayer(f)\)
7.      \(\text{max}_f \leftarrow CPN.getMaxLayer(f)\)
8. foreach \(i \in (\text{min}_f, \text{max}_f) \cap Im\) do
9.      \(d \leftarrow CREATEDummyNode(CPN, f, i)\)
10.    \(p(d) \leftarrow vc\)
11. end
12. end

4.4 Expansion

As described in the expansion section 3.4 of chapter 13 are there some specific procedures with the expansion of the nodes from the different contractions. In this section we describe those specific procedures.
4.4.1 Expansion for parallel contraction

The expansion of a parallel contraction is performed in two steps. First we place the contracted nodes by performing the crossing reduction and the node placement. Second we place the dummy nodes contracted into the contraction node, again by performing crossing reduction and node placement. This is easily done due to the fact that we have to create an extra contraction node, containing only the dummy nodes.

4.4.2 Expansion for series contraction

There are two specific adaptations an expansion of the series node should have. One adaptation for the crossing reduction, and one for the node placements.

Crossing reduction

With the contraction of series, we contract three nodes. There are seven different ways these nodes can have overlapping with each other. By mirroring and node replacement we can bring these cases back to 3, see Figure 4.10. The first one is when no node overlaps other nodes (case 0). The second one is when two nodes overlap each other, but not the third (cases 1, 2 and 4). And the third one is when one node overlaps the other two (cases 3, 5 and 6). Due to the precondition p1 of the series contraction, we cannot obtain a situation where all the nodes overlap each other.

![Figure 4.10: The 6 different cases of overlapping, brought back to three representations.](image)

When we expand a node, with an overlapping situation of case 3, we should add an extra rule to the crossing reduction. This rule is that nodes \( v_1 \) and \( v_2 \) must lay on the same level. So when the leveling step of the crossing reduction is performed, these two nodes should be put on the same level. Otherwise we cannot guarantee that there is enough drawing area. For example when node \( v_1 \) is paced above \( v_3 \) and \( v_2 \) underneath \( v_3 \), we should have a height of at least the three nodes.

Node placement

For the node placements we should also take the overlapping cases into account. Because
we take for the level height, the height of the highest node on that level. If the biggest nodes lay on different levels, we also get short of drawing space. Take for example the left part of the Petri net on Figure 4.11. Here we have an overlapping case 1. The two highest nodes $t_1$ and $p_2$, both 75 high, are placed on level 1 and 2. Therefore levels 1 and 2 are both 75 high. When we add level 2 and two parallel offsets we get a height of 225. But the contraction node has the height of one big node with two dummy nodes, which is $75 + 25 + 25 + (2 \cdot 25) = 175$. Therefore we should calculate the level heights for each interval of nodes separately, with overlapping nodes taken together.

Algorithm 23 is the algorithm for this method. First the a set of each separated intervals of nodes is retrieved (line 1). This is done by algorithm 24, which is a long and straightforward algorithm. It retrieves the nodes and checks which nodes are overlapping and returns the set of intervals.

Then for each interval it performs about the same actions as the algorithm that places the nodes of the top level net. One difference is that it checks the height of the contents of the nodes in the interval (lines 8...11), and aligns this contents to the middle of the nodes (line 13). Therefore nodes on layers with fewer other nodes, will be placed in the middle, like node $t_3$ in the right example of Figure 4.11.

For the parallel contraction we do not have this problem, because the nodes are all overlapping, so the node height is calculated such that all nodes can be placed below each other. And for the expansion of a contracted node of a parallel contraction, we have only one node, so for that this problem also won’t appear.
Algorithm 23: PositioningNodesSeries(CPN, \( v_c \))

**Input:** Contracted Petri Net \( CPN = (P, T, D, F, p, \{L_1, ..., L_h\}) \)

**Input:** Contraction node \( v_c \) that contains a series contraction

**Result:** The child nodes of \( v_c \) has its position

1. \( I \leftarrow \text{CREATESETOfINTERVALS}(v_c) \)
2. \( \text{foreach } [\text{m, n}] \in I \text{ do} \)
   3. \( \text{maxLevelNumber} \leftarrow (\uparrow: L \in \{L_m, ..., L_n\}: |L \cap p^1(v)|) \)
   4. \( \text{LevelHeights}[] \leftarrow \text{create array}(\text{maxLevelNumber}) \)
3. \( \text{foreach } L \in \{L_m, ..., L_n\} \text{ do} \)
   4. \( \text{foreach } v \in L \cap p^1(v) \text{ do} \)
   5. \( \text{LevelHeights}[v.\text{level}] \leftarrow \text{LevelHeights}[v.\text{level}] \uparrow v.\text{height} \)
6. \( \text{ContentsHeight} \leftarrow 0 \)
7. \( \text{foreach } i \in (1, \text{maxLevelNumber}] \text{ do} \)
   8. \( \text{ContentsHeight} \leftarrow \text{ContentsHeight} + \text{LevelHeights}[i] \)
   9. \( \text{LevelYPositions}[1] \leftarrow v_c.y + (v_c.\text{height} - \text{ContentsHeight})/2 \)
   10. \( \text{foreach } v \in L \cap p^1(v) \text{ do} \)
    11. \( \text{LevelYPositions}[i] \leftarrow \text{LevelYPositions}[i - 1] + \text{LevelHeights}[i - 1] + ((i - 1) \cdot \text{ParallelOffset}) \)
   12. \( \text{foreach } v \in p^1(v) \cap \bigcup \{L_m, ..., L_n\} \text{ do} \)
   13. \( \text{min_v} \leftarrow \text{CPN.getMinLayer}(v) \)
   14. \( v.x \leftarrow ((\text{min_v} - 1) \cdot \text{SeriesOffset}) - (\text{NodeWidth}/2) \)
   15. \( v.y \leftarrow \text{LevelYPositions}[v.\text{level}] + ((\text{LevelHeights}[v.\text{level}] - v.\text{height})/2) \)
Algorithm 24: CreateSetOfIntervals($CPN$, $v_c$)

Input: Contracted Petri Net $CPN = (P, T, D, F, p, \{L_1, ..., L_h\})$

Input: Contraction node $v_c$ that contains a series contraction

Result: The child nodes of $v_c$ has its position

$V_{pt} \leftarrow p^1(v_c) \cap (P \cup T)$

$I \leftarrow \emptyset$

{It’s a series contraction}

if $|V_{pt}| = 3$ then

{ $v_1, v_2, v_3$ } $\leftarrow V_{pt}$

overlapOptions $\leftarrow 0$

if $CPN$.overlap($v_1, v_2$) then

overlapOptions $\leftarrow$ overlapOptions + 1

if $CPN$.overlap($v_1, v_3$) then

overlapOptions $\leftarrow$ overlapOptions + 2

if $CPN$.overlap($v_2, v_3$) then

overlapOptions $\leftarrow$ overlapOptions + 4

in $v_1 \leftarrow [CPN$.getMinLayer($v_1$), $CPN$.getMaxLayer($v_1$)]

in $v_2 \leftarrow [CPN$.getMinLayer($v_2$), $CPN$.getMaxLayer($v_2$)]

in $v_3 \leftarrow [CPN$.getMinLayer($v_3$), $CPN$.getMaxLayer($v_3$)]

switch overlapOptions do

 case 0

 I $\leftarrow \{in_{v_1}, in_{v_2}, in_{v_3}\}$

 case 1

 I $\leftarrow \{in_{v_1} \cup in_{v_2}, in_{v_3}\}$

 case 2

 I $\leftarrow \{in_{v_1} \cup in_{v_3}, in_{v_2}\}$

 case 3

 I $\leftarrow \{in_{v_1} \cup in_{v_3} \cup in_{v_2}\}$

 case 4

 I $\leftarrow \{in_{v_1}, in_{v_2} \cup in_{v_3}\}$

 case 5

 I $\leftarrow \{in_{v_1} \cup in_{v_3} \cup in_{v_2}\}$

 case 6

 I $\leftarrow \{in_{v_1} \cup in_{v_3} \cup in_{v_2}\}$

 else

{It’s a self loop contraction}

{ $v_1, v_2$ } $\leftarrow V_{pt}$

in $v_1 \leftarrow [CPN$.getMinLayer($v_1$), $CPN$.getMaxLayer($v_1$)]

in $v_2 \leftarrow [CPN$.getMinLayer($v_2$), $CPN$.getMaxLayer($v_2$)]

if $CPN$.overlap($v_1, v_2$) then

 I $\leftarrow \{in_{v_1} \cup in_{v_2}\}$

 else

 I $\leftarrow \{in_{v_1}, in_{v_2}\}$

return I
Chapter 5

The case study

In this chapter we perform a case study. Unfortunately the algorithm isn’t completely implemented, therefore the layouts in this chapter are partly created by hand. One disadvantage for this is that there’s no investigation performed of the different parameters of our algorithm, therefore, we show the lay-outs in the setting we discussed. More about the parameters can be read when the future work is discussed in Section 6.2.2. Another disadvantage is that we only have a small amount of statistics. We first discuss how the case study should be performed when the algorithm has been completely implemented. Second, we discuss the examples we showed in the introduction. Here we discuss the benefits and the limitation of those layouts.

In the last section we discuss the set-up of a user case study.

5.1 The case study

In this section we first discuss how the actual case study should be performed. Then we discuss the layouts we created for the Petri nets described in the introduction.

5.1.1 Setup of the case study

If the algorithm is completely implemented, we should create a batch-program that takes a number of Petri nets and returns the corresponding layouts. With the creation of the layouts, statistics can be automatically calculated. The following metrics can be automatically measured:

- The number of nodes in the top level layout.
Evaluation

- Number of crossings (CROSSING)
- Number of dummy nodes (bends) (BEND)
- The length of the edges (ARC_LENGTH)
- The area size of the Petri net (AREA)

The first won’t say much about the quality of the layout, but it’s good to know while adjusting the application of the rules. Then we can test how much contraction can be obtained. The other metrics do say something about the quality of the layout. They are also coupled to the aesthetics.

The main idea is to perform a lot of batch runs on the same set of Petri nets, with different parameter sets. This set of Petri nets is the so-called training set. We can also use the outcomes to fine tune the parameters. From these runs we can retrieve a couple of parameter sets which gives the best result. We use these parameter sets for the user case study. For the user case study we should use another set of Petri nets as input, the so called test set.

5.1.2 Results of the case study

In this section we take the three Petri nets of the introduction and construct a layout for them with our proposed algorithm.

Petrinet 1
This Petri net can be contracted to 6 nodes. On the layout of the top-level net, we add 10 dummy nodes. Figure 5.1 shows the top level layout of this Petri net.

Figure 5.2 shows the bottom layout of the Petri net. This Petri net does not contain crossings. It contains 16 bending points. To point them out we have left the dummy nodes visible. So we did get rid of the half of the bending points. Also, transition t10 is placed between the two connected places. In this case, our algorithm improves on the following aesthetics: CROSSING, ARC_LENGTH, BEND, FLOW. Unfortunately we have one drawback on the AREA aesthetic. First of all, more vertical area is used. Secondly, there are some empty spots on the net. Take for example the 6 dummy nodes in the right bottom corner. These can be placed higher. This is because we use the height of the highest node when we build the grid.

Also the four dummy nodes to the left of the six can be placed higher. These are placed in this way due to the contraction. Take the part of the contraction node on figure 5.3.
In this situation (this is before the nodes are placed) we see that by the contraction of parallel transitions with $t_{10}$ and the other nodes, the dummy nodes are placed above the contracted node. But there is enough area in the contracted node.

**Petrinet 2**

This Petri net can be contracted to a single node. After the expansion we get the layout of figure 5.4. It contains only 4 bending points, which is a lot fewer than the 14 of the initial situation. This layout is also much more symmetric, even the cardinal directions are almost correct, (east and west are turned around). One other advantage is the area space, we used now only 7 layers, which is less than the 9 layers of the initial situation. There is one drawback to this layout, which is that the number of crossings is 2. This is the same as for the initial situation, but we can have less when the places $ne$ and $sw$ are switched. This wasn’t the case because when they were placed, transitions $w$ and $c$ where still contracted. So the barycenters were the same and, by reversion, the number of crossing would be the same, because the arcs $(sw, w)$ and $(sw, c)$ were contracted to one. One solution could have been to add the cost of the number of original arcs to it. In this case, our algorithm improves on the following aesthetics: AREA, ARC_LENGTH, BEND, SYMMETRY.
Figure 5.2: The bottom level layout of Petri net 1

Figure 5.3: The contractions of a group nodes of Petri net 1

Petrinet 3
This Petri net can be contracted to a single node. After the expansion we get the layout of figure 5.5. Which is a great difference to the initial situation. First of all the many long arcs with bending points are gone. Second of all, with 13 layers, we have reduced the number with 10. Also the number of crossings is strongly reduced. In this case, our algorithm improves on the following aesthetics: CROSSING, AREA, ARC_LENGTH, BEND,
5.2 The user study

One important aspect is that the Petri nets should be understandable to users. People who must work with them should be able to understand them quickly. The aesthetic requirements are a tool to make those nets understandable, but the opinion of the user is subjective. That’s why a user study is helpful to get insight into the quality of the layout of the Petri nets.

5.2.1 Participant

The participants of the case study can all be persons who have knowledge about Petri nets. It’s unnecessary to ask other persons, as they won’t use Petri nets. From this group of persons with knowledge, we should distinguish between people who work frequently with Petri net, like the users of ProM, and a group of persons who don’t work often with Petri nets, like student who learn about Petri nets. Of course the first group is the most important, because they will be the intended end users. But if there is a significant
difference between the outcomes in both groups, this can be useful information. It can be the case that a 'perfect' layout for people who work frequently with Petri nets is different than for people who work rarely with them. For example an experienced user can have a better knowledge of certain patterns (nodes connected in some standard way to each other) and if they are placed a certain way they see quickly the patterns. So one outcome can be a set of parameters for experienced user and one set of parameters for learning users.

5.2.2 Questionnaire

The questionnaire is quite straightforward. There is a set of test Petri nets, and for each Petri net a couple of layouts are created with different parameter sets. The participants should indicate which Petri net is the best, by giving it points. For each question the same amount of points can be given. This way the participant can indicate how much one layout is better than the other. If for example one layout is perfect and the other ones are worse, the participant gives this net the full amount of points, but when two lay-outs are best, the participant can give both layouts the half the points. This way we do not only check which lay out is better, but also how much it’s better.

The number of different parameter sets should not be too big, because the participant should get a good overview of the different layouts. With a too large set, it is less easy to get that overview. Also the parameter sets should give clearly different layouts, because we don’t want the user to spend their time on finding out what’s the difference, which they likely will do when the difference is not very visible.

The test Petri net set should also not too big. There should be a great variation in Petri nets, however, such as in the size, the density (the number of edges vs. the number of nodes) and properties of the Petri net.

5.2.3 Outcome

There can be different outcomes from this user case study. For example we can get one parameter set, which is overall the best. But there’s also a possibility that there are certain parameters sets, which are good to visualize certain Petri nets. Then we can adapt the algorithm such that it takes a particular parameter set for a certain Petri net.

Of course can the outcome be a good start for further investigations, like the fine tuning of a certain parameter set.
Chapter 6

Conclusions

In this final chapter this thesis is concluded. First we give the conclusions of our algorithm. Then we provide some future work and recommendations for research.

6.1 Conclusion

We have developed an algorithm to visualize Petri nets. This algorithm is much more suited to visualize Petri nets than current graph layouts. The algorithm has a big set of parameters and is because of that very flexible. This can be an advantage but also a disadvantage. The advantage is that users can adapt the algorithm in such a way that it fits their expectations, or the algorithm can be specialized in different ways for certain types of Petri nets. The disadvantage is that there are many possibilities, and no such thing as one perfect algorithm.

The algorithm we provided is a working one on paper, but has not been implemented completely. Therefore a small case study by hand has been performed. A case study has been proposed, which is to be carried out when the implementation is ready. This case study is also a preparation for the user case study we have described.

6.2 Future work

In this section we provide some possible improvements and extensions which can be applied in the future. We first discuss the parameters of the algorithm.
6.2.1 Assumptions

Let us recall the assumptions made at the beginning and discuss whether or not we can need these assumptions.

**Workflow nets (WF-nets)**

In our algorithm we use the properties of WF-nets, so we can not just omit this assumption. If we change the methods where we use these properties into methods that do not need these properties, the algorithm is, of course, suitable for normal Petri nets. But this may not be the best solution. Due to the WF-nets, the FLOW was important, therefore we performed layering. When we have a general Petri net, this FLOW aesthetic can be less important, or may be omitted. Then the layering step can also be omitted and we can just start with the contraction of the nodes. This way the crossing reduction and placement should be adapted, because we won’t have layers anymore.

**Small**

We didn’t use this assumption, so we can also lay-out enormous WF-nets. An investigation should reveal whether the algorithm is fast enough and if the result is still readable.

**Size of the nodes**

We use through the whole algorithm the property of equal node size, therefore we cannot omit this assumption. One way to resolve this is to resize all the nodes virtually to the size of the biggest node, but this can have tremendous consequences on the area aesthetic.

6.2.2 Parameters

In our algorithm there are many parameters, which can be explored more. In this subsection we give an overview of those parameters.

**The order on the rules**

The order on the execution of the rules can be can be investigated further. Maybe Petri nets with certain properties require a different order than other Petri nets.

**The place on the Petri net**

The order on the nodes, which should be investigated first in the contraction can be looked at further. We now look at the nodes in the order of the layer numbers. But one
can also think of investigating the order on the nodes by starting from the middle and going to the end.

We can also combine the order of the rules and the place on the net. For example we can create a subset on the nodes, which are important. This can be either properties of the nodes/Petri net, or we let the user define those nodes. Then we can perform the contraction rules only on that subset, and when all the contraction rules rules are performed, start again with the contraction, but now including the other nodes, so that the important nodes are contracted first and will be placed near each other.

**Cost function for the crossing reduction**

We use a higher number for a connection between two temporary nodes in the leveled barycentric crossing reduction. The value now is initially 10. An investigation can be done on the best value for this cost. One other solution can be made by examining the number of arcs between the two layers in the contraction node. Then we have the absolute cost of the crossing.

We can also take the number of contracted arcs into account as cost for contraction arcs.

**Levels for the crossing reduction**

We use the minimum level number for the crossing reduction. This can introduce extra crossings. Therefore we can use higher level numbers, which may be give us less crossings and a better layout.

6.2.3 Limitations

Finally, we give some limitations currently present in the algorithm.

**Alignment of the nodes**

There are two limitations on the alignment of the nodes. First of all we create a grid, on which we align the top level nodes. There is a possibility that certain levels become too large. We could try to find a way where the nodes will be aligned closer to each other.

**Sortation of the barycenters**

Another limitation is that we align the node to the top of the area space. In the crossing reduction we have defined a level number. When we order the nodes on the barycenters, we can keep the level number into account as the index, and place the nodes to the level closest to the index, if possible. See for example Figure 6.1a. Here the nodes, with the
corresponding level numbers are ordered from bottom to top. In Figure 6.1b, they are ordered according to the level number. Maybe this can be taken a step further like in Figure 6.1c, when two nodes with the same barycenters are sorted before and after the index.

![Figure 6.1: Different ways to sort the barycenters.](image-url)
Appendix A

Contraction rules

Recall the definitions of overlap (Definition A.1) and preLayerSet and postLayerSet (Definition A.2).

Definition A.1 (overlap). Let $CPN = (P, T, D, F, p, \{L_1, ..., L_h\})$ and $v_1, v_2 \in P \cup T$ be nodes. Then $CPN.overlap(v_1, v_2) = \exists L_i : \{v_1, v_2\} \in L_i$.

Definition A.2 (preLayerSet and postLayerSet). Let $CPN = (P, T, D, F, p, \{L_1, ..., L_h\})$ and $v \in P \cup T$ be a node, with $max_v = CPN.getMaxLayer(v)$, $min_v = CPN.getMinLayer(v)$ then:

- $CPN.preLayerSet(v) = \{x : x \in \bullet v \cup v \bullet : CPN.getMaxLayer(x) < min_v\}$
- $CPN.postLayerSet(v) = \{x : x \in \bullet v \cup v \bullet : CPN.getMinLayer(x) > max_v\}$

A.1 Series contraction rules

<table>
<thead>
<tr>
<th>Contraction Series Transitions Strong(CSTS)</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Preconditions</strong></td>
</tr>
<tr>
<td>$\bullet p = {t1}$</td>
</tr>
<tr>
<td>$p \bullet = {t2}$</td>
</tr>
<tr>
<td>$\bullet t2 = {p}$</td>
</tr>
</tbody>
</table>

![Series Contraction Diagram]
Appendix A. Contraction Rules

Contraction Series Places Strong (CSPS)

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bullet t = { p_1 } )</td>
<td>![Diagram CSPS - Places Strong]</td>
</tr>
<tr>
<td>( t \bullet = { p_2 } )</td>
<td></td>
</tr>
<tr>
<td>( p_1 \bullet = { t } )</td>
<td></td>
</tr>
<tr>
<td>( \bullet p_2 = { t } )</td>
<td></td>
</tr>
<tr>
<td>( \neg \text{overlap}(p_1, p_2) )</td>
<td></td>
</tr>
</tbody>
</table>

Contraction Series Transitions In Weak (CSTIW)

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bullet p = { t_1 } )</td>
<td>![Diagram CSTIW - Transitions In Weak]</td>
</tr>
<tr>
<td>( p \bullet = { t_2 } )</td>
<td></td>
</tr>
<tr>
<td>( t_1 \bullet = { p } )</td>
<td></td>
</tr>
<tr>
<td>( \neg \text{overlap}(t_1, t_2) )</td>
<td></td>
</tr>
</tbody>
</table>

Contraction Series Places In Weak (CSPIW)

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \bullet t = { p_1 } )</td>
<td>![Diagram CSPIW - Places In Weak]</td>
</tr>
<tr>
<td>( t \bullet = { p_2 } )</td>
<td></td>
</tr>
<tr>
<td>( p_1 \bullet = { t } )</td>
<td></td>
</tr>
<tr>
<td>( \neg \text{overlap}(p_1, p_2) )</td>
<td></td>
</tr>
</tbody>
</table>
### Contraction Series Transitions Out Weak (CSTOW)

**Preconditions**

- $p = \{t_1\}$
- $p_\bullet = \{t_2\}$
- $t_2 = \{p\}$
- $\neg \text{overlap}(t_1, t_2)$

**Representation**

![Diagram](image1)

### Contraction Series Places Out Weak (CSPOW)

**Preconditions**

- $t = \{p_1\}$
- $t_\bullet = \{p_2\}$
- $p_2 = \{t\}$
- $\neg \text{overlap}(p_1, p_2)$

**Representation**

![Diagram](image2)

### Contraction Series Transitions Weak (CSTW)

**Preconditions**

- $p = \{t_1\}$
- $p_\bullet = \{t_2\}$
- $\neg \text{overlap}(t_1, t_2)$

**Representation**

![Diagram](image3)
### Contraction Series Places Weak (CSPW)

**Preconditions**

- $\bullet t = \{p1\}$
- $t\bullet = \{p2\}$
- $\neg \text{overlap}(p1, p2)$

**Representation**

![Diagram of CSPW]

### Parallel contraction rules

#### Contract Parallel Transitions (CPT)

**Preconditions**

1. $\forall t \in t\ast: \bullet t1 = \bullet t$
2. $\forall t \in t\ast: t1\bullet = t\bullet$
3. $\forall t \in \{t1\} \cup t\ast$
   - $\forall p \in \bullet t1 \cup t1\bullet$
   - $\neg \text{overlap}(t, p)$

**Representation**

![Diagram of CPT]
### Appendix A. Contraction Rules

#### Contract Parallel Places (CPP)

**Preconditions**

<table>
<thead>
<tr>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\forall p \in p^* : p_1 = p)$</td>
</tr>
<tr>
<td>$(\forall p \in p^* : p_1 \cdot = p \cdot)$</td>
</tr>
<tr>
<td>$(\forall p \in {p_1} \cup p^* : \forall t \in \bullet p_1 \cup p_1 \bullet : \neg overlap(p,t))$</td>
</tr>
</tbody>
</table>

#### Contract Parallel Transitions No Direction (CPTND)

**Preconditions**

<table>
<thead>
<tr>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\forall t \in t^* : \bullet t_1 \cup t_1 \bullet = \bullet t \cup t \bullet)$</td>
</tr>
<tr>
<td>$(\forall t \in {t_1} \cup t^* : \forall p \in \bullet t_1 \cup t_1 \bullet : \neg overlap(t,p))$</td>
</tr>
</tbody>
</table>
### Contract Parallel Places No Direction (CPPND)

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\forall p \in p^* : p_1 \cup p_1^{\bullet} = p \cup p^{\bullet})$</td>
<td>![CPPND Diagram]</td>
</tr>
<tr>
<td>$(\forall p \in {p_1} \cup p^* : \forall t \in p_1 \cup p_1^{\bullet} : \neg \text{overlap}(p, t))$</td>
<td>![CPPND Diagram]</td>
</tr>
</tbody>
</table>

### Contract Parallel Transitions In Weak (CPTIW)

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$t_1 \supseteq t_2$</td>
<td>![CPTIW Diagram]</td>
</tr>
<tr>
<td>$t_1^{\bullet} = t_2^{\bullet}$</td>
<td>![CPTIW Diagram]</td>
</tr>
<tr>
<td>$(\forall p \in t_1 \cup t_1^{\bullet} : \neg \text{overlap}(t_1, p) \land \neg \text{overlap}(t_2, p))$</td>
<td>![CPTIW Diagram]</td>
</tr>
</tbody>
</table>
### Contract Parallel Places In Weak (CPPIW)

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Representation</th>
</tr>
</thead>
</table>
| •\(p1 \supseteq •p2\)  
  \(p1\bullet = p2\bullet\)  
  \((\forall t \in •p1 \cup p1\bullet : \neg overlap(p1, t) \land \neg overlap(p2, t))\) | ![Diagram of CPPIW condition](image) |

### Contract Parallel Transitions Out Weak (CPTOW)

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Representation</th>
</tr>
</thead>
</table>
| •\(t1 = •t2\)  
  \(t1\bullet \supseteq t2\bullet\)  
  \((\forall p \in •t1 \cup t1\bullet : \neg overlap(t1, p) \land \neg overlap(t2, p))\) | ![Diagram of CPTOW condition](image) |
### Contract Parallel Places Out Weak (CPPO\textsuperscript{W})

** Preconditions **

\[
\begin{align*}
\bullet p_1 &= \bullet p_2 \\
\bullet p_1 &\supseteq \bullet p_2 \\
(\forall t \in \bullet p_1 \cup p_1) : \neg overlap(p_1, t) \land \neg overlap(p_2, t)
\end{align*}
\]

** Representation **

![Diagram of Contract Parallel Places Out Weak (CPPO\textsuperscript{W})]

### Contract Parallel Transitions Weak (CPT\textsuperscript{W})

** Preconditions **

\[
\begin{align*}
\bullet t_1 &\supseteq \bullet t_2 \\
\bullet t_1 &\supseteq \bullet t_2 \\
(\forall p \in \bullet t_1 \cup t_1) : \neg overlap(t_1, p) \land \neg overlap(t_2, p)
\end{align*}
\]

** Representation **

![Diagram of Contract Parallel Transitions Weak (CPT\textsuperscript{W})]
### Appendix A. Contraction Rules

#### Contract Parallel Places Weak (CPPW)

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$p_1 \supseteq p_2$</td>
<td><img src="cppw.png" alt="CPPW" /></td>
</tr>
<tr>
<td>$p_1 \bullet \supseteq p_2 \bullet$</td>
<td></td>
</tr>
<tr>
<td>$(\forall t \in p_1 \cup p_1 \bullet : \neg \text{overlap}(p_1, t) \land \neg \text{overlap}(p_2, t))$</td>
<td></td>
</tr>
</tbody>
</table>

#### Contract Parallel Transitions Previous Layer (CPTPL)

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$(\forall t \in t^* : \text{preLayerSet}(t_1) = \text{preLayerSet}(t))$</td>
<td><img src="cptpl.png" alt="CPTPL" /></td>
</tr>
<tr>
<td>$(\forall t \in {t_1} \cup t^* : \forall p \in \text{preLayerSet}(t_1) : \neg \text{overlap}(t, p))$</td>
<td></td>
</tr>
</tbody>
</table>
### Appendix A. Contraction Rules

#### Contract Parallel Places Previous Layer (CPPPL)

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Representation</th>
</tr>
</thead>
</table>
| \((\forall p \in p^* : \) \)  
  \(preLayerSet(p1) = preLayerSet(p)\)  
  \((\forall p \in \{p1\} \cup p^* : \) \(\forall t \in preLayerSet(p1) : \) \(\neg overlap(p, t)\) | ![CPPPL Diagram] |

#### Contract Parallel Transitions Next Layer (CPTNL)

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Representation</th>
</tr>
</thead>
</table>
| \((\forall t \in t^* : \) \)  
  \(postLayerSet(t1) = postLayerSet(t)\)  
  \((\forall t \in \{t1\} \cup t^* : \forall p \in postLayerSet(t1) : \) \(\neg overlap(t, p)\) | ![CPTNL Diagram] |
### A.3 Other contraction rules

**Contraction of Self Loops (CSL)**

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>$\bullet v_1 = v_1 \bullet$</td>
<td><img src="image" alt="Diagram of Contraction of Self Loops" /></td>
</tr>
<tr>
<td>$</td>
<td>\bullet v_1</td>
</tr>
</tbody>
</table>
### Contraction of dual Arcs (CA)

<table>
<thead>
<tr>
<th>Preconditions</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>( \exists (p, t) \in F \land \exists (t, p) )</td>
<td><img src="image" alt="Diagram" /></td>
</tr>
</tbody>
</table>
Bibliography


