Visualization of Large Process Models

by

To Tu Cuong

A thesis submitted in partial fulfillment for the degree of European Masters in Informatics

in the
Department of Information Engineering and Computer Science
Department of Computer Sience
Business Integration Technologies Group
1st Reader: Prof. Fabio Casati from University of Trento
2nd Reader: Prof. Matthias Jarke from RWTH Aachen

November 2009
Declaration of Authorship

I, To Tu Cuong, declare that this thesis titled, ‘Visualization of Large Process Models’ and the work presented in it are my own. I confirm that:

■ This work was done wholly or mainly while in candidature for a degree at this University.

■ Where any part of this thesis has previously been submitted for a degree or any other qualification at this University or any other institution, this has been clearly stated.

■ Where I have consulted the published work of others, this is always clearly attributed.

■ Where I have quoted from the work of others, the source is always given. With the exception of such quotations, this thesis is entirely my own work.

■ I have acknowledged all main sources of help.

■ Where the thesis is based on work done by myself jointly with others, I have made clear exactly what was done by others and what I have contributed myself.

Signed: 

Date: 
“A picture is worth a thousand words.”
Business process models can become large and unstructured. To get a better overview of such a process model, techniques are needed that show a large process model as a meaningful composition of its parts. In this project, techniques are developed, implemented and evaluated that provide a decomposition of a large process model and lay it out in a fully-automatic or user-driven fashion. Our hypothesis is that a hierarchical decomposition of a process model produces a better layout of the business process models. When a process model is decomposed into much smaller components, the layout algorithm can be applied locally to each component. This reduces the complexity of the layout algorithm since it does not have to deal with the entire process model at the same time. Later these layout results will be combined to produce the final result.
Many thanks and gratitude to Dr. Thomas Gschwind, Dr. Jana Koehler, Professor Fabio Casati, and Professor Matthias Jarke for dedicating their effort and time in this project and having patience and endurance to guide me throughout this project.
# Contents

Declaration of Authorship i

Abstract iii

Acknowledgements iv

1 Introduction 1

1.1 BPMN Diagram Layout Problem 2

1.2 An Automatic Layout Algorithm for BPMN Diagram 3

1.3 Overview 4

2 Motivation 5

2.1 Business Process Modeling 5

2.2 Business Process Model Layout: State of the Art 6

2.3 Layout Aesthetics and Requirements for Process Models 7

3 Graph Layout Algorithms 9

3.1 yFiles 9

3.2 Graphviz 10

3.3 Pigale 10

4 Process Model Layout Algorithm 12

4.1 Process Structure Tree 12

4.2 Process Model Fragment Classification 14

4.3 The Recursive Layout Algorithm 15

4.4 Layout of Different Fragment Types 17

4.5 The Bounding Box Computing Algorithm 18

4.6 Parallel Edges between Process Elements 19

5 Layout of Planar Fragments 20

5.1 The Left-Right Planarity Algorithm 20

5.1.1 Orientation 21

5.1.2 Testing 26

5.1.3 Merging the Constraints of Back Edges 29

5.1.4 Embedding 29
Dedicated to my parents and my girlfriend
Chapter 1

Introduction

Every organization has its purpose. It may build or sell computers, e.g., Apple. It may be in the hotel business such as Hilton. In order to achieve its business goals efficiently, its operation is broken down into smaller business units. They may be Marketing, Billing, Delivery, or Human Resources. All units work together to contribute towards the organization’s business goals. Each unit, in turn, has its own internal activities. For example, Delivery will be responsible for packaging products, tracking the package etc. In order to fulfill these responsibilities, they create a number of business process or their own way of doing things. With the advance of Information Technology, these business processes are automated using IT systems. But there is a gap between the real business processes and the ones captured by the IT systems. Often, the processes in the IT systems do not fully reflect the corresponding process in reality. We need to make sure they are consistent with each other. This is where Business Modeling comes into the picture.

Business Modeling ensures consistency and thoroughness in capturing relevant information such that both business analysts and developers can understand the business requirement and capture them correctly in the IT system. To model a business process, an analyst can use a notation language such as UML Activity Diagrams, or BPMN (Business Process Modeling Notation [1]). The result will be business process models. These models will serve as references in implementing an IT system, thus it is important for stakeholders to understand them. Unfortunately, business process models usually include lots of activities and conditions for them to execute and they get complex as more business logics are integrated into them, hence become less readable for the stakeholders. To remedy this problem, we propose a fully automatic algorithm to lay them out, in the context of BPMN used as modeling language.
1.1 BPMN Diagram Layout Problem

BPMN is a modeling notation that is disseminated by OMG (Object Management Group [2]). It consists of the following elements:

**Flow Objects** are the main graphical elements that define the behavior of a business process. There are three flow objects: events, activities, and gateways.

**Connecting Objects** are graphical elements that help connect flow objects together. There are three connecting objects: sequence flow, message flow, and association.

**Swimlanes** are ways of grouping modeling elements. There are two way to do so: pools and lanes.

**Artifacts** are used to provide additional information about process models. There are three standardized artifacts: data object, group, and annotation.

The graphical notations for the BPMN elements are shown in figure 1.1. For detailed meaning of these graphical notations, please refer to [3].

![Image of BPMN Graphical Notation](image_url)

**Figure 1.1:** BPMN Graphical Notation
An automatic layout algorithm for BPMN process models is difficult to design. This is due to the nature of BPMN process models. They include various elements of different types. They have no predefined structure. Moreover, an automatic layout algorithm has to support the order between process model elements and clusters of them since a process model represent ordered activities and related ones of an organization.

1.2 An Automatic Layout Algorithm for BPMN Diagram

We have implemented our process model layout algorithm in the context of IBM WebSphere Business Modeler (WBM), which supports BPMN. Although WBM already has its own automatic layout algorithm, there are areas that we believe can be improved. WBM uses the yFiles library to layout process models. According to its documentation, it implements a variety of layout styles such as hierarchical layout, orthogonal layout of nodes, or orthogonal edge routing. Since yFiles algorithms were not designed specifically for process models, the layout of process models is not always optimal with regards to minimizing the number of crossing edges and sometimes the position of the process model elements. yFiles is a proprietary library and we do not have any insights on how the layout algorithms are implemented, therefore we cannot adapt them to solve our problem.

Our focus is on positioning process model events, gateways, activities, and the connecting objects. We do not support pools and lanes layout.

Implementing a layout algorithm for business process models is challenging. They can be large, or cyclic. The semantics of business process model may also affect the layout. One promising solution to deal with the complexity of the models is to use graph theory to analyze and decompose them into smaller components. This reduces the complexity of the layout algorithm since it does not have to deal with the entire process model at the same time. Later these layout results will be combined to produce the final result.

To be able to apply graph theory, we need to convert a business process model into a graph. We have the following definition:

**Definition 1.1 (BPMN Graph).** A BPMN graph is a graph $G = (V, E)$ with the following additional information:

- A mapping of nodes: $V \rightarrow F$, where $F$ denotes the set of flow objects of a process model. Each $v \in V$ is mapped to exactly one $t \in T$ and vice versa.
- A mapping of edges: $E \rightarrow C$, where $C$ denotes the set of connecting objects of a process model. Each $e \in E$ is mapped to exactly one $c \in C$ and vice versa.
1.3 Overview

This thesis is organized as follows. In chapter 2, we further describe Business Process Modeling and motivate the importance of business process models. We also explain why general layout algorithms are not suitable for BPMN process models. Moreover, we define relevant layout aesthetics criteria for business process model. Then, in chapter 3, we give an overview of graph drawing libraries and softwares. We present our layout algorithm in chapter 4. Then, we describe the layout algorithms for planar and non-planar components of business process models in chapter 5 and chapter 6 respectively. Our layout algorithm evaluation is given in chapter 7. Future work is discussed in chapter 8. Finally, we give our summary and conclusion in chapter 9.
Chapter 2

Motivation

In this chapter, we will discuss the importance of business process modeling hence motivates a need for an automatic layout algorithm for business process model. To judge the quality of business model layout, we also present some aesthetics and requirements.

2.1 Business Process Modeling

Companies have many reasons to model their business processes. Companies who have merged together want to examine their processes across their lines of business to determine the best one. Other companies are looking to improve their existing business processes, or even automate them. In some countries, companies are required to document their business processes.

Business process modeling captures business processes, which have a specific ordering of activities with clearly defined input and output values that produces a specific business value. For example, an insurance claim process takes a customer’s document to analyze whether the customer is eligible for the insurance claim. Different stakeholders may build business models to meet a wide range of business objectives. For example, an analyst needs a high-level view of a process to make strategic decisions. A developer can use a process model as the input to implement a solution.

Before modeling a business process, an analyst can gather requirements from different sources such as text files, spreadsheets or PowerPoint. The analyst uses this information together with existing process models to construct the models. After modeling the business processes, the analyst can simulate them using business modeling tools such as WBM. It provides many advanced features that allow her to accurately simulate models. When the design of business models have been proven by simulation, the analyst can
add technical details to them. Then they can be exported, creating useful artifacts for implementing IT systems using standards such as XSD, WS-BPEL, or WSDL. We also can reuse business models, designed using WBM, in IBM Rational Software Architect. This enable software architects to create UML models based on the information captured by the business analyst.

2.2 Business Process Model Layout: State of the Art

With business process model being so important to many companies, a good understanding of them is desirable. A layout algorithm of process models that provides clean, represented, and easy to understand layouts contributes to our understanding. Business process models are graphs with their nodes and edges being the elements and links, respectively, in the process models. We then can apply graph layout algorithms on the process models. Unfortunately, although there are many graph layout algorithms today, most of them target a specific diagram such as organization chart, circuit design, or social network. Hence they are not suitable for BPMN. The problem with these layout algorithms is that they do not take into account specific constraints that can improve the readability of BPMN diagrams.

Force-based layouts such as in [4] generate rather organic layout diagram, which is not suitable for laying out business process models. Moreover, their result may differ in different runs, hence will provide unpredictable layout for business process models.

Orthogonal layouts - e.g., [5] - produce layout with links consisting of vertical and horizontal line segments. This is desirable for BPMN diagram but since the common goal of these layout algorithms are, e.g., minimizing the area occupied by a graph, they may ignore the requirements for a BPMN diagram. Nevertheless, modified orthogonal layout algorithms may be able to generate satisfying layouts for BPMN diagrams.

Hierarchical layouts, as in [6], are more suited for BPMN diagrams since the hierarchy given by gateways, and order of the process elements can be taken into account. Moreover layout constraints of BPMN diagram can be integrated to improve the layout quality.

In [7], the authors introduce the concept “division” to decompose BPMN diagrams into smaller pieces. Although this approach give us an overview that lacks in large diagrams, it does not improve the existing algorithms significantly. The actual layout include two steps. First, a BPMN diagram is decomposed into subgraphs. The result is an initial orthogonal layout. Second, a sketch-driven orthogonal algorithm is applied to
lay subgraphs out. Although the algorithm consists of these two complex phase, the runtime properties was not provided.

In [8], the authors propose a simple algorithm to layout a process model. It focuses mainly on local changes which does not require knowledge on the structure of a process model and gives higher priority to nodes over edges. Although the authors provide statistics on its runtime, the formal complexity was not given.

To our knowledge, all process model layout algorithms treat models as general graphs. We believe to be able to layout a process model we must have some sort of understanding about its structure.

### 2.3 Layout Aesthetics and Requirements for Process Models

Aesthetics of a layout measures the graphical property of the drawing that improves its readability. Besides general graph layout aesthetics, business process models have specific requirements about their layout. For our algorithm evaluation, we consider the following business process model layout requirements and graph aesthetics:

For model layout requirements, we have:

**Flow:** Process model elements have a processing order. This order constitutes a flow from left to right in the business process model. A business process model layout should maximize the number of edges respecting the flow. We can compute the ratio between the number of edges that respect the flow and the total edges in the diagram to measure this requirement.

**Orthogonal:** Edges in a business process model consists of horizontal and vertical line segments. We prefer to have straight edges in the diagram.

**Bimodality:** Process model elements has two sides. The incoming edges enter an element from the left of it. The outgoing edges leave an element from the right of it.

For graph aesthetics, we have:

**Crossing:** Crossings of edges is harmful for the business process model readability. A business model layout should minimize the number of crossings.
Area: A business process model layout need to be compact. In other words, the layout should minimize the drawing space.

Cluster: It is desired to cluster related process model elements.
Chapter 3

Graph Layout Algorithms

A graph’s layout is a way to draw it. There are several popular graph layouts such as orthogonal layout, tree layout, or circular layout. Moreover, we also have different algorithms to generate these layouts. For a business process model, the hierarchical layout with orthogonal edge routing is preferred. In this chapter, we will introduce several popular graph libraries and softwares. We consider both open-source and proprietary ones.

3.1 yFiles

yFiles is a Java class library that provides algorithms for graph analysis, visualization, and automatic layouts of graphs. It has graph layout algorithms such as hierarchical layout, organic layout orthogonal layout, tree layout, circular layout. It has been used to draw entity relationship diagrams, UML diagrams, etc. It also provides edge routing algorithms which are suitable for interactive or incremental graph drawing. For example, some edges should be re-drawn, after the user moves some nodes. We give an overview of graph library and software in section.

yFiles library includes three main components:

- **Basic** serves as the main part of the library. It provides classes and data types for graph analysis tasks. Moreover we also have a wide variety of graph and network algorithms that form an ideal toolkit for any network analysis tasks.

- **Viewer** is in charge of everything relating to the user interaction. It provides the graph viewer component and Swing-based GUI elements. One important aspect of viewer
is its support for different graph formats such as GML (Graph Modeling Language, YGF (Y Graph Format) and its printing capabilities.

Layout provides automatic layout algorithms mentioned above. Additionally, it provides edge routing algorithms that make it possible to re-route edges in an existing graph.

3.2 Graphviz

graphViz is an open source graph visualization software. It provides several graph layout programs:

dot implements the algorithm presented in [9]. It makes hierarchical or layered drawing of directed graphs. The layout algorithm includes 4 passes. In the first pass, it finds an optimal rank assignment. Then in the second pass, it arranges the vertex within ranks to reduce crossings. The third pass computes optimal coordinate for nodes. Finally, the forth pass calculates splines to draw edges. The algorithm is fast and generates a good layout.

neato and fdp produce “spring model” layouts. neato implements the Kamada-Kawai algorithm [10] and fdp uses the Fruchterman-Reingold heuristic [4] that handles large graphs and clustered undirected ones.

twopi makes radial layout [11]. The nodes are placed on concentric circles depending their distance from a given root node.

circo makes circular order [12, 13]. This is suitable for representing networks whose diagram of multiple cyclic structures such as telecommunications networks.

graphViz takes graph descriptions in a simple text language and produces diagrams in different formats. Since it is an open source project, we can take a look at the source code to get more insights.

3.3 Pigale

Pigale is a public implementation of a graph algorithm library and editor. It includes a graph editor and a C++ algorithm library essentially focused on planar graphs. It provides algorithms for graph analysis such as Fraysseix-Rosenstiehl left-right algorithm
[14] for planarity test and embedding computation, which is probably the fastest planarity test [15]. It also implements a wide variety of graph drawing algorithms such as Schnyder algorithm [16], Tutte bary centric representation of 3-connected graphs, or visibility representation of planar graphs [17]. Additionally, it even provides graph visualization in 3-D. Like Graphviz, Pigale is an open source project, therefore we can learn from their implementation.
Chapter 4

Process Model Layout Algorithm

In this chapter, we will present our layout algorithm. Our hypothesis is that a hierarchical decomposition of a process model produces a better layout of the business process model. When a process model is decomposed into much smaller components, the layout algorithm can be applied locally to each component. This reduces the complexity of the layout algorithm since it does not have to deal with the entire process model at the same time. Later these layout results will be combined to produce the final result. The decomposition step has been presented in [18] which bases itself on an algorithm by Hopcroft and Tarjan [19] that finds the triconnected components of a biconnected graph. The result of this phase is a hierarchical tree structure that consists of components (fragments) of the original process model. We also analyze the planarity of these components and compute their embedding, if any, since we want to lay planar components out without crossings of edges.

In section 4.1, we introduce the process structure tree of business process models. Then in section 4.2, we classify this tree’s fragments into different categories. From this classification, we illustrate our recursive layout algorithm in section 4.3. This algorithm, in turn, uses different layout strategies for each fragment type and an bounding box computation algorithm. These strategies are introduced in section 4.4. And in section 4.5, we presents the algorithm for calculating bounding boxes of process model elements. Finally, we shows how to layout parallel edges between process model elements in section 4.6.

4.1 Process Structure Tree

A process model is converted to a workflow graph, which is parsed to generate a process structured tree [18]. This tree is a hierarchical representation of a process model as
shown in Figure 4.1. It consists of single-entry-single-exit (SESE) fragments that are typically much smaller than the original process model. These fragments are either nested or disjoint. For example in Figure 4.1, we the SESE fragment Y, which in turn contains the SESE fragments X and W.

The decomposition of workflow graph into a process structure tree is structurally unique and modular, where modular means that a local change of the workflow graph only cause a local change of the decomposition. This is crucial for our layout algorithm. First, it is structurally unique therefore our layout algorithm will be consistent, generating the same layout for the same model in every run. We consider that models stored in different physical files are different. Second, it is modular hence any changes in the process model can cause only local changes in the decomposition. Therefore, only local changes can happen in our layout result. This helps to preserve the mental map of users about the process model.
4.2 Process Model Fragment Classification

The process structure tree provides us a hierarchical decomposition of fragments for a business process model. To lay the model out, we need to lay out individual fragments. Each of them may have different structure, hence we need to classify them into separate categories. This classification later will help us lay out each fragment. We classify the process structure tree’s fragments as structured and unstructured fragments (Figure 4.2). Structured fragments again are classified as sequence, parallel, alternative, and loop fragment. Unstructured fragments also are classified as planar and non-planar fragments. The formal definition of these fragment types are as follows:

**Structured fragment:** A fragment that is either a sequence, an alternative, a parallel, or a loop fragment, is structured.

**Sequence fragment:** A sequence fragment consists of a sequence of one or more process elements. An example is shown in the figure 4.3(a)

**Alternative and Parallel fragment:** An alternative (parallel) fragment consists of a decision (fork) followed by a merge (join) that enclose two or more branches of process elements. Examples are shown in the figure 4.3(b) and 4.3(c)
Loop fragment: A loop fragment consists of a merge, followed by a sequence of tasks, followed by a decision, and optionally a sequence of rework tasks. An example is shown in the figure 4.3(d)

Unstructured: An unstructured fragment is a fragment that is not structured. Additionally, we classify this category into two sub-categories: planar unstructured fragment and non-planar unstructured fragment. An example is shown in the figure 4.3(e)

Using classification we can improve the layout algorithm provided by yFiles.

4.3 The Recursive Layout Algorithm

We already have the fragment classification. What we need to do is to come up with layout algorithms for them and combine these algorithms into one process model layout algorithm. As we already know, a process model is a fragment, which in turn contains other child fragments. Therefore we can lay out the child fragments separately and then combine the layout result recursively. The process model layout algorithm will have two phases. First, we need to calculate bounding boxes of fragment. Second, from the information about the bounding boxes, we can layout the process model.

Let’s us explain more about the first phase of the process model layout algorithm. A fragment, which is laid out using our layout algorithms, will have a bounding box that contains child fragments within it. This fragment may contain other child fragments. To calculate the bounding box of the parent fragment, its height and its width, we need to calculate the bounding boxes of the child fragments and then “lay” them out according to the parent fragment type. Of course, in this phase, we do not really lay the process element out but instead we use the layout algorithms only for calculating the bounding box.

In the second phase, since we have all the information about bounding box of fragment, all we need to do is to supply the (left, top) coordinate of the process model bounding box. Then we apply the fragment layout algorithms recursively together with pre-computed bounding boxes in the first phase. This time, their (left, top) coordinates are also calculated so we can lay the fragments out in their correct position.

Notice that for structured and unstructured fragments, we have different strategies to lay them out. The detailed strategies for laying out structured fragments are presented in section 4.4. For unstructured fragments, we test for their planarity. If they are planar,
we try to lay them out without any edge crossings. If they are non-planar, we will use yFiles to lay them out. The planarity testing algorithm will be discussed in section 5.1.

To make it easier to understand our process model layout algorithm, we will use one small example. In Figure 4.4, we have a process model which contains a sequence fragment at the highest level. This sequence includes three fragments: two single tasks (Check Claim and Record Claim) and one alternative fragment. The alternative fragment, in turn, contains two fragments which are a single task (Settle Claim) and a sequence fragment whose two single tasks (Reject Claim and Close Claim). The bounding boxes are represented by dash-lined red rectangles. To calculate the bounding box of the process model, we need to calculate the bounding boxes of its three fragments. The bounding boxes of the two single-task fragments are provided, we assume, by a BPMN editor program. In our case, it is WBM. For the alternative fragment’s bounding box, we need to calculate the bounding boxes of its child fragments. After having all the bounding boxes calculated, we can compute the bounding box of the process model using the sequence layout algorithm.

![Figure 4.4: Lay out Process Models Using Process Structured Tree](image)

**Algorithm 1**: layoutFragment(fragment f)

**input**: the process structure tree of the model

**input**: the bounding boxes of the fragments

```plaintext
begin
  if f is a structured fragment then
    Lay f out according to its type. During this process, layout each f’s child fragment f_i: layoutFragment(f_i)
  else
    if f is a planar fragment then
      Use Left-Right algorithm to lay f out. During this process, layout each f’s child fragment f_i: layoutFragment(f_i)
    else
      Use yFiles to lay f out. During this process, layout each f’s child fragment f_i: layoutFragment(f_i)
  end
end
```
4.4 Layout of Different Fragment Types

We have different layout algorithms for each type of fragment:

**Sequence fragment:** This fragment’s elements will be laid out from left to right. Each fragment’s elements will have a constant horizontal distance from each other. Moreover, the fragments will be aligned such that there is no bend points in the link connecting any two consecutive elements.

**Alternative fragment and Parallel Fragment:** This fragment’s elements will be laid out from left to right starting from the decision (fork) node, then the tasks, and finally the merge (join) node. The tasks will be laid out in the middle of the space formed by the decision (fork) node and the merge (join) node. Moreover, they are also laid out from top to bottom such that if we draw a line between the two gateways of the fragment, the tasks will occupy equal spaces above and below this line. Besides, the tasks have a constant vertical distance from each other.

**Loop fragment:** This fragment’s elements will be laid out from left to right starting from the merge node, then the tasks, and finally the decision node. The tasks of a loop fragment are classified into body tasks and rework tasks. The body tasks are laid out in the same fashion like we did for those in an alternative fragment. The rework tasks are laid out like a sequence fragment, starting from a position that below and to the right of the decision gateway. Similarly with the above layouts, the body tasks also have a constant vertical distance from each other and the rework ones have a constant horizontal distance from each other.

**Unstructured Fragment:** We classify this type of fragment into planar and non-planar ones. For planar fragments, we try to lay out them using the Left-Right algorithm, which will be introduced later in section 5.1. For non-planar ones, we use yFiles to lay out them.

There are subtleties in drawing a fragment’s links. For example, the gateway links need to be drawn neatly as shown in the figure 4.5(a). Also a sequence fragment’s links are drawn without any bends as shown in the figure 4.5(b). And for a loop fragment, the link between its rework task and merge node is drawn under the rework task as shown in the figure 4.5(b)
4.5 The Bounding Box Computing Algorithm

Algorithm 1 needs bounding boxes of a process model’s fragments for its inputs. These bounding boxes are calculated according to its corresponding fragment types. For example, let us consider a sequence fragment’s bounding box computation. Since, as we discussed in section 4.4, its fragments will be laid out from left to right and they will have a predefined constant horizontal distance from each other. Therefore the width of its bounding box will be calculated according the formula \( box\_width = total\_width\_of\_elements + (n - 1) \times horizontal\_distance \), where \( n \) is the number of its fragments. Its bounding box’s height is more difficult to calculate. We need to imagine that the sequence fragments are already laid out and the link between any two consecutive elements has no bends. This requires us to move up or down the sequence fragment’s elements to avoiding bending points. Then the bounding box’s height will be the difference between the highest coordinate and the lowest coordinate of fragment’s elements.

There are subtleties that need to be taken care of to achieve a good layout result. For example, for alternative fragment type, we need to have extra space to accommodate branch names of the decision node.

Similarly to algorithm 1, the bounding box computation algorithm is also recursive. Its pseudocode is shown in algorithm 2.
Algorithm 2: calculateBoundingBox(fragment f)

input: the process structure tree of the model

begin
1 if f is a structured fragment then
2 Calculate f’s bounding box according to its type. During this process, calculate each f’s child fragment’s bounding box f_i: calculateBoundingBox(f_i)
else
5 if f is a planar fragment then
6 Use Left-Right algorithm to calculate f’s bounding box. During this process, calculate each f’s child fragment’s bounding box f_i: calculateBoundingBox(f_i)
else
8 Use yFiles to calculate f’s bounding box. During this process, calculate each f’s child fragment’s bounding box f_i: calculateBoundingBox(f_i)
end

4.6 Parallel Edges between Process Elements

Links are laid out using WBM default algorithm which does not always produce optimal results. For example in figure 4.6a, we have parallel edges between two tasks which has crossing edges. This is due to the fact that WBM uses a simple heuristic for laying out edges. One improvement can be as in figure 4.6b. To address this problem, we have implemented a transformation that lays out parallel edges between tasks in the whole process models. This was done to become familiar with the API. For the next version, this algorithm will be improved and integrated with the layout algorithm discussed in section 4.3.

(a) Crossing Parallel Edges between Two Tasks  (b) Non-crossing Parallel Edges between Two Tasks

Figure 4.6: Crossing Edges vs. Non-crossing Edges
Chapter 5

Layout of Planar Fragments

To layout planar fragments, the first step is to test them for planarity. The second step is to construct its corresponding planar embedding. This embedding will allow us to lay planar fragments out. In planarity testing, there are two main directions: the vertex-addition pioneered by Lempel et al [20] and the path-addition approach pioneered by Hopcroft et al [21, 22]. Both are difficult to implement. To display a planar embedding is also a hard step [23]. Therefore we decide to choose the Left-Right Planarity Test, which is originally developed by de Fraysseix et al [14]. Ulrik Brandes later gives a more detailed explanation [24] together with some implementation improvements. The algorithm does not have complex data structures (e.g., [25]) or a complicated embedding phase (e.g., [26]). Moreover its runtime performance was confirmed to be the best [15]. Our description of the algorithm here bases on [24]. We also provides a runtime example to help readers understanding the algorithm more easily. Given a graph G, the Left-Right Test has two outputs:

1. Decide whether G is planar
2. If G is planar, find a planar embedding

5.1 The Left-Right Planarity Algorithm

We have the main algorithm that combine two main steps orientation and testing. The algorithm will return true and generate a graph embedding if the graph is planar, otherwise it will stop when it determines that the graph is nonplanar.
Algorithm 3: Left-Right Planarity Algorithm

output: true if the graph is planar, otherwise false

begin

// Orientation phase
for $v \in V$ do
    if $\text{height}[v] = \text{INFINITY}$ then
        $\text{height}[v] \leftarrow 0$
        add $s$ to the root list
        DFSO($v$)

// Testing phase
sort adjacency lists according to non-decreasing nesting depth
for $s \in \text{Roots}$ do
    testDFS($s$)

end

In order to have a better understanding of the algorithm, we will use the exemplary graph in figure 5.1(b) as the input for the Left-Right planarity algorithm. This graph was converted from an unstructured fragment shown in figure 5.1(a) using definition 1.1.

![Figure 5.1: Conversion of an Unstructured Fragment into a Graph G](image)

5.1.1 Orientation

The pseudocode of this orientation phase is provided in algorithm 4. It conducts a depth-first search on the graph $G$. This depth-first search classifies the edges in $G$ into
two set, the set \( T \) of tree edges and the set \( B \) of back edges [27]. This oriented edges form a DFS-oriented graph \( G^* = (T \cup B) \).

Several helpful data, which will be used in the testing phase, are also calculated as follows:

- **parent_edge**: an edge array that store parent edge of a vertex. If \((u, v)\) is a directed edge \(\in T\) from \(u\) to \(v\), then \((u,v)\) is the parent edge of \(v\). At the beginning, parent_edge values are initialized to null.
- **height**: an integer array that stores height of a vertex. Height value of a vertex is its distance from the root of DFS tree. At the beginning, height values are initialized to INFINITY.
- **lowpt**: an integer array that stores an edge’s lowest return point’s value. The return points of a tree edge \(v \rightarrow w \in T\) are the ancestors \(u\) of \(v\) with \(u \leftarrow v \rightarrow w \leftarrow x \rightarrow u\) for some descendant \(x\) of \(w\). A back edge \(v \leftarrow w\) has exactly one return point, its target \(w\). A back edge \(x \leftarrow u\) is a return edge for every tree edge \(v \rightarrow w\) with \(u \leftarrow v \rightarrow w \leftarrow x \rightarrow u\), and for itself. The value of a return point is its height from the root of the DFS-graph \(G^*\).
- **lowpt2**: an integer array that stores an edge’s second lowest return point’s value.
- **nesting_depth**: an integer array that stores an edge’s nesting depth value. This value mimics the partial order \(<\). It is twice the lowest return point of any cycle containing \(e\), plus one if \(e\) is chordal. Recall that to embed a planar graph, we need at each vertex an order of adjacent edges. The Left-Right algorithm computes this order using the partial order \(<\). We define that \(e_1 < e_2\) if and only if the lowpoint of \(e_1\) is strictly lower than that of \(e_2\). In case they have the same lowpoint but only \(e_2\) has another return point, we let \(e_1 < e_2\). If \(nesting\_depth(e_1) < nesting\_depth(e_2)\) then \(e_1 < e_2\).

Let us walk through the algorithm of this phase. At lines 2, the parent edge of vertex \(v\) is assigned to \(e\). From line 3 to line 35, we process each non-oriented incident edges \((u,v)\) of \(u\). First the non-oriented edge \((u,v)\) in the original graph \(G\) will be oriented from \(u\) to \(v\). Then depending on the type of \((u,v)\), we have different calculation for lowpoint and lowpoint2 values of \((u,v)\).

After calculation of lowpt, lowpt2 values, and edge orientation, at lines 20-23 we calculate the nesting_depth value of the newly oriented edge \((u,v)\) according to the nesting_depth, which equals twice the height value of value of lowpoint of \((u,v)\) and plus 1 if \(e\) is chordal.
Algorithm 4: DFSO(vertex u)

begin
  e ← parent_edge[u]

  foreach adjacent edge adjE of u do
    v ← adjE.target

    if adjE is oriented then
      continue
    else
      orient edge (u,v)
    end

  if height[v] = INFINITY then /* tree edge */
    lowpt[u][v] ← height[u]
    lowpt2[u][v] ← height[u]
    parent_edge[v] ← (u, v)
    height[v] ← height[u] + 1
    DFSO(v)
  else
    lowpt[u][v] = height[v]
    lowpt2[u][v] = height[u]
  end

  // calculate nesting_depth
  nesting_depth(u, v) ← 2 * lowpt(u, v)

  if lowpt2(u, v) < height[u] then /* chordal edge */
    nesting_depth(u, v) ← nesting_depth(u, v) + 1
  end

  update lowpoint value of parent edge e
  if e ≠ null then
    if (lowpt(u, v) < lowpt(e.source, e.target) then
      lowpt2(e.source, e.target) ←
      min{lowpt(e.source, e.target, lowpt2(u, v))}
    else
      lowpt2(e.source, e.target) ←
      min{lowpt(e.source, e.target, lowpt2(u, v))}
    end
  end

end
(e has more than one return point). Because \((u, v)\) is a newly explored edge, it will affect the \(\text{lowpt}\) and \(\text{lowpt2}\) values of parent edge \(e\) of \(u\). Therefore we need to update these 2 values at lines 25-34. For example, in Table 5.1, the lowpoint values of edge \((5,6)\) is updated after the outgoing edge \((6,2)\) of vertex 6 is processed.

We also give one example of runtime behavior of algorithm 4 based on the exemplary graph (see Figure 5.1) in Table 5.1 and the DFS-oriented graph \(G^*\) generated in this phase is shown in Figure 5.2. Let us walk through Table 5.1. At row 1, since the depth first starts at the vertex 0, we have the tree edge \((0,1)\) oriented and its lowpoint and lowpoint2 are 0 and 0 respectively. Similarly at rows 2-5, we have tree edges \((1,2), (2,3), (3,5), \) and \((5,6)\) oriented and their lowpoint values are also calculated. At row 6, we encounter the outgoing back edge \((6,2)\) of vertex 6. In this case its lowpoint values and nesting depth are calculated. At row 7, the algorithm then backtracks at vertex 5 and updates lowpoint values of the tree edge \((5,6)\). In this case its lowpoint values and nesting depth are calculated. At row 8, it again encounters another outgoing back edge \((6,3)\) of vertex 6. The lowpoint values and nesting depth of back edge \((6,3)\) are then calculated. Like at row 6, the algorithm updates lowpoint values of \((5,6)\) and since all of its outgoing edges have been processed, its nesting depth is also calculated. Continuing in this fashion, the algorithm will orient and calculate all the lowpoint values and nesting depth for edges in the original graph \(G\).
Algorithm 5: testDFS(vertex v)

begin

  $e \leftarrow parent\_edge[v]$

  foreach outgoing edge $e_i$ of $v$ do

    if stack $S$ is empty then
      $stack\_bottom(e_i) \leftarrow null$
    else
      $stack\_bottom(e_i) \leftarrow S.peek$
    end

    if $e_i$ is a tree edge then
      testDFS($e_i.target$)
    else
      $lowpt\_edge(e_i) \leftarrow e_i$
      Push a new conflict pair ($\emptyset, [e_i, e])$ onto stack $S$
    end

    if $lowpt(e_i) < height[v]$ then
      if $e_i$ is the first outgoing edge then
        $lowpt\_edge(e) \leftarrow lowpt\_edge(e_i)$
      else
        addConstraintsOf($e_i, e$)
      end
    end

  end

if $e \neq null$ then

  trimBackEdges($e.source$)

  side of $e$ is side of a highest return edge

  if $lowpt(e) < height[u]$ then

    $hL \leftarrow null$
    $hR \leftarrow null$

    if $S.peek.L \neq null$ then
      $hL \leftarrow S.peek.L.high$
    end

    if $S.peek.R \neq null$ then
      $hR \leftarrow S.peek.R.high$
    end

    if $hL \neq null$ AND ($hR = null OR lowpt(hL) > lowpt(hR)$) then

      $ref(e) \leftarrow hL$
    else
      $ref(e) \leftarrow hR$
    end

  end

end
5.1.2 Testing

As we see the tree edges in T cannot cross each other. If the graph G is non-planar or there are crossings of edges, it is because of the back edges. The Left-Right algorithm tries to partition back edges into two classes, referred to as left and right. The back edges in one class do not cross each other. The back edges in different classes can cross each other that is why they are put into two different classes. This partition of the back edges are called LR-parition [24]. We repeat its definition here for the reader’s convenience.

**Definition 5.1** (LR partition). Let \( G^* = (V, T \cup B) \) be a DFS-oriented graph. A partition \( B = L \cup R \) of its back edges into two classes, referred to as left and right, is called left-right partition, or LR partition for short, if for every fork \( u \rightarrow v \in T \) and \( e_1, e_2 \in E^+(v) \)

1. All return edges of \( e_1 \) ending strictly higher than \( lowpt(e_2) \) belong to one class and
2. All return edges of \( e_2 \) ending strictly higher than \( lowpt(e_1) \) to the other.

A consistent LR partition is an LR partition in which all the back edges of a tree edge that end at its lowpoint are on the same side. If the DFS-oriented graph \( G^* \) admits a consistent LR partition then it is planar otherwise, non-planar. Therefore the purpose of testing phase is to determine whether a consistent LR partition exists. Before conducting a second depth first search on the oriented graph \( G^* \), we sort adjacency list of \( G^* \) according to non-decreasing order of nesting_depth values. By doing this, we
achieve a partial order of outgoing edges. Later in the embedding phase, we will have a complete order around any vertex of G* if it is planar. The edge traversal order in the second depth first search is different from that in the first phase since we already sorted adjacency lists. This depth first search will halt if the graph is no-planar.

The idea is that we need to detect and maintain all the pairwise constraints among back edges. The pairwise constraints are associated with a tree edge and we have only 2 type of constraints: same- and different-constraint. If two edge are in one class, we say they subject to same-constraint. If two edge are in different classes, we say they subject to different-constraint. The back edges that belong to the same class will be linked together using a ref pointer (Figure 5.3(a)). Current implementation used a 2-dimensional matrix ref of reference edge to represent this linkage. For example, ref[u][v] will contain the reference edge of (u,v) and from that reference edge we can get its reference edge also. This linkage forms what we call an interval. Two intervals that subject to different-constraint forms a conflict pair (Figure 5.3(b)).

![Figure 5.3: Data Structures of LR Testing](image)

In algorithm 5 for testing planarity, we use the following data structures:

**S:** A global stack of conflict pairs. The second depth first search uses S to build incrementally the entire bipartition of edges by merging the conflict pairs. This is done by processing the DFS-tree G* bottom-up. The constraints associated
with an edge can be determined by merging those associates with its outgoing edges. S is a stack of conflict intervals denoted as L and R (left and right). One interval is formed by high and low edge.

**stack_bottom**: A conflict pair array that records the top of stack S when an outgoing edge $e_i$ of a vertex v is processing.

**lowpt_edge**: An edge array that contains the return edge of a tree edge and this return edge has the lowest return point.

Let us walk through algorithm 5. At line 2, e is set to the parent edge of v. Then for each outgoing edges $e_i$, we process them as follows:

1. We record the top of the stack S when we start processing $e_i$.
2. If $e_i$ is a tree edge, we recursively call algorithm 5 on its target.
3. If $e_i$ is a back edge, we create a conflict pair with empty left interval, and right interval with high and low both equal to $e_i$. Then we push it onto stack S. In this way, we start accumulating the back edges, we encounter when processing $e_i$.
4. If $e_i$ has return edges (lowpt($e_i < \text{heigh}(v)$)) then if $e_i$ is the first outgoing edge, which means it has the lowest return point between the outgoing edges since we sorted the outgoing edges according to nesting_depth order, we store $e_i$ in the lowpt_edge array. Otherwise we merge the constraints associated with $e_i$ with those associated with e.

After processing all outgoing edges of vertex v, we need remove all back edges from the conflict pairs on S that have the source of parent edge $e = u \rightarrow v$, which is u, as their return point because these are not the return edges of e or any lower tree edge, therefore they are not subjected to any constraints of unprocessed tree edges. This is done by calling the function trimBackEdges at line 24. For the details of this function, please refer to [24].

Finally, the side of the parent edge is to be the sign of its highest return edge, which is one of e’s return edges whose highest return point. This is done at lines 26-39.

To illustrate how algorithm 5 runs, we will give a runtime example of it on the oriented graph G in figure 5.2. The algorithm starts from the source node 0. It visits tree edges $(0,1), (1,2), (2,3), \text{and} (3,4)$. Then it visits the back edge $(4,1)$. This edge will be pushed onto stack S. Now the content of S will be $L = \text{Empty} \ R = (4,1) -(4,1)$. Since it is the first outgoing edge, lowpt_edge(3,4) will be lowpt_edge(4,1). The algorithm continues to
visit other outgoing edges of vertex 4. It visits (4,7), then (7,1). Again, (7,1) is a back edge and will be pushed onto S. Stack S now contains \( L = \text{Empty} \), \( R = (7,1) \). Similarly, the edge (7,3) will also be pushed onto S. There are no more edges to visit from vertex 7. The algorithm now adds the constraints of (7,3) with those of parent edge (4,7). After this step, the algorithm will trim all the back edges ending at vertex 4. In this case, there is no back edge in S that ends at 4. Hence S is intact. After trimming, the algorithm backtracks to vertex 3 and continues to process other constraints associated with the edge (4,7), (4,2) to form the constraints associated with edge (3,4). It keeps on processing the whole oriented graph \( G \) in a depth first search manner until it backtracks at vertex 0. Hence the graph \( G \) is planar.

5.1.3 Merging the Contraints of Back Edges

In line 19 of algorithm 5, we merge all the constraints of the next outgoing edge \( e_i \), which is a back edge of the tree edge \( e \). If we cannot merge these constraints, it means the graph is not planar and the testing algorithm will stop. The details of the merging procedure is shown in algorithm 6. This algorithm includes two main phases. First is to merge all the return edges of \( e_i \) on one side (right side in the algorithm) since they all end higher than the lowpt(e) (LR partition). During this phase, if there is a conflict pair of the stack S, which is full on both sides, then obviously these conflict return edges cannot be merged on one side. Therefore the graph is not planar (lines 6-10 of algorithm 6). Second is to merge the conflicting return edges of \( e_1, \ldots, e_{i-1} \) into the left interval. Return edges of \( e_1, \ldots, e_{i-1} \) with lowpoints higher than \( \text{lowpt}[e_i] \) are subject to pairwise same-constraints or different-constraints with respect to some return edges of \( e_i \). Therefore we need to merge these edges on the same side (left side in the algorithm). If there is such a pair that contains two intervals ending above \( \text{lowpt}[e_i] \), we will violate the previous constraints and hence the graph is not planar.

5.1.4 Embedding

If the graph is planar then in this phase a graph embedding will be constructed. Compared to other algorithms, the embedding construction phase of the left-right algorithms is very simple. The LR ordering of the outgoing edges is achieved by sorting them according to their nesting_depth on both sides. Since the outgoing edges are already sorted, when we travel the DFS forest for the third time, we will encounter back edges as required in the definition of the LR ordering.

For the oriented graph \( G \) in figure 5.2, we have its graph embedding and corresponding graph embedding data structure shown in figure 5.4. In figure 5.4(a), at each vertices of
Algorithm 6: Adding contraints associated with $e_i$

input: Edges: $e_i, e$

begin

\[ P \leftarrow (\emptyset, \emptyset) \]
\[ Q \leftarrow \text{null} \]

// merge return edges of $e_i$ into $P.R$

while $S\text{.peek}() \neq \text{stack\_bottom}(e_i)$ do

\[ Q \leftarrow S\text{.pop}() \]

if $Q.L \neq \emptyset$ then

\[ \text{swap } Q.L, Q.R \]

end

if $Q.L \neq \emptyset$ then

\[ \text{HALT: not planar} \]

else

\[ \text{if } \text{lowpt}(Q.R\text{.low}) > \text{lowpt}(e) \text{ then} \]

\[ \text{if } P.R\text{.high} = \emptyset \text{ then} \]

\[ \text{P.R.high} \leftarrow Q\text{.right}\text{.high} \]

else

\[ \text{ref}(P.R\text{.low}) \leftarrow Q.R\text{.high} \]

end

\[ P.R\text{.low} \leftarrow Q.R\text{.low} \]

else /* make consistent */

\[ \text{ref}(Q.R\text{.low}) \leftarrow \text{lowpt\_edge}(e) \]

end

end

// merge conflicting return edges of $e_1, \ldots, e_{i-1}$ into $P.L$

while conflict($S\text{.top()}\text{.L}, e_i$) OR conflicting($S\text{.top()}\text{.R}, e_i$) do

\[ Q \leftarrow S\text{.pop}() \]

if conflict($Q.R, e_i$) then

\[ \text{swap } Q.L, Q.R \]

end

if conflict($Q.R, e_i$) then

\[ \text{HALT: not planar;} \]

else /* merge interval below lowpt($e_i$) into $P.R$ */

\[ \text{ref}(P.R\text{.low}) \leftarrow Q.R\text{.high} \]

if $Q.R\text{.low} \neq \text{null}$ then

\[ P.R\text{.low} \leftarrow Q.R\text{.low} \]

end

if $P.L \neq \emptyset$ then

\[ P.L\text{.high} \leftarrow Q.L\text{.high} \]

else

\[ \text{ref}(P.L\text{.low}) \leftarrow Q.L\text{.high} \]

end

\[ P.L\text{.low} \leftarrow Q.L\text{.low} \]

end

if $P \neq \emptyset$ then

\[ S\text{.push}(P) \]

end

end
Algorithm 7: Determine whether Edge $e$ conflicts with Interval $I$

**input**: Interval $I$, edge $e$

**output**: true if $e$ conflicts with Interval $I$, otherwise false

begin
1. return ($I \neq \emptyset$ AND lowpt($I$.high) > lowpt($e$))
end

Algorithm 8: embedDFS(vertex $v$)

begin
1. foreach outgoing edge $e_i$ of $v$ do
2.  $w \leftarrow$ target($e_i$)
3.  if $e_i = \text{parent}_\text{edge}[w]$ then /* tree edge */
4.     make $e_i$ first edge in the adjacency list of $w$
5.     leftRef($v$) $\leftarrow e_i$
6.     rightRef($v$) $\leftarrow e_i$
7.     embedDFS($w$)
8.  else /* back edge */
9.     if side($e_i$) = 1 then
10.        place $e_i$ directly after rightRef($w$) in the adjacency list of $w$
11.     else
12.        place $e_i$ directly before leftRef($w$) in the adjacency list of $w$
13.        leftRef($w$) $\leftarrow e_i$
14.     end
15. end
end

For graph $G$, we have a circular order of edges starting from the unique incoming tree edge. For example, at vertex 3, we have a circular order of edges $(2,3), (7,3), (3,4), (3,5)$, and $(6,4)$. And we use a data structure, figure 5.4(b), to represent this embedding. It is a graph representation data structure. For each vertices, we store a linked list of edges that connect to it. We label incoming (outgoing) edges as in (out) ones. The first edge in any lists will be the unique incoming tree edge of the corresponding vertex. For example, at vertex 3’s linked list, we have $(3,2)$ as the unique incoming tree edge of vertex 3.

5.2 Drawing the Graph Embedding

We use graph embedding provided for us by the previous section to layout the unstructured fragment shown in figure 5.1(a). The result is shown in figure 5.5. As we can see, we do not get a good layout. This is due to the fact that a planar graph embedding does not provide a topological order between the graph elements. Moreover, we do not take into account the entry and exit links of the fragment when we test for planarity.
These links may cross the fragment’s links hence introducing new edge crossing. For example, the links between Task 5 and Stop node crosses the fragment’s links (figure 5.5). In chapter 8, we will discuss a possible solution to overcome this problem. For the experimental purpose, we decide to use yFiles to compute bounding box and lay out planar unstructured fragments.
Chapter 6

Layout of Non-planar Fragments

As we have the layout algorithm for structured fragments, we now turn our attention to the non-planar unstructured fragments. Since non-planar unstructured fragments have at least one edge crossing in their graph embedding, it is impossible to avoid edge crossing in the final layout result. Nevertheless, a non-planar unstructured fragment still has a main direction or a flow from left to right. We can exploit this property to achieve the desired layout result. Therefore we decide to use yFiles’s hierarchical layout algorithm. This algorithm portrays the precedence relation of directed graphs, hence can highlight the flow of a non-planar fragment. We also need to integrate the non-planar layout algorithm into our existing layout algorithm since a business process model includes both planar fragments and non-planar fragments.

The yFiles library provides a hierarchical layout algorithm for directed graphs (digraph). This algorithm aims to highlight the main direction or flow within a directed graph therefore it is suitable for BPMN. First, we need to create a graph that reflects the process model. For each process model’s elements, we create a graph node. Then for each directed links between process model’s elements, we create a directed edge between the corresponding nodes. Second, this graph will be the input for yFiles hierarchical layout algorithm, which takes several parameters such as orientation, type of edge routing, minimal edge distance. In our case, we provide left-to-right orientation and orthogonal edge routing. This algorithm generates the graph layout, which provides the graph’s bounding box and the relative position of graph’s node within this bounding box (see figure 6.1). With the bounding boxes for non-planar fragments calculated, we can calculate the bounding box for the whole process model using our existing algorithms. Hence, we integrate the yFiles hierarchical layout algorithm with our algorithm nicely.
Figure 6.1: Using yFiles to Lay Out Non-planar Fragments
Chapter 7

Evaluation

In this chapter, we present our findings about different fragment type percentages. We also conduct experiments to compare our layout algorithm against the current used one in WBM. Moreover, we also compare our layout result against other popular BPMN editors. In order to compare the layout results, we choose several sample process models which includes various types of fragments then we use the aesthetics criteria introduced in section 2.3 to judge the quality of layout results. Finally, we summarize all the evaluation results to show that our layout is good.

7.1 Fragment Classification and Planarity Analysis

We have analyzed 677 business process models to quantify the percentages of different fragment types. Moreover, for unstructured fragments, we conduct a planarity testing. As result illustrated in figure 7.1 shows, we have a domination of structured fragment. They occupy 72.7% of all the fragment types. Therefore, by focusing on laying out structured fragment like we did, we can achieve a good layout result for process models.

Figure 7.1: Fragment Classification Result
Within unstructured fragments, we have the number of planar fragments (22%) dominated the number of non-planar fragments (5.3%). Hence there is a good opportunity that we can achieve better layout for process models by laying out planar fragment nicely.

### 7.2 Against WBM Auto-layout and Other Business Process Editor Layout

We pick four sample process models and compare layout results achieved by our layout algorithm and the same by WBM auto layout and other business process editors using the aesthetics criteria proposed in 2.3. These processes although are small but representative. They include different fragment types combined together when modeling a process.

#### 7.2.1 WBM Auto-layout

![Figure 7.2: Process Model 1- A loop fragment nested within a parallel one](image)
The comparison result is summarized in table 7.1. In most of our layout results, although our process models occupy more space than the ones laid out by current WBM auto layout, they have more symmetry, and less edge crossings. Except for process model 3, we have more edge crossings. These crossings occur in the unstructured fragment (see the red rectangle in figure 7.4) that is laid out using yFiles. Hence we cannot avoid them. Overall, we have achieved a better layout.

Let us walk through different process models and compare in details the layout results. For convenience, we shall refer to our layout results as our layout, and the WBM’s ones as WBM layout.

In figure 7.2, we have a loop fragment which consists of 2 single tasks and a loop fragment. It is the loop fragment that gives troubles to the WBM auto layout due to its back edge which forms a cycle and cycles always make graph drawing difficult. As you can see from figure 7.2, WBM layout uses a hierarchical layout algorithm that treats process elements, gateways and tasks, as graph nodes and assigns them to different layer. Therefore WBM layout is not aware of process model semantics. Hence it tends to produce asymmetric layout. Additionally, WBM layout does not avoid edge crossings even in a simple case like this one. Whereas, our layout has a nice symmetry and no edge crossings.

![Figure 7.3: Process Model 2 - A loop fragment followed by a parallel one](image-url)
In figure 7.3, WBM also does not give good edge routing result. It has an unnecessary edge crossing. Our layout solved this problem.

![Figure 7.3: WBM Layout](image)

(a) WBM Layout

![Figure 7.4: Our Layout](image)

(b) Our Layout

**Figure 7.4:** Process Model 3 - An unstructured fragment nested within a parallel one

In figure 7.4, we have a process model which consists of a parallel fragment that includes sequence fragments and an unstructured fragment (marked by red rectangle in Our Layout subfigure). This unstructured fragment, in turn, consists of several gateways and a sequence fragment (marked by green rectangle in Our Layout subfigure). For the structured fragments, we use our hierarchical algorithm to lay them out. For the unstructured fragment we use yFiles to lay it out. This demonstrate our algorithm’s ability to combine different strategies to lay out a process model. Regarding the layout, our layout produce less number of bends. Although it has more number of edge crossings, this is due to yFiles since we use it to layout the unstructured fragments. This motivates us to find a better way to lay out unstructured fragments. In the chapter 8, we will discuss more in details.

In figure 7.5, the process model is big. It has a large alternative fragment, which in turn contains a parallel fragment. As we can see, WBM layout have not only edge crossings but also edges overlapping with process’s elements. On the contrary, our algorithm produce symmetrical, no-crossings, no-overlapping layout.
From table 7.1, we can see that overall there is a trade-off between minimizing the number of edge crossing and number of bends. Trying to reduce one will increase the other. We decide to give priority to avoiding edge crossings since we consider they cause more confusion for the users.

Table 7.1: WBM Auto-layout vs. Our Layout

<table>
<thead>
<tr>
<th></th>
<th>WBM Auto-layout</th>
<th>Our Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Model 1</td>
<td>#crossings = 2, #bends = 16</td>
<td>#crossings = 0, #bends = 18</td>
</tr>
<tr>
<td>Process Model 2</td>
<td>#crossings = 1, #bends = 17</td>
<td>#crossings = 0, #bends = 14</td>
</tr>
<tr>
<td>Process Model 3</td>
<td>#crossings = 0, #bends = 40</td>
<td>#crossings = 2, #bends = 36</td>
</tr>
<tr>
<td>Process Model 4</td>
<td>#crossings = 3, #bends = 39</td>
<td>#crossings = 0, #bends = 42</td>
</tr>
</tbody>
</table>

Besides compare our algorithm with WBM auto-layout one, we also pick two popular
BPMN editors to compare our layout result with theirs. They are Business Process Visual Architect (Visual Paradigm) and Borland Together.

### 7.2.2 Business Process Visual Architect (Visual Paradigm)

Business Process Visual Architect (BPVA) [28] provides various layout style such as hierarchical layout, orthogonal layout, or organic layout. It also provides auto layout feature. Again, like in section 7.2, we use the same 4 process models to compare BPVA’s layout result with our layout result.

![BPVA Layout](image1.png) ![Our Layout](image2.png)

**Figure 7.6: Process Model 1: BPVA Layout vs. Our Layout**

Although BPVA provides auto layout feature, we decide to compare our layout against its hierarchical layout instead. The reason is that BPVA’s auto layout does not provide a consistent layout result. Its result depends on the locations of the process elements at the time we invoke the auto layout feature. Whereas BPVA’s hierarchical layout provides much better and consistent result. Moreover it is suitable for BPMN modeling convention imposing the order between process elements. Despite BPVA’s hierarchical layout draws process model in a top-down fashion, for the sake of comparison, we consider this as equivalent to the left-right fashion. As figure 7.6 shows, BPVA does not provide a good layout even for this simple process model. We have overlapping edge and the semantics of loop fragment is not shown clearly. This is most probably due to the fact that BPVA is not aware of the loop fragment or any semantics of the process model and its layout algorithm only treats the process elements as graph nodes.

In figure 7.7, BPVA layout result is more compact but it does not present to the user a good structure of the process model. In particular, one of loop fragment’s task is drawn on the same level with the parallel fragment’s task causing confusion for users.
Figure 7.7: Process Model 2: BPVA Layout vs. Our Layout

In figure 7.8, BPVA layout result again does not show a good structured of the process model. Its layout result contains even overlapping between a task and an edge.

In figure 7.9 BPVA layout result also suffer from the same symptoms like the previous cases, i.e., unnecessary edge crossings.

In table 7.2, we have statistics of BPVA layout results versus our layout result. We can see our algorithm does a much better job in minimizing the number of edge crossings. The reason BPVA layout has much less number of bends is partly due to BPVA notation for BPMN gateways. Its gateway representation does not have branches whereas WBM’s one has.
Chapter 7. Evaluation

Figure 7.9: Process Model 4: BPVA Layout vs. Our Layout

Table 7.2: BPVA Layout vs. Our Layout

<table>
<thead>
<tr>
<th>BPVA Auto-layout</th>
<th>Our Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Model 1</td>
<td>#crossings = 1, #bends = 4</td>
</tr>
<tr>
<td>Process Model 2</td>
<td>#crossings = 1, #bends = 7</td>
</tr>
<tr>
<td>Process Model 3</td>
<td>#crossings = 2, #bends = 19</td>
</tr>
<tr>
<td>Process Model 4</td>
<td>#crossings = 3, #bends = 25</td>
</tr>
</tbody>
</table>

7.2.3 Borland Together

We use the same sample models to compare our layout against Borland Together [29] layout. As you can see from figure 7.10 to figure 7.13, our layout presents the models more symmetrically and with less number of edge crossings. From the table 7.3, we notice that the number of bends in Borland Together layout is very low compared to our layout. This is due to the polyline drawing style that Borland Together adopted. This drawing style help reducing the number of bends but at the same time does not produce a nice edge routing compared to the orthogonal style.

Table 7.3: Borland Together Layout vs. Our Layout

<table>
<thead>
<tr>
<th>Borland Together Layout</th>
<th>Our Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Model 1</td>
<td>#crossings = 0, #bends = 3</td>
</tr>
<tr>
<td>Process Model 2</td>
<td>#crossings = 0, #bends = 2</td>
</tr>
<tr>
<td>Process Model 3</td>
<td>#crossings = 4, #bends = 12</td>
</tr>
<tr>
<td>Process Model 4</td>
<td>#crossings = 0, #bends = 9</td>
</tr>
</tbody>
</table>
Figure 7.10: Process Model 1: Borland Together Layout vs. Our Layout

Figure 7.11: Process Model 2: Borland Together Layout vs. Our Layout
Figure 7.12: Process Model 3: Borland Together Layout vs. Our Layout

Figure 7.13: Process Model 4: Borland Together Layout vs. Our Layout
7.2.4 Comparison Summary

For your convenience, in table 7.4, we have a summary of all evaluations shown in the previous sections. As you can see, in most of the cases, our layout results have less edge crossings. Although we have more bend points in our layouts, we achieve more symmetric process models. Additionally, our process model layouts show clearly the clusters of related activities. For example, in figure 7.5, the nested alternative node, which has a big parallel fragment in it, was laid out in a much clearer way by our layout than WBM layout.

<table>
<thead>
<tr>
<th>Process Model</th>
<th>WBM Auto-layout</th>
<th>BPVA Auto-layout</th>
<th>Borland Together Layout</th>
<th>Our Layout</th>
</tr>
</thead>
<tbody>
<tr>
<td>Process Model 1</td>
<td>#crossings = 2, #bends = 16</td>
<td>#crossings = 1, #bends = 4</td>
<td>#crossings = 0, #bends = 1</td>
<td>#crossings = 0, #bends = 18</td>
</tr>
<tr>
<td>Process Model 2</td>
<td>#crossings = 1, #bends = 17</td>
<td>#crossings = 1, #bends = 7</td>
<td>#crossings = 0, #bends = 2</td>
<td>#crossings = 0, #bends = 14</td>
</tr>
<tr>
<td>Process Model 3</td>
<td>#crossings = 0, #bends = 40</td>
<td>#crossings = 2, #bends = 10</td>
<td>#crossings = 4, #bends = 12</td>
<td>#crossings = 2, #bends = 22</td>
</tr>
<tr>
<td>Process Model 4</td>
<td>#crossings = 3, #bends = 39</td>
<td>#crossings = 3, #bends = 25</td>
<td>#crossings = 0, #bends = 9</td>
<td>#crossings = 0, #bends = 42</td>
</tr>
</tbody>
</table>
Chapter 8

Future Work

In this chapter, we present two possible improvements of our layout algorithm. First is to make our process model layout more compact. Second is to apply the upward planarity notation in laying out the planar unstructured fragment.

Figure 8.1: Process Model Compactness
8.1 Compactness

Our process models layout are not always compact. We can see this clearly from the figure 8.1, the red rectangle areas are not utilized. The reason is that in our hierarchical layout algorithm, we keep bounding boxes of process model fragments separately. To reduce the occupied space of process model, we can allow these bounding boxes to overlap each other. For example, with the process model in figure 8.1, we can squeeze the two bounding boxes of the two sequence fragments (two black rectangles) closer to each other, allowing them to overlap. This way we will save the wasted red rectangle area hence achieve more compact process models.

8.2 Upward Planarity

As we already know from chapter 5.2, it is not sufficient to check for planarity of process model fragments and then lay them out without edge crossings. With its directed nature, it is more suitable to check for its upward planarity property. A graph is upward planar if it admits a planar and upward drawing. That is all the directed edges are monotonically increase in the vertical direction and there are no edge crossings. Since a process model fragment is a single source (entry) and single sink (exit) digraph, if it admits an upward planar drawing, this drawing will not have the problem of introducing edge crossings outside the fragment drawing. In figure 8.2(a), we have one example of upward planar graph. While in figure 8.2(b), we can see a directed planar graph does not necessary admit an upward and planar drawing at the same time. There are several algorithms for
testing upward planarity. In [30], the authors give an $O(n^2)$-time testing algorithm for digraphs that have a single source. Bertolazzi et al. [31] improve this result by showing an $O(n)$ testing algorithm for the same class of digraphs.

In [32], the authors present layout algorithm for UML activity diagram. This algorithm does use upward planarity criteria in one of its phases. And since UML activity diagrams also share some similar structures such as decision gateway, activity nodes with business process model, we expect the same idea can be applied to layout process models.
Chapter 9

Summary and Conclusion

In this thesis, we have shown the importance of business process modeling, in particular business process models. They are used as tools in discussion about the actual processes in an organization, or as references when implementing softwares. Therefore, to understand them is very important. An automatic layout algorithm that produces good layouts of business process models will contribute to our understanding of them.

By investigating different graph drawing libraries yFiles, graphViz, and Pigale, we realize that normal layout style such as hierarchical, circular, or organic layout algorithm does not work well on business process models. The problem lies in the inherent order between business process model elements and clusters of them. If we want to achieve a good layout for process models, we must take into account these properties while designing a layout algorithm. In this project we have developed our own hierarchical layout algorithm that respects the order and clusters of business process model elements. Our algorithm is based on a hypothesis that is a hierarchical decomposition of a process model produces a better layout of the business process models. The decomposition of a process model provides a set of fragments. We classify these fragments and apply suitable layout algorithms to each fragment type. Among different fragment types, unstructured fragments are the hardest to layout. This is obviously due to its nature.

We again classify this fragment type into two categories: planar and non-planar ones. With planar unstructured fragments, we try to lay them without any edge crossings. While for non-planar fragments, we use the hierarchical layout algorithm from yFiles. The result from laying out planar unstructured fragment shows that it is not enough to consider planarity criteria since it will introduce other edge crossings outside the fragment itself. As a workaround, we fall back on yFiles to layout planar unstructured fragments. In chapter 8, we introduce one possibility to overcome this problem: the upward planarity criteria. In chapter 7, we conduct two main experiments. First is to quantify the percentage of different fragment types. Second is to compare our layout
result against WBM and other business process editors. In the first experiment, we
discover a dominance of structured fragments which supports our hypothesis. Moreover,
we also have a dominance of planar unstructured fragments. This motivates our future
work of trying to laying out them more nicely. In the second experiment, we confirm
our layout algorithm performance. That is it achieves better results compared to not
only the current auto layout of WBM but also other popular business process editors’
layout.

By decomposing business process models into fragments and classify these fragments, it
allows us to combine different layout strategies for different fragment types. This distin-
guishes our approach from others. In the future, if there are new structured fragment
types, all we need to do is to design new layout strategies for them and combine these
strategies into our existing hierarchical layout algorithm rather than design a new layout
algorithm from scratch.
Bibliography


