

CZECH TECHNICAL UNIVERSITY IN PRAGUE
FACULTY OF ELECTRICAL ENGINEERING



DIPLOMA THESIS

Diagnostic Tool for Industrial Networks
Based on PROFINET

Prague, 2012

Author: Petr Ladman

Declaration

I declare that this diploma thesis is my own work and I used only literature quoted in the attached reference list.

In Prague, _____

Signature

Acknowledgements

I would like to thank my supervisor Ing. Pavel Burget, Ph.D. for his guidance and useful comments. I would also like to thank Ing. Martin Žídek and Ing. Jiří Čihák from ANF DATA spol. s r.o. for their support and helpful advice.

Abstrakt

Tato diplomová práce se zabývá implementací nové komponenty pro kreslení topologie průmyslové sítě. Kreslení topologie se skládá ze dvou částí. První částí je nalezení vhodných pozic pro uzly a druhou částí je nalezení cest pro hrany mezi uzly. Komponenta byla doplněna o vlastnosti porovnávání topologie. Pro tento účel byly navrženy dva algoritmy. Výstupem prvního algoritmu je topologie, která obsahuje uzly a hrany ze dvou porovnávaných topologií. Případné odlišnosti jsou zvýrazněny při jejím vykreslení. Druhý algoritmus porovnává topologii průmyslové sítě s topologií nakonfigurovanou v prostředí SIMATIC STEP 7. Účelem tohoto algoritmu je nalezení zařízení v síti odpovídajícího nakonfigurovanému. Komponenta byla implementována v jazyce C# v prostředí Visual Studio 2010 a integrována do diagnostického nástroje PRONETA.

České vysoké učení technické v Praze
Fakulta elektrotechnická

katedra řídicí techniky

ZADÁNÍ DIPLOMOVÉ PRÁCE

Student: **Bc. Petr Ladman**

Studijní program: Kybernetika a robotika
Obor: Systémy a řízení

Název tématu: **Diagnostický nástroj pro průmyslové sítě PROFINET**

Pokyny pro vypracování:


1. Seznamte se se softwarovým nástrojem PRONETA pro diagnostiku průmyslových sítí typu PROFINET.
2. Seznamte se s dostupnými algoritmy a softwarovými knihovnami pro automatické kreslení diagramů.
3. Proveďte analýzu možností pro porovnávání síťových topologií v sítích PROFINET.
4. Vyberte knihovnu či algoritmus pro účely vykreslení topologie průmyslové sítě a implementujte s jeho pomocí novou komponentu pro kreslení topologie pro nástroj PRONETA v jazyce C#.
5. Doplněte komponentu o vlastnosti porovnávání topologie.

Seznam odborné literatury:

Giuseppe Di Battista, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. 1994. Algorithms for drawing graphs: an annotated bibliography. Comput. Geom. Theory Appl. 4, 5 (October 1994), 235-282.

Vedoucí: Ing. Pavel Burget, Ph.D.

Platnost zadání: do konce letního semestru 2012/2013


prof. Ing. Michael Šebek, DrSc.
vedoucí katedry




prof. Ing. Pavel Ripka, CSc.
děkan

V Praze dne 19. 12. 2011

Abstract

This thesis is concerned about the implementation of a new component for the drawing of industrial network topology. The drawing of topology consists of two parts. The first part is the calculation of the positions of vertices and the second part determines the edge routes between these vertices. The component was supplemented with functionality for the topology comparison. Two algorithms were implemented for this purpose. The output of the first algorithm is a merged topology which consists of the vertices and the edges from two compared topologies. Possible differences are marked in the visualization. The second algorithm compares the topology of industrial network and the topology which were configured in SIMATIC STEP 7. The purpose of this algorithm is to find a device in the network corresponding to the configured device. The component has been implemented in C# in Visual Studio 2010 and integrated to the PRONETA diagnostic tool.

Contents

List of Figures	ix
List of Tables	xi
List of Algorithms	xii
List of Abbreviations	xiii
1 Introduction	1
1.1 Motivation	1
1.2 Problem Statement	2
1.3 Contribution	3
2 PRONETA	5
2.1 Functionality Overview	5
2.2 Information Problems	6
3 Algorithms and Frameworks	8
3.1 Algorithms	8
3.1.1 Hierarchical Drawings	8
3.1.2 Rooted Tree	9
3.2 Frameworks	10
3.2.1 Requirements	10
3.2.2 Considered Frameworks	10
3.2.2.1 MSAGL - Microsoft Automatic Graph Layout	10
3.2.2.2 yFiles	11
3.2.2.3 Graphviz	12
3.2.2.4 Quickgraph	13

3.2.2.5	Graph \sharp	13
3.2.3	Chosen Framework	13
4	Topology Drawing	15
4.1	Layout Algorithm	15
4.1.1	Solution	16
4.1.1.1	Graph Initialization	16
4.1.1.2	Root Vertex	17
4.1.1.3	Tree Layout	18
4.1.1.4	Final Placement	22
4.1.2	Optimization	25
4.2	Routing Algorithm	28
4.2.1	Solution	28
4.2.1.1	Ring Detection	28
4.2.1.2	Orthogonal Visibility Graph	30
4.2.1.3	Pathfinding	34
4.2.1.4	Post-Processing	36
4.2.2	Optimization	39
4.2.2.1	Appearance Optimization	39
4.2.2.2	Performance Optimization	39
5	Topology Comparison	42
5.1	Comparison Algorithm for Visualization of Changes	42
5.1.1	Matching Vertices	43
5.1.2	Merged Topology	43
5.2	Comparison Algorithm for Name Assignment	46
5.2.1	Largest Similar Subgraphs	46
5.2.2	Nearest Different	47
5.2.3	Match priority	49
6	Component Structure	51
6.1	C \sharp and WPF	51
6.2	Topology Control	52
6.2.1	Structure of Topology Control Library	52
6.3	Graph \sharp	54
6.3.1	GraphSharp	54

6.3.2	GraphSharp.Controls	55
6.4	WPF Extensions	55
6.5	WPF Toolkit	56
6.6	Proneta.Domain and PronetaUtilities	56
7	Graphical User Interface	57
7.1	Overview	57
7.1.1	Topology View	59
7.1.2	Visualization of Topology Comparison	61
7.2	Configuration View	62
7.2.1	Overview	62
7.2.2	Configuration Selection	63
7.2.3	Device Name Assignment	64
7.2.4	Reconstructed switch	65
8	Conclusion	67
	References	71
A	Content of the Attached CD	I

List of Figures

1.1	Graph terminology	2
2.1	PRONETA	7
3.1	Graph visualization using MSAGL	11
3.2	Graph visualization using yFiles	12
3.3	Graph visualization using Graphviz	13
3.4	Graph visualization using Graph# and Sugiyama layout algorithm	14
4.1	Removed unnecessary edge crossings	17
4.2	Vertices placed in a grid	19
4.3	Determining the vertex position	20
4.4	Tree layout with ordered steps	21
4.5	Determining the positions of disconnected vertices	24
4.6	Visualization of the topology without edges	25
4.7	Optimized tree layout	27
4.8	Not optimized tree layout	27
4.9	Ring detected in the topology	29
4.10	Important nodes in a visibility graph	30
4.11	Horizontal segments and nodes after vertical scan	31
4.12	Final visibility graph	32
4.13	Order of the edge segment types	36
4.14	The edge routes without and with post-processing	37
4.16	AA tree skew method from [17]	40
4.17	AA tree split method from [17]	40
4.15	Appearance optimization	40
4.18	Subgraph optimization	41
5.1	An example of the nearest different evaluation	49

6.1	Diagram of dependencies between projects and used libraries	52
6.2	The Graph# architecture	54
7.1	Vertex representing a device in visualization.	58
7.2	Topology View	59
7.3	Visualization of the merged topology	61
7.4	Configuration View	62
7.5	Selected configured vertex	64
7.6	Name assignment	65
7.7	Reconstructed generic Ethernet switch in the topology.	66

List of Tables

7.1	Available zooming functions.	58
7.2	Available functions in Topology View.	60
7.3	Available functions in Configuration View.	63

List of Algorithms

1	Finds new root vertex	18
2	Calculates the position of the vertex with at least one edge	20
3	Calculates the position of the vertices with at least one edge	22
4	Calculates position of vertex without edges	23
5	Calculates positions of vertices without edges	24
6	Calculates optimal positions of vertices with at least one edge	26
7	Explores edges and marks edges in the ring	29
8	Vertical scan which creates the horizontal segments	32
9	Horizontal scan which creates the vertical segments and their intersections	33
10	Creates the node connections and the intersections of the segments . . .	34
11	Finds paths for each edge through visibility graph	35
12	Determines level of the edge segment	37
13	Calculates the final position of the edge routing points	38
14	Creates new vertices of the merged topology.	44
15	Adds edges to the merged topology	45
16	Compares edges until the first difference is found	47
17	Calculates number of vertices to the nearest different	48
18	Determines priority of the matching devices	50

List of Abbreviations

DCE/RPC Distributed Computing Environment/Remote Procedure Call

DCP Discovery Control Protocol

DOT Plain text graph description language

EPM DCE/RPC Endpoint Mapper

GDI+ Graphics Device Interface

GLEE Graph Layout Execution Engine

GSDML GSD Markup Language

IP Internet Protocol

LIFO Last In First Out

MAC Media Access Control

MSAGL Microsoft Automatic Graph Layout

PRONETA PROFINET Network Analyzer

SNMP Simple Network Management Protocol

WPF Windows Presentation Foundation

XAML Extensible Application Markup Language

XML Extensible Markup Language

XPS XML Paper Specification

Chapter 1

Introduction

1.1 Motivation

Industrial networks used to control production machinery require high availability to keep possible productivity losses to a minimum. The challenges to overcome are how the needed device information can be acquired and how it can be processed and passed on, in order for the detected error to be quickly fixed. This thesis is focussed on the second challenge to provide information about the current state of network in a visual form. Service personnel using the diagnostic tool, developed within this thesis, will be able to quickly and efficiently identify and fix detected errors.

PROFINET is one of the most widespread industrial network protocols based on Ethernet. It is the open industrial Ethernet standard of PROFIBUS & PROFINET International (PI). Advantages of Ethernet are combined with industrial experience of PROFIBUS. Siemens AG is one of the leading companies in the production of technologies based on this protocol. It provides diagnostic tools based on SIMATIC WinCC to display information about devices in PROFINET networks which were configured in SIMATIC STEP 7 project as stated in [24].

To analyze whole network topology it is necessary to retrieve information about all devices, even if they are not present in the current project or they are not properly configured. PROFINET Network Analyzer (PRONETA) meets this requirement. It is being developed by ANF DATA spol. s r.o. which is a subsidiary company of Siemens

AG Austria. As the name suggests PRONETA is a diagnostic tool used for industrial networks based on PROFINET. It is capable of retrieving information about all devices in the network, even if they don't have an assigned name or IP¹ address. A brief introduction to PRONETA and its functionalities will be presented in the next chapter.

This thesis is concerned about the implementation of a new component for drawing of industrial network topology. PRONETA already contains a component for topology visualization, but it uses obsolete rendering technologies and will be replaced with this new component. The developed component is supposed to be used not only for drawing of topology but it shall also provide useful functionality for comparison of topologies.

1.2 Problem Statement

The study of network topology uses graph theory. Topology is then represented by a graph. This graph is a collection of vertices and edges which connect pairs of vertices. In the following text each vertex represents a device and each edge represents a connection in the network. Device ports are connection ports to each vertex. This port number is assigned to this port. The visualization of this change of the terminology is shown in Figure 1.1.

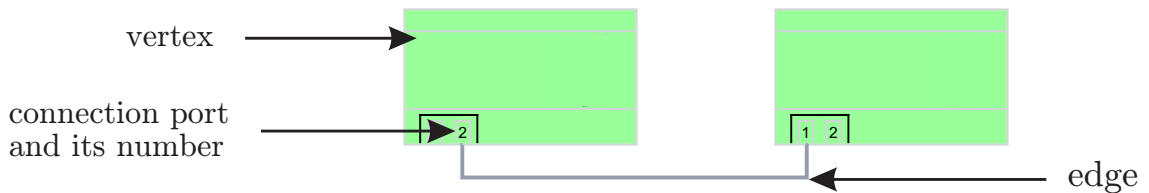


Figure 1.1: Graph terminology

The implementation of the visualization component introduces three main problems. First of all, algorithms for the topology drawing need to be created. It consists of the implementation of a layout algorithm and edge routing algorithm. The layout algorithm determines positions of vertices. Final layout has to achieve maximum possible readability and place vertices in a way to avoid unnecessary edge crossings. Edge routing algorithm determines edge routes between vertices. Edge route is visual representation of edge. According to the requirements the edges have to be routed orthogonally, and have to

¹Internet protocol

prevent overlapping of the vertices. Moreover, the edge routes have to be recalculated during the manipulation with vertices.

Secondly, algorithms for topology comparison need to be created. The first algorithm has to compare two topologies to create one merged topology. The merged topology will be used for visualization of recognized differences. It consists of all unique vertices and edges existing in both compared topologies. Therefore the only complication is to find matching vertices and edges. This topology comparison will detect changes between a previous state and the current state of the same network. The second algorithm has to compare the analyzed network topology and the topology configured in the SIMATIC STEP 7 project. This comparison is much more difficult because of the high probability of missing information about physical devices in the analyzed network. This problem will be described in the next chapter. Nevertheless, this algorithm will be used for finding proper matching vertices for device name assignment.

Finally, the graphical user interface (GUI) has to be implemented. This task is no less important than all previous. Maximum attention has to be focused on the clarity of the final GUI. It has to be obvious for the user how to use all available functions without having to read a long list of instructions. The final appearance of vertices and edges in visualization was defined by ANF DATA spol. s r.o.

1.3 Contribution

The primary object of this thesis is to implement a component for topology drawing and integrate it into PRONETA. This component will be supplemented with functionality for topology comparison. It will be implemented in C# with the use of WPF for rendering.

A unique name has to be assigned to each device in PROFINET networks. The controller is then able to automatically assign IP addresses from the SIMATIC STEP 7 project to each device. Because of this requirement, the task of this thesis was extended with an implementation of a user interface for device name assignment. Functionality for transmitting the configuration to physical devices is already present in PRONETA.

This thesis is divided into the following chapters:

- CHAPTER 1: Presents the current state of network diagnostics in modern industrial solutions.
- CHAPTER 2: Provides a basic description of PRONETA and its functionality.

- CHAPTER 3: Describes available algorithms and compares available frameworks for graph visualization and it also presents a framework which was used in the developed component.
- CHAPTER 4: Describes core algorithms for topology visualization.
- CHAPTER 5: Presents two different algorithms for topology comparison. The first creates a merged topology for visual comparison of differences between two topologies. The second finds proper device matches between network topology and configured topology for name assignment.
- CHAPTER 6: Basic description of structure of the created component.
- CHAPTER 7: Describes graphical user interface and use of a name assignment.
- CHAPTER 8: This chapter summarizes all important algorithms and functionality of the developed component. It also contains several suggestions for improvements which can be implemented in the future.

Chapter 2

PRONETA

As we mentioned before, PROFINET Network Analyzer (PRONETA) is a diagnostic tool used for industrial networks based on PROFINET. The idea of this diagnostic tool is that any engineer can analyze any PROFINET network with a common computer using a standard Ethernet network card, Ethernet cable and PRONETA. In this chapter we present functionality related to the implementation of a new component for topology visualization.

2.1 Functionality Overview

PRONETA communication functionality is based on the WinPCAP library. This library allows PRONETA to capture and transmit network packets. This is used by Network Scanner which performs cyclic scanning to monitor the network and keeps information about its current state updated. It is used for acquiring device information using the following communication protocols:

- **DCP** - Discovery Control Protocol - A communications protocol that allows to find every PROFINET device on a network. It retrieves information for identification of devices such as the name of station, device type, ip address etc.
- **SNMP** - Simple Network Management Protocol - Through SNMP the LLDP-MIB¹ can be accessed which is the interface between the LLDP agent and the network management station. LLDP-MIB keeps information about devices directly

¹Link Layer Discovery Protocol-Management Information Base

connected to the ports of the analyzed device. It provides port id, name of station and MAC² address of these devices. We also retrieve information about the network load.

- **DCE/RPC** - Distributed Computing Environment/Remote Procedure Call - This communication protocol is used to acquire I&M³ records. It provides information about the firmware and hardware versions and vendor. It can also retrieve additional information about hardware configuration of an analyzed device.

Once the Network Scanner receives information about all the devices in the network, Device List will open automatically. Device List shows all the detected devices in the table and detailed information about each of them.

PRONETA is also capable of parsing exported SIMATIC STEP 7 project in XML format into the internal instance of topology. This topology will be used for name assignment in the new component.

PRONETA already contains a component for topology drawing but it is built on obsolete rendering technology (GDI+) and will be replaced.

2.2 Information Problems

The LLDP agent transmits and receives LLDP packets, which are also called protocol data units(PDUs). It can operate in transmit only mode, receive only mode or both transmit and receive mode. Each device in the network has its own LLDP agent. If the LLDP agent is in receive only mode, the device does not send LLDP-MIB. This results in the fact that topology can be virtually divided into several disconnected subgraphs because it is not possible to find all the connections between the devices. Some devices can be even completely disconnected, even if they are detected with DCP.

In the PROFINET networks it is possible to use generic Ethernet switches. These switches often do not support LLDP. This results in fact that LLDP packets will be forwarded on all ports. This in turn causes the devices to think that each one of them is connected directly to all the other devices connected to the switch. These devices will have more than one connection to one port in visualization. To avoid this problem, a generic

²Media Access Control address

³Identification and maintenance data

Ethernet switch between the devices in the network is reconstructed using information from devices which are directly connected to this switch. This feature was implemented in PRONETA especially for the purposes of the topology drawing.

The information about the communication protocols were taken from [19]. PRONETA main window is shown in Figure 2.1.

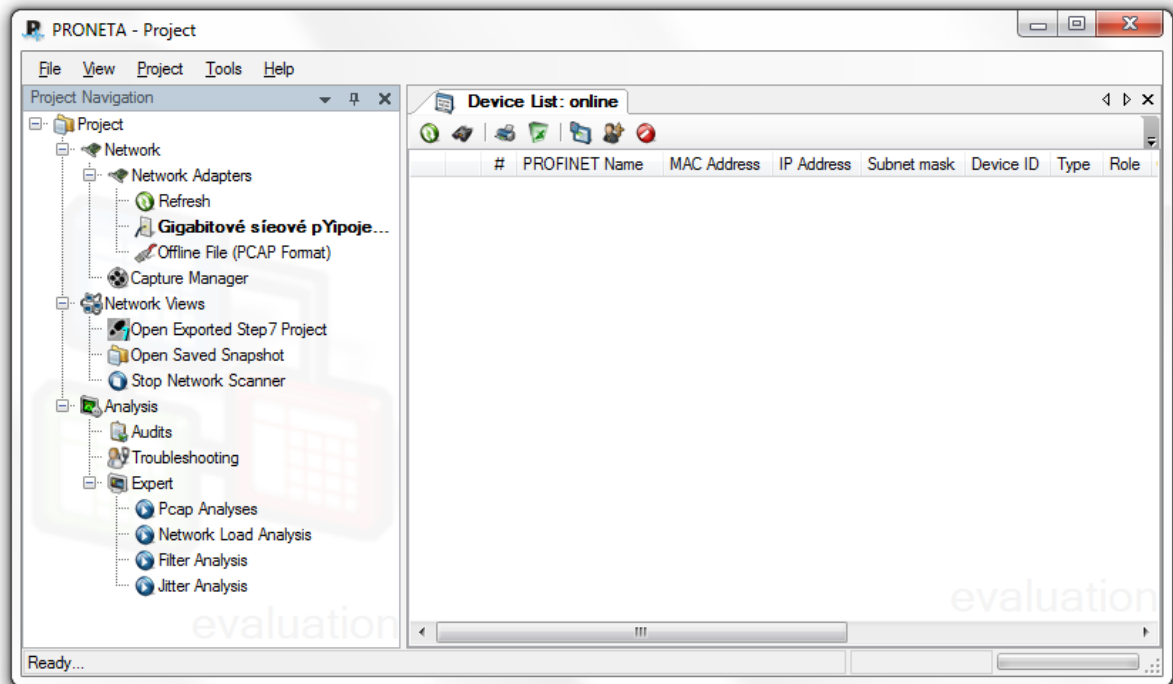


Figure 2.1: PRONETA

Chapter 3

Algorithms and Frameworks

The available algorithms and frameworks for graph visualization are described in this chapter.

3.1 Algorithms

In this section we introduce several available layout algorithms which should be considered for the drawing of the network topology. The information about these algorithms were taken from [1].

3.1.1 Hierarchical Drawings

This algorithm is often called *Sugiyama* after one of its inventors. It arranges a graph by placing vertices in different layers in such a way that most visual edges flow in the same direction and the number of their intersections is minimized. It is often chosen for graphs with the flow of information in one direction. This is not exactly the case of network topology. Nevertheless, graphs arranged by this hierarchical layout show the hierarchy very clearly and because of that, this algorithm should be considered. This hierarchical approach consists of four main steps:

- *Cycle Removal* - Temporarily reverses the direction of the edges to make the directed graph acyclic. This preprocessing step is needed if the input graph contains cycles.

- *Layer Assignment* - Assigns vertices to horizontal layers and determines their y-coordinate.
- *Crossing Reduction* - Receives properly layered graph as the input and produces newly layered graph with order specified for each vertex in each layer. It orders the vertices within each layer in such a way that the number of edge crossings is reduced.
- *Horizontal Coordinate Assignment* - Determines an x-coordinate for each vertex.

3.1.2 Rooted Tree

A rooted tree is a connected acyclic graph. It consists of a tree and a distinguished vertex which is the root. It is often used for graphs with no cycles. Vertices are not allowed to have more than one parent. The problem is that the network topology can contain rings and the graph of such topology has at least one cycle. We can use this algorithm in a way that the cycles are disconnected during the calculation. In our case we can consider network topology as a ordered tree which consists of a rooted tree and, for each vertex, an ordering of its children. These children are ordered according to the connection port number of their edges to the parent. The effect of this ordering in topology drawing will be shown later.

The described method for the construction of drawings uses the terms *depth* and *height*. The depth of a vertex of a tree is the number of edges of the path of the tree between the vertex and the root. The height of the tree is the maximum depth of a vertex of the tree.

The following method was considered for the construction of drawings of network topology in a form of a tree. *Layering* method determines horizontal layer of each vertex in a way that the vertex v of depth i has a y-coordinate $y(v) = -i$. The layered drawing is strictly downward. Avoiding crossings in a layered drawing is reached by ensuring that two vertices on the same layer have the same left-to-right relative order as their parents. Once the vertical layers are determined, the algorithm has to compute only x-coordinates. The requirement is that the parent will be placed within the horizontal span of its children and possibly in a central position. This method is possible to use for trees with more than two children for each vertex and can build trees from the left to the right instead of original vertical orientation.

3.2 Frameworks

In this section we introduce several frameworks which were considered as a basis for the developed component. All the mentioned frameworks were developed for graph visualization. Each figure in this chapter was made without any additional changes to the frameworks. They were used in their original state. The figures shows graph visualization setup which was the closest to fulfilling the requirements.

3.2.1 Requirements

The framework used for graph visualization must fulfill the following requirements:

- It has to be allowed to use the framework for commercial purposes because PRONETA is a commercial product.
- It should be written in C# and it should directly support WPF.
- It should include orthogonal routing algorithm or provide a source code for its additional implementation.
- It should be possible to directly manipulate each vertex from GUI, one at a time.
- Edge routes should be recalculated during vertex manipulation.

3.2.2 Considered Frameworks

3.2.2.1 MSAGL - Microsoft Automatic Graph Layout

MSAGL is a .NET tool for graph visualization which was developed by Lev Nachmanson at Microsoft Research. It produces layered or hierarchical layouts because it is based on the Sugiyama layout algorithm. Orthogonal routing algorithm is not implemented yet. Direct manipulation of vertex position is possible and edge routes are recalculated during this process. This framework is written in C# but it doesn't directly support WPF. Instead of WPF, it is based on GDI+. An example of graph visualization is presented in Figure 3.1. For additional information see [2].

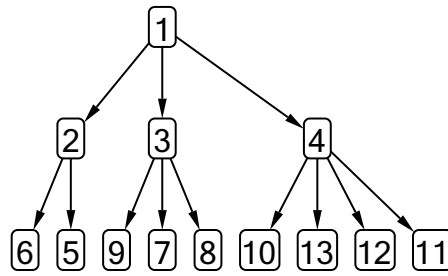


Figure 3.1: Graph visualization using MSAGL

It is also a commercial product which can be purchased at a Microsoft Store for \$280 and according to its license, it can be used for commercial purposes. It is distributed in a combined binary and source code state. The price of this product wasn't a problem because this framework was already available in ANF DATA spol. s r.o.. The former version of MSAGL is Graph Layout Execution Engine (GLEE). It is free, but it cannot be used for commercial purposes.

3.2.2.2 yFiles

These diagramming components are developed by yWorks. At first it was available only for Java. Nowadays, it is also available for .Net and WPF and for web technologies such as FLEX, AJAX and Microsoft Silverlight. It provides a very powerful solution for graph layout and orthogonal routing visualization. According to their references, direct manipulation of vertex position is not implemented. An example of graph visualization is shown in Figure 3.2.

The framework is distributed in a binary state for commercial purposes. A license for a single developer costs \$7.200. The distribution in a binary state and its high price automatically removed this choice from a list of usable frameworks. From its demo projects, we were not able to decide whether it is the best solution or not. For additional information see [23].

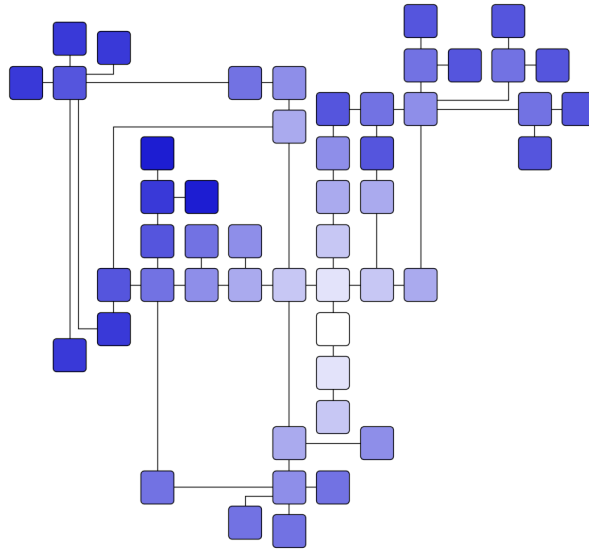


Figure 3.2: Graph visualization using yFiles

3.2.2.3 Graphviz

Graphviz was developed by AT&T¹ Labs Research for drawing graphs specified in DOT² language scripts. It is an open source graph visualization software. It takes descriptions of graphs in a simple text language and makes graphs in formats such as images or SVG for web pages. It also supports vector formats such as PDF or Postscript. This framework visualizes only static graphs. The support of direct manipulation of vertex position is not available. It has to be implemented in the host application. Using graphs in a form of simple text language isn't a preferable solution, because every graph vertex must contain an object which represents a device. Passing an object through simple text language is not possible. It is written in C and C++, which is also not optimal. An example of graph visualization is presented in Figure 3.3. It is distributed under Eclipse Public License in a binary and a source code state. Its license allows changes to the source code and the application using this framework can be used for commercial purposes. For additional information see [4].

¹American Telephone and Telegraph Company in [4]

²DOT is a plain text graph description language in [3].

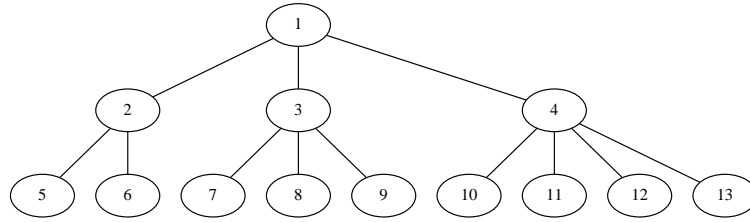


Figure 3.3: Graph visualization using Graphviz

3.2.2.4 Quickgraph

Quickgraph is a .NET library which provides generic directed or undirected graph data structures. It comes with searching algorithms such as depth first search, breath first search or A* search. It can be used as basis for the application for graph visualization. It already officially supports the mentioned frameworks (MSAGL, its former version GLEE, and Graphviz) to render the graphs.

This library is mentioned, because it is used also as a basis for Graph#. It does not fulfill the requirements enough, to be considered as a standalone basis for automatic graph visualization. For additional information see [5].

3.2.2.5 Graph#

Graph# is a graph layout framework based on Quickgraph. It uses its data structures and provides additional functionality. It is written in C# and directly supports WPF. The availability of user interaction with vertices is provided by the WPF Extensions library. The routing of edges is recalculated during direct manipulation with vertices. Orthogonal routing algorithm is not present, but it can be implemented within the structure of the framework. An example of graph visualization is shown in Figure 3.4.

This framework is distributed under Microsoft Public License (Ms-PL) in a binary and a source code form. Its license allows changes to the source code and the application using this framework can be used for commercial purposes. For additional information see [6].

3.2.3 Chosen Framework

We have explored all the mentioned frameworks except yFiles because of its high price. Only Graph# fulfills all the requirements which are necessary.

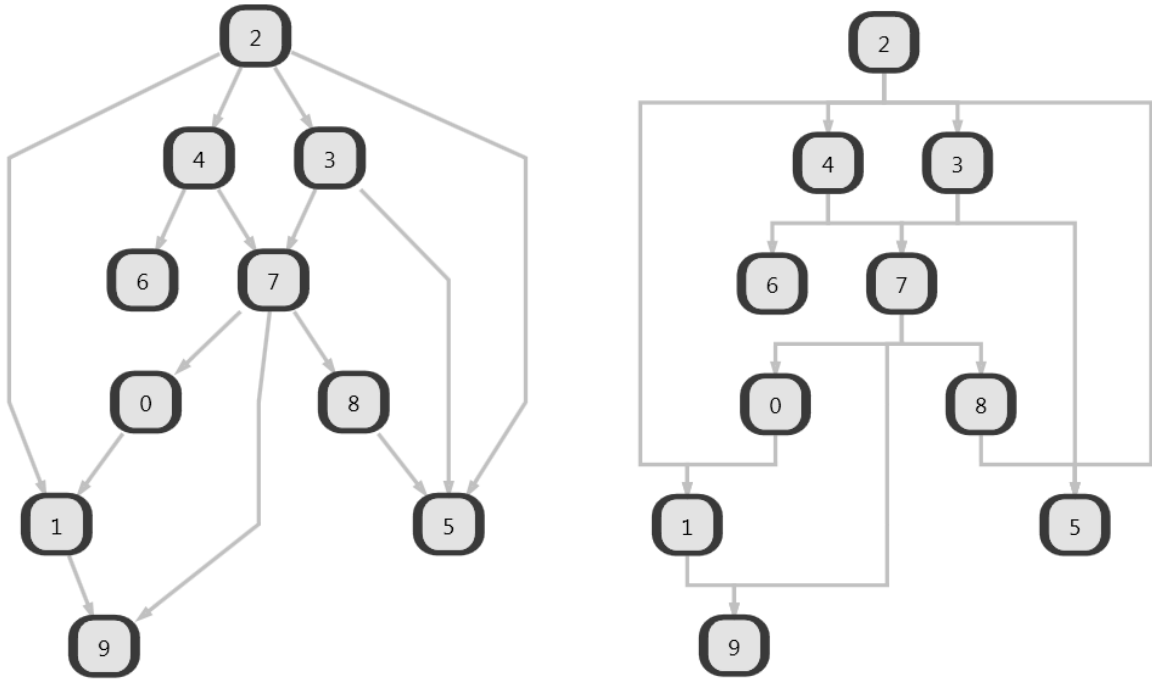


Figure 3.4: Graph visualization using Graph# and Sugiyama layout algorithm

This framework contains the following libraries:

- Base of framework (GraphSharp.dll) - It contains algorithms for layout calculation, routing edges, overlap removal and highlighting vertices and edges. It also implements modified graph data structures based on the *Quickgraph* library.
- Graph visualization (GraphSharp.Controls.dll) - The most important class in this library is *GraphLayout*. It contains methods for initialization of all available algorithms. Other important classes are *VertexControl* and *EdgeControl* which represent vertices and edges in graph visualization. All mentioned classes can be used directly in WPF.

The internal functionality of this framework will be presented in chapter 6.

Chapter 4

Topology Drawing

The core algorithms used for topology drawing are described in this chapter. We implement these algorithms within the Graph \sharp structure. The core algorithms are layout algorithm and routing algorithm. The edge routing algorithm is usually a part of the layout algorithm. In the Graph \sharp structure, these two algorithms are separated.

4.1 Layout Algorithm

The layout algorithm determines the positions of vertices in visualization. Vertices have to be organized in a way to provide the best possible readability of the final layout. After some exploration of available algorithms, we consider to use of the *Sugiyama* and the *Rooted Tree* layout algorithm. These two types of layout algorithms are included in the Graph \sharp . Unfortunately, the readability of the layout calculated with these algorithms wasn't very satisfying because of the requirement of the horizontal orientation of the network visualization and the positions of the connection ports at the bottom of the vertices. Instead of changing the available algorithms, we decided to develop a new layout algorithm and integrate it into the Graph \sharp algorithm structure. The developed layout algorithm is a combination of both approaches. It is built from a root vertex as in the tree layout algorithm and vertices are placed in horizontal layers in a specific way to avoid edge crossings as in the *Sugiyama* layout algorithm. This approach is based on the user experience with PROFINET networks visualization.

In chapter 2 we mentioned that topology can be divided into several subgraphs and some vertices can be actually completely disconnected from others. These subgraphs and

disconnected vertices have to be very well arranged to form a compact layout. It means that it should cover the smallest possible area and keep high readability.

4.1.1 Solution

Vertices and edges in Graph \sharp are generic objects. Both of them have to be implemented in a host application. Because of this, an internal graph with new object of vertices and edges has to be created to provide additional functionality used inside the algorithm. The solution for a layout algorithm consists of three main tasks. The first task is the selection of a root vertex. The second task is the tree layout algorithm which calculates the positions of vertices with at least one edge in each subtree. The third task is the placement of disconnected vertices. The whole layout is calculated using only one pass through the graph. The developed algorithm is fast enough to implement optimization process described in 4.1.2.

4.1.1.1 Graph Initialization

One of the inputs of the layout algorithm is a graph which represents the topology. This graph contains vertices connected with edges which have an assigned direction. An object, which contains the original vertex, its current size and position in visualization, is created for each original vertex. Two objects are created for each original edge. One with the same source and target vertex and the second with a reversed source and target vertex. Each edge has two assigned connection ports, one for the source vertex and one for the target vertex. Each connection port has assigned a number. Then, each edge is added into the list of edges of its source vertex. This list of edges is ordered according to the number of connection port connected to source vertex in ascending order. Ordering of this list decreases the number of edge crossings, because the edges are explored from the last. It means that the target vertex of the last edge is placed first. Effect of proper edge ordering is shown in Figure 4.1.

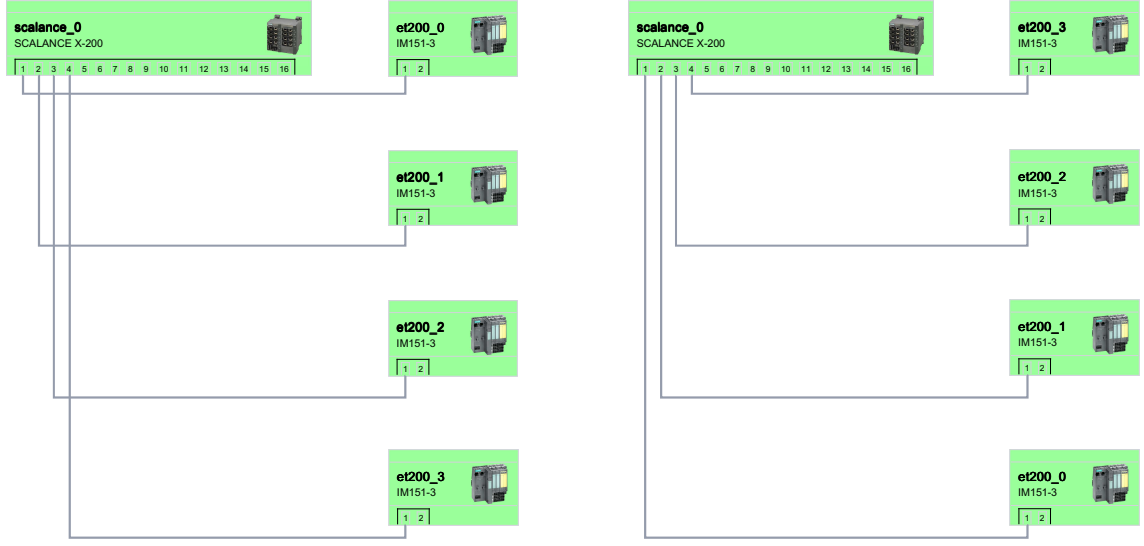


Figure 4.1: Removed unnecessary edge crossings

4.1.1.2 Root Vertex

The tree layout algorithm starts building a subtree from the root vertex. The proper selection of this root vertex is very important. It can minimize the number of edge crossings and can also reduce the size of the whole layout. The selection of the root vertex is based on an observation that the best choice is the vertex with the maximum degree. It means that the root vertex is connected to other vertices with maximum number of edges. In PROFINET networks it is often the most occupied switch.

One of the vertices in network topology visualization represents a computer which is scanning the network using PRONETA. This vertex is one of the most interesting vertices because it is probably connected to a well-known position in topology. It can be the preferred starting point for network analysis. To meet this requirement, the root vertex can be also selected by the user directly in GUI. The selected vertex is then the first root vertex in the layout algorithm, even if it has only one edge.

At the beginning of the algorithm the list of potential root vertices is compiled. Each vertex in this list has at least one edge. These potential root vertices are ordered according to the degree in descending order. Root vertex is always the first vertex in this ordered list. Only root vertex selected by the user in visualization has a higher priority. Pseudocode of method *FindNextRoot* is presented in Algorithm 1.

Algorithm 1 Finds new root vertex

```

1: function FINDNEXTROOT
2:   if root selected by user exists and is not placed then
3:      $root =$  root vertex selected by user
4:     remove  $root$  from  $potentialRoots$ 
5:     return  $root$  was found
6:   end if
7:   if  $potentialRoots$  is not empty then
8:      $root =$  first vertex from  $potentialRoots$ 
9:     remove  $root$  from  $potentialRoots$ 
10:    return  $root$  was found
11:  end if
12:  return  $root$  wasn't found
13: end function

```

4.1.1.3 Tree Layout

The the first version of this algorithm was using the method similar to the *Layering* method for constructing a drawing of rooted tree described in 3.1.2. This was based on determining the horizontal and vertical layer for each vertex and the tree was build from the left to the right. It results in a layout where the vertices are placed in a grid. The exploration of edges and vertices was the same as in the final version of a tree layout algorithm and it will be described later in this chapter. In Figure 4.2 we can see that the appearance of the layout using this first approach, was not very satisfying. Horizontal gaps between vertices were dependent on the greatest vertex width in the previous vertical layer.

The final approach is based on the observation that it is necessary to consider only the positions of the parent vertex and the last-placed vertex if we want to properly draw network topology. These two vertices determine the position of the current vertex. The current vertex is the vertex which position has to be calculated. The parent vertex was already placed and it is directly connected to the current vertex. The last-placed vertex was placed before the current vertex. The horizontal layer is also calculated because it is used for the placement of disconnected vertices.

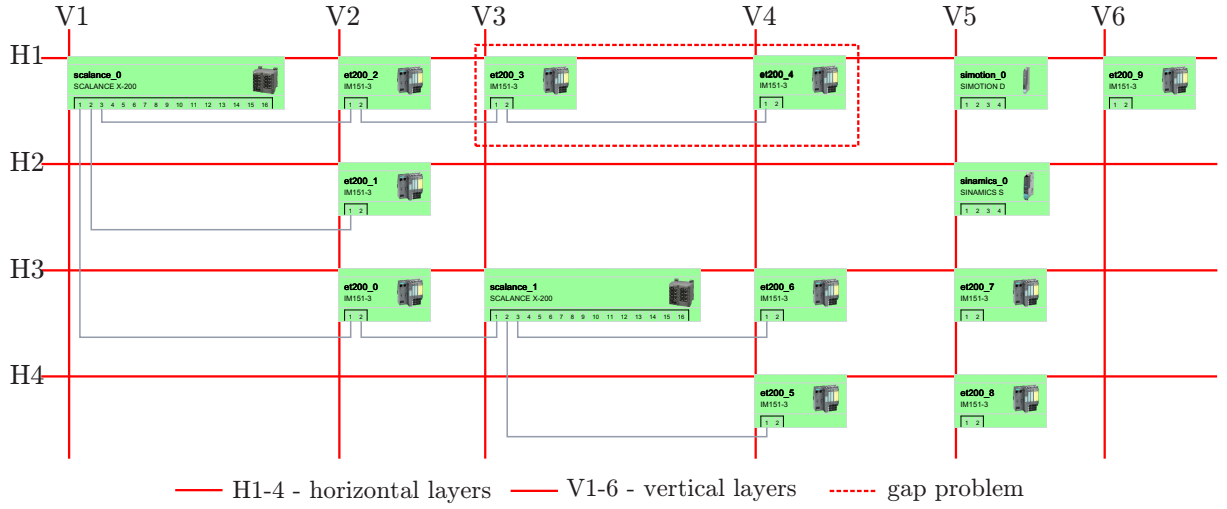


Figure 4.2: Vertices placed in a grid

If the root vertex is found, its position is calculated. It is marked as the last-placed vertex and edges are added into the list of explored edges. The list of explored edges is used as LIFO structure because the edge which was stored last is selected as first. Then, the last edge is selected and removed from this list. Its source vertex is marked as a parent vertex and target vertex as a current vertex. The horizontal coordinate X and the vertical coordinate Y of the current vertex position are calculated by using method, *CalculatePosition*, which is presented in Algorithm 2.

If the last-placed vertex is a parent vertex, the horizontal layer of the current vertex is the same as the horizontal layer of the parent vertex. Otherwise, the horizontal layer of the current vertex is always equal to the horizontal layer of the last-placed vertex incremented by one. An example of the current vertex, parent vertex and the last-placed vertex is presented in Figure 4.3.

The current vertex is then removed from the list of potential root vertices. All edges from current vertex except the edge to the parent vertex and edges to the already placed vertices are added into the list of explored edges. Edges to the already placed vertices are not added because they are closing the ring in topology. This exploration of edges is repeated until the list of explored edges is empty. If the list of potential root vertices is not empty, new root vertex is found and the exploration of edges starts again. This process is repeated until the list of potential root vertices is empty. All vertices with at least one edge are then placed. The ordered sequence of the placement of vertices is presented in Figure 4.4. Pseudocode of the tree layout algorithm is the Algorithm 3.

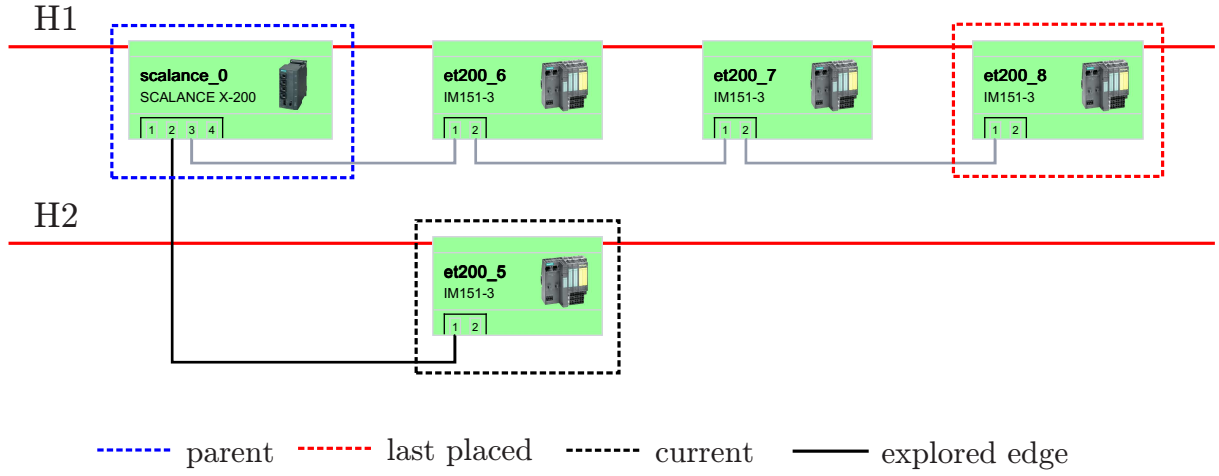


Figure 4.3: Determining the vertex position

Algorithm 2 Calculates the position of the vertex with at least one edge

```

1: function CALCULATEPOSITION(parent, lastPlaced)
2:   if parent is assigned then
3:      $X = \text{GETHORIZONTALPOSITION}(\textit{parent})$ 
4:     if parent is lastPlaced then
5:        $Y = \text{vertical position of } \textit{parent}$ 
6:     else
7:        $Y = \text{GETVERTICALPOSITION}(\textit{lastPlaced})$ 
8:     end if
9:   else
10:    set horizontal position  $X$  to 0
11:    if lastPlaced is assigned then
12:       $Y = \text{GETVERTICALPOSITION}(\textit{lastPlaced})$ 
13:    end if
14:  end if
15: end function
  
```

Algorithm 2 contains the following methods:

- **GetHorizontalPosition** calculates the horizontal position of the current vertex as a sum of the horizontal position of the *parentVertex*, the width of the *parentVertex* and the horizontal distance. The horizontal distance is the distance between vertices defined as in the parameters of a layout algorithm as shown in Figure 4.4.
- **GetVerticalPosition** calculates the vertical position of the current vertex as a sum of the vertical position of the *lastPlacedVertex*, the height of the *lastPlacedVertex* and the vertical distance. The vertical distance is the distance between vertices defined as in the parameters of a layout algorithm as shown in Figure 4.4.

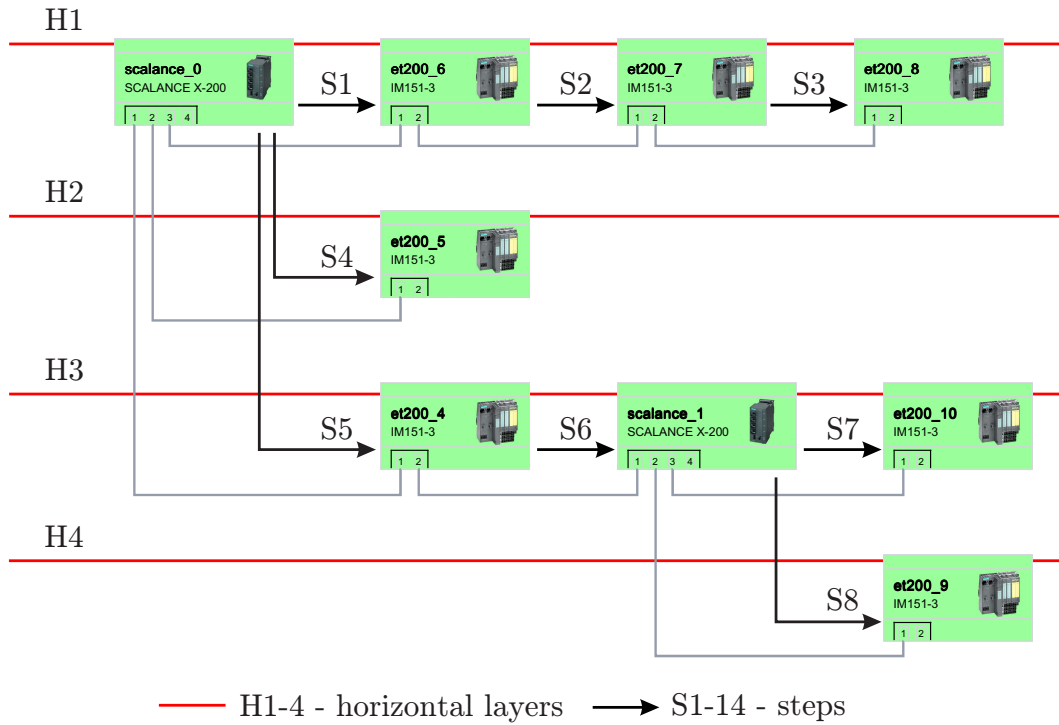


Figure 4.4: Tree layout with ordered steps

Algorithm 3 Calculates the position of the vertices with at least one edge

```

1: function CALCULATETREELAYOUT
2:   while FINDNEXTROOT( ) do
3:     PLACEVERTEX(null, rootVertex)
4:     ADDEDGES(null, rootVertex)
5:     while exploredEdgesList is not empty do
6:       currentEdge = the last edge from exploredEdgesList
7:       remove currentEdge from exploredEdgesList
8:       parent = source vertex of currentEdge
9:       current = target vertex of currentEdge
10:      if currentVertex is not placed then
11:        PLACEVERTEX(parent, current)
12:        ADDEDGES(parent, current)
13:      end if
14:    end while
15:  end while
16: end function

```

Algorithm 3 contains the following methods:

- **FindNextRoot** is described in detail in 4.1.1.2.
- **PlaceVertex**(*parent*, *current*) calculates the position of the current vertex in the visualization. It also marks the vertex which has its right border at maximal horizontal position (right border vertex). This method also determines the width and the height of the whole layout.
- **AddEdges**(*parent*, *current*) iterates through the list of edges of the current vertex and into the *exploredEdgesList* adds edges, which target vertex is not equal to the *parentVertex* and isn't placed yet.

4.1.1.4 Final Placement

This part of layout algorithm determines the positions of the vertices without edges. These disconnected vertices are placed in one or more columns next to the right border of the layout. The position of the disconnected vertex is determined by the right border vertex and the last-placed vertex. The right border vertex was found in the tree layout

algorithm and its right border in visualization has a maximal horizontal position. The maximal horizontal layer is equal to the horizontal layer of the last-placed vertex in the tree layout. The horizontal coordinate X and the vertical coordinate Y of the current vertex position are calculated by using method, *CalculateDisconnectedPosition*, which is presented in Algorithm 4.

At first, a list of vertices without edges is compiled and the current horizontal layer is set to zero. In each step the disconnected vertices are placed under each other. The current horizontal layer is incremented by one in each step until it reaches the maximal horizontal layer. The current horizontal layer is again set to zero and new right border vertex is used.

If all the vertices in topology are without edges, the maximal horizontal layer is calculated in a way that the number of vertices in the horizontal direction is equal to the number of vertices in the vertical direction. In Figure 4.6 we can see an example of this case.

Algorithm 4 Calculates position of vertex without edges

```

1: function CALCULATEDISCONNECTEDPOSITION(rightBorder, lastPlaced)
2:   if lastPlaced is assigned then
3:      $x$  =horizontal position of lastPlaced
4:      $y$  =GETVERTICALPOSITION(lastPlaced)
5:   else
6:      $x$  =GETHORIZONTALPOSITION(rightBorder)
7:     set vertical position  $y$  to 0
8:     if rightBorder is vertex in subtree then
9:       increment  $x$  by horizontal distance
10:    end if
11:  end if
12: end function

```

Algorithm 4 contains the following methods:

- **GetHorizontalPosition** and **GetVerticalPosition** were already described in 4.1.1.3.

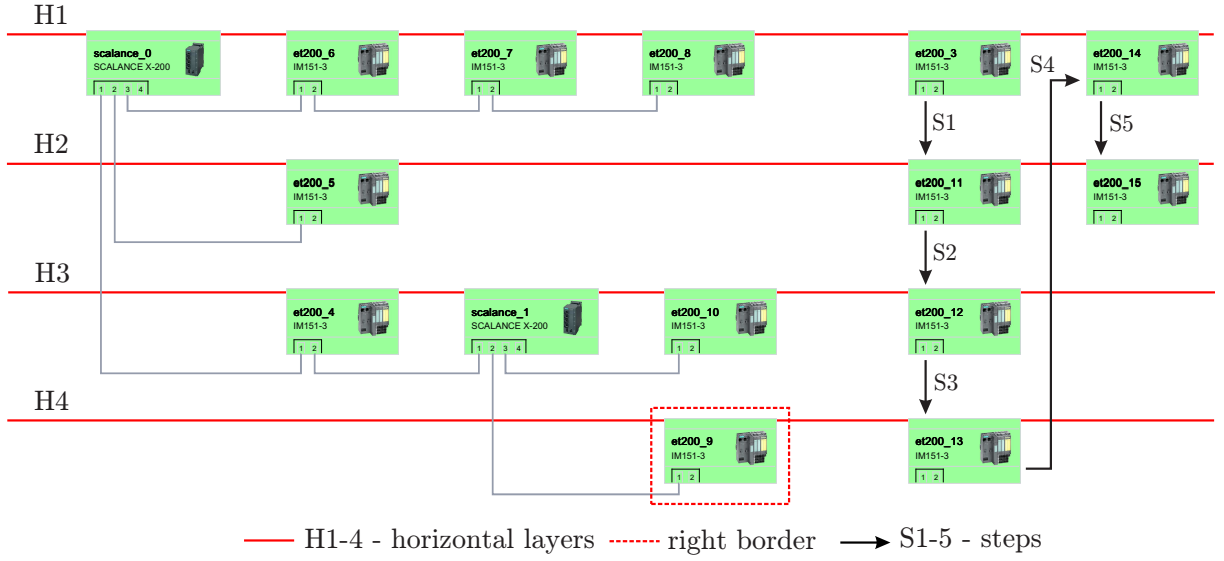


Figure 4.5: Determining the positions of disconnected vertices

Algorithm 5 Calculates positions of vertices without edges

```

1: function CALCULATEDISCONNECTEDVERTICESPOSITIONS
2:   if any subtree doesn't exist then
3:      $maxHorizontalLayer = \text{is square root of number of vertices rounded up}$ 
4:   end if
5:   for each vertex in disconnectedVerticesList do
6:      $PLACEVERTEX(oldRightBorderVertex, currentVertex)$ 
7:     increment currentHorizontalLayer by 1
8:     if currentHorizontalLayer is equal to maxHorizontalLayer then
9:       set currentHorizontalLayer to 0
10:      set oldRightBorderVertex to newRightBorderVertex
11:      set newLastPlacedVertex to null
12:    end if
13:  end for
14: end function
  
```

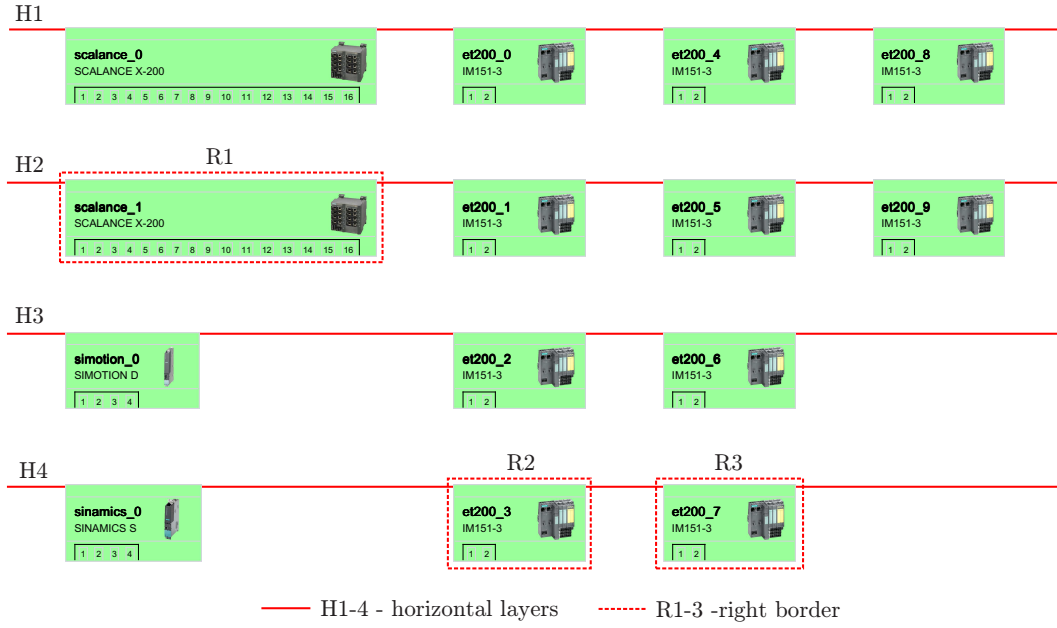


Figure 4.6: Visualization of the topology without edges

4.1.2 Optimization

In 4.1.1.2, it was stated that selection of proper root vertex can reduce the size of the whole layout. The optimization process is based on the recalculation of the layout for each potential root vertex in the current subtree. The potential root vertex has the same degree as the first root vertex in the current subtree. The root vertex is optimal if the size of the current subgraphs is the smallest possible. In Figure 4.7 and Figure 4.8 we can see that the optimized layout covers a smaller area than not optimized. Pseudocode of a modified tree layout algorithm with optimization is Algorithm 6.

Algorithm 6 Calculates optimal positions of vertices with at least one edge

```

1: function CALCULATETREELAYOUT
2:   while FINDNEXTROOT( ) do
3:     ADDBRANCHROOT(rootVertex)
4:     while FINDOPTIMALROOT( ) do
5:       PLACEVERTEX(null, rootVertex)
6:       ADDEDGES(rootVertex)
7:       while exploredEdgesList is not empty do
8:         currentEdge = the last edge from exploredEdgesList
9:         remove currentEdge from exploredEdgesList
10:        parentVertex = source vertex of currentEdge
11:        currentVertex = target vertex of currentEdge
12:        if currentVertex is not placed then
13:          PLACEVERTEX(parentVertex, currentVertex)
14:          ADDEDGES(parentVertex, currentVertex)
15:          ADDBRANCHROOT(currentVertex)
16:        end if
17:      end while
18:      OPTIMIZEBRANCH( )
19:    end while
20:  end while
21: end function

```

Algorithm 6 contains the following methods:

- **FindOptimalRoot** returns the next available root vertex for the current subgraph. If the tree layout was already calculated for each potential root vertex of the current subgraph, it returns the optimal root vertex. The root vertex selected by the user is returned as optimal and optimization of the current subgraph is aborted.
- **AddBranchRoot**(*currentVertex*) finds potential root vertices. If the current vertex has the same number of edges as the current root vertex, it is marked as another potential root vertex of the current subgraph.
- **OptimizeBranch** calculates the size of the current subgraph and decides whether the current root vertex is the optimal choice or not. The root vertex is optimal if

the size of the current subgraph is the smallest possible. It also resets the layout of the current subgraph for recalculation with another root vertex.

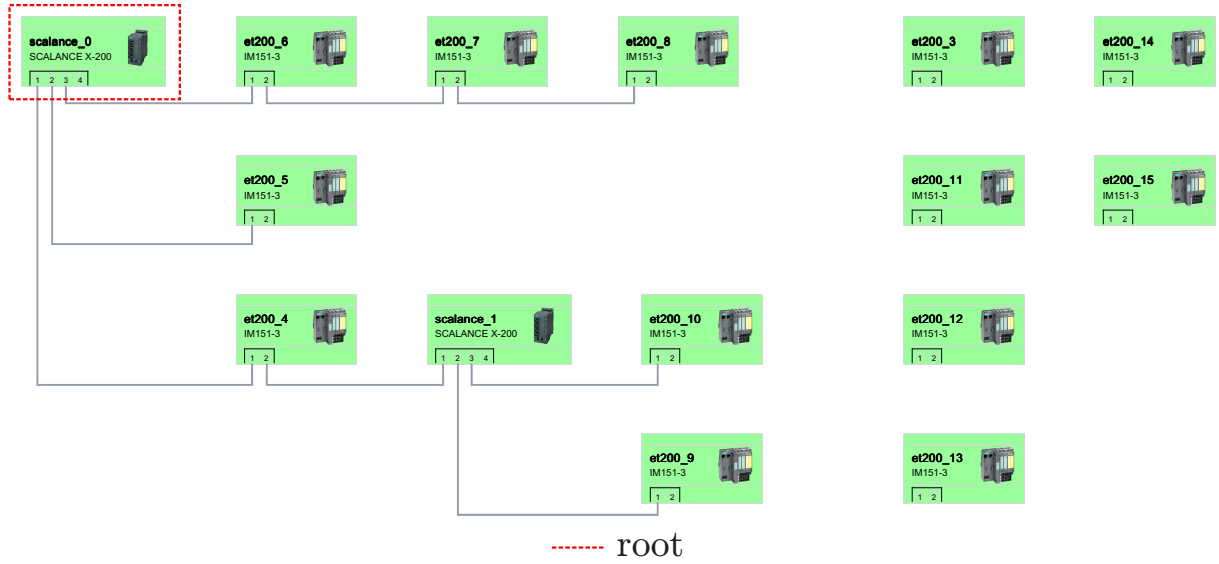


Figure 4.7: Optimized tree layout

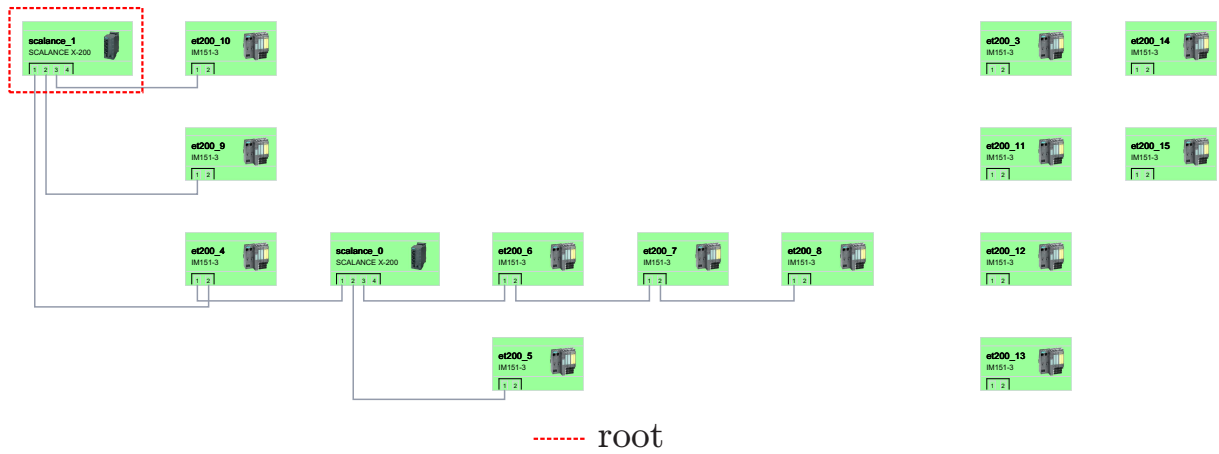


Figure 4.8: Not optimized tree layout

4.2 Routing Algorithm

Routing algorithm determines the edge routes between the vertices. This problem is often solved by using *ad-hoc* heuristics that lead to aesthetically unpleasing routes. Edges can be routed through vertices and some of them can even share their paths. We will introduce our own implementation of orthogonal edge routing algorithm based on [9] which removes the mentioned problems.

The ring is a standard type of network topology. PROFINET supports the ring topology for its high availability. If a cable or device fails, then the topology will be automatically changed to the line topology and the network remains active as stated in [24]. To provide information about this feature, this algorithm also contains a detection of the rings in the topology .

4.2.1 Solution

First of all, an internal graph of topology has to be created as in 4.1.1.1. Second of all, the rings in topology are detected. Finally, orthogonal routing algorithm is computed. Orthogonal routing algorithm consists of three steps. At first, an orthogonal visibility graph is created. Then, optimal edge routes are found using A* algorithm which searches through the orthogonal visibility graph. Optimality is reached by minimization of edge route length and the number of bends. Finally, the post-processing removes shared parts and determines actual positions of edge routes in visualization.

4.2.1.1 Ring Detection

PRONETA can detect newly connected devices and connections in a network. These devices and connections can be immediately added as vertices and edges into the topology visualization without additional layout recalculation. Because of that ring detection algorithm is integrated into the edge routing algorithm instead of the layout algorithm. Any edge added into the topology causes recalculation of edge routes and one or more rings are detected.

This part of the algorithm detects rings. It is actually a modified version of the tree layout algorithm described in 4.1.1.3. Of course, the positions of vertices are not calculated. The main difference is the method *AddEdges* itself. The ring is detected when the target vertex of the edge is already explored and it is not the parent vertex. Pseudocode of modified method *AddEdges* is presented in Algorithm 7. Rings which

were found in topology are visualized with different colour. In Figure 4.9 we can see an example of a detected ring.

Algorithm 7 Explores edges and marks edges in the ring

```

1: function ADDEDGES(parentVertex, currentVertex)
2:   for each edge from currentVertex do
3:     if target vertex of edge is not parentVertex then
4:       if target vertex of edge is not explored then
5:         add edge into edgeList
6:         edgeToVertex of target vertex of edge is edge
7:         target vertex of edge is now explored
8:       else
9:         currentEdge = edge
10:        while currentEdge is not empty and is not in the ring do
11:          mark currentEdge as edge in the ring
12:          if source of currentEdge is target of edge then
13:            break while cycle
14:          end if
15:          currentEdge = edgeToVertex of source vertex of currentEdge;
16:        end while
17:      end if
18:    end for
19:  end function

```

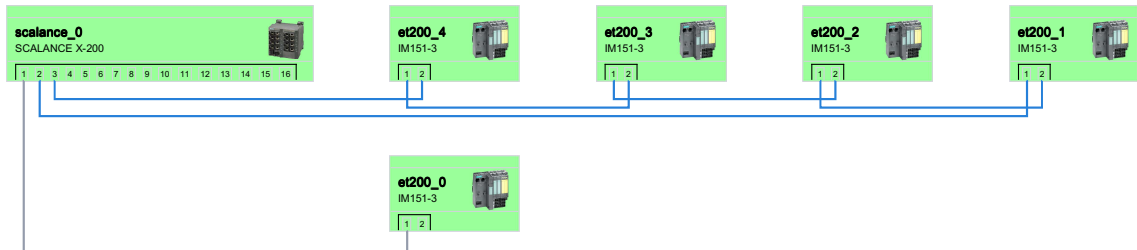


Figure 4.9: Ring detected in the topology

4.2.1.2 Orthogonal Visibility Graph

The orthogonal visibility graph is determined by horizontal and vertical lines of visibility from the corners of bounding box of each vertex and their connection ports. This orthogonal edge routing algorithm is based on an observation that it is necessary to consider only routes in the orthogonal visibility graph. This visibility graph consists of nodes. Each node is orthogonally connected to its nearest neighbours in a way that there is no intervening vertex between them. The node represents a point through which an edge can be routed.

First of all, the important nodes need to be created. Important nodes are nodes at positions of corners of bounding box of each vertex and nodes at positions of connection ports. An example of created important nodes is presented in Figure 4.10. We can see that vertices are not in the middle of their bounding boxes. It is caused by fact that distances between vertices are constant but the width of the vertices can differ. The nodes are created at positions defined by a grid. This approach is used to minimize quantity of nodes. It is obvious that pathfinding is faster when visibility graph contains less nodes to search in.

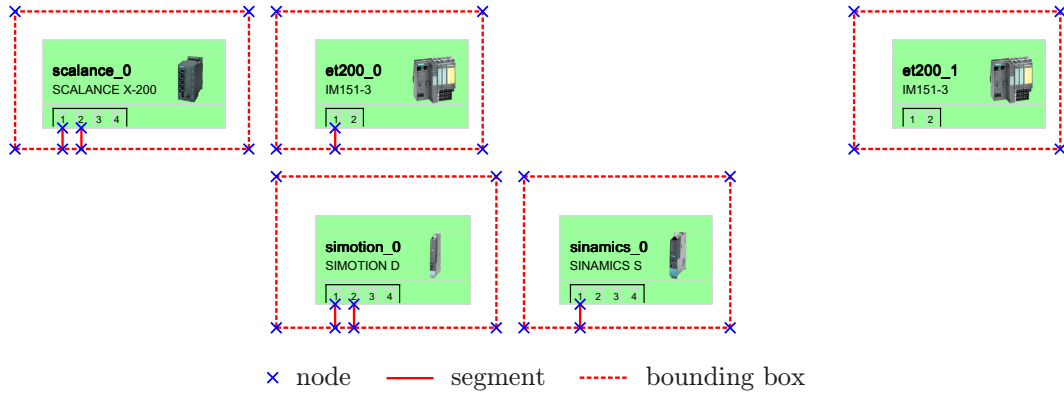


Figure 4.10: Important nodes in a visibility graph

Second of all, important horizontal segments are generated. The modified line sweep algorithm described in [10,11] is used for this task. This algorithm uses a vertical sweep through the current nodes in the visibility graph, keeping a horizontal scan line list of open nodes. The vertical scan starts all top-left and top-right nodes and stops all bottom-left and bottom-right nodes of bounding boxes of the vertices. This step ensures that the horizontal segments will not overlap the vertices. Left and right nearest neighbours are found or created for each node. Each node then has references to its

nearest left and right neighbours. These nodes can be corner nodes of surrounding vertices, nodes created on border of bounding boxes of surrounding vertices or nodes created on border position of the whole layout. References to neighbours represents horizontal segments in the visibility graph. An example of created horizontal segments is presented in Figure 4.11. Pseudocode of *VerticalScan* method is presented in Algorithm 8 where Y represents vertical coordinate and X represents horizontal coordinate of a node. Data structure *nodesYTree* contains all nodes available after creation of important nodes ordered according to vertical and horizontal position. The scan is performed from the top to the bottom and from the left to the right.

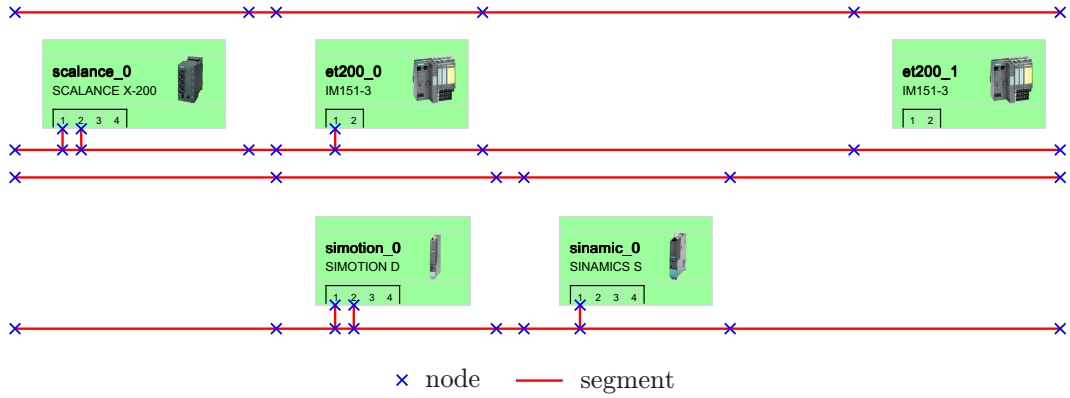


Figure 4.11: Horizontal segments and nodes after vertical scan

Finally, vertical segments are generated. This uses a horizontal sweep through the current nodes in the visibility graph, keeping a vertical scan line list of open nodes and list of open horizontal segments. The horizontal scan starts all top-left and bottom-right nodes and stops all top-right and bottom-right nodes of bounding boxes of the vertices. This step ensures that the horizontal segments will not overlap the vertices. This task is connected with a calculation of positions of intersections between horizontal and vertical segments. New nodes are created at positions of these intersections. References to all available neighbours are assigned to each node during this process. An example of a visibility graph is presented in Figure 4.12. Pseudocode of *HorizontalScan* method is presented in Algorithm 9. Data structure *nodesXTree* contains all nodes available after creation of important nodes and horizontal segments ordered according to horizontal and vertical position. The scan is performed from the left to the right and from the top to the bottom.

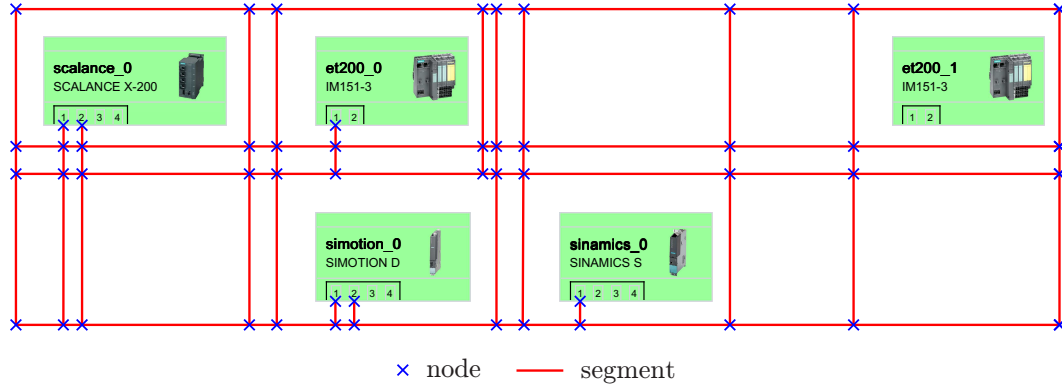


Figure 4.12: Final visibility graph

Algorithm 8 Vertical scan which creates the horizontal segments

```

1: function VERTICALSCAN
2:   for each  $Y$  in  $nodesYTree$  do
3:      $newNodes$  = all nodes with vertical position equal to  $Y$  in  $nodesYTree$ 
4:     for each  $node$  in  $newNodes$  do
5:       if  $node$  starts vertical scan then
6:         add  $node$  into  $openNodes$ 
7:       end if
8:     end for
9:     for each  $node$  in  $newNodes$  do
10:       $maxLeftX$  = get closest smaller  $X$  from  $openNodes$ 
11:       $minRightX$  = get closest larger  $X$  from  $openNodes$ 
12:       $nodeLeft$  =  $ADDNODE(maxLeftX, Y)$ 
13:       $nodeRight$  =  $ADDNODE(minRightX, Y)$ 
14:      set references to horizontal( $nodeLeft$  and  $nodeRight$ ) neighbours of  $node$ 
15:    end for
16:    for each  $node$  in  $newNodes$  do
17:      if  $node$  stops vertical scan then
18:        remove  $node$  from  $openNodes$ 
19:      end if
20:    end for
21:  end for
22: end function
  
```

Algorithm 6 contains the following methods:

- **AddNode**(X, Y) finds or creates node at position defined in parameters.

Algorithm 9 Horizontal scan which creates the vertical segments and their intersections

```

1: function HORIZONTALSCAN
2:   for each  $X$  of  $nodesXTree$  do
3:      $newNodes$  = all nodes with horizontal position equal to  $X$  in  $nodesXTree$ 
4:     for each  $node$  in  $newNodes$  do
5:       if  $node$  starts horizontal scan then
6:         add  $node$  into  $openNodes$ 
7:       end if
8:       if  $node$  starts horizontal segment then
9:         add  $node$  into  $openSegmentNodes$ 
10:      end if
11:    end for
12:    for each  $node$  in  $newNodes$  do
13:       $maxTopY$  = closest smaller  $Y$  from  $openNodes$ 
14:       $minBottomY$  = closest larger  $Y$  from  $openNodes$ 
15:       $CREATE\_NODE\_CONNECTIONS(X, maxTopY, minBottomY)$ 
16:    end for
17:    for each  $node$  in  $newNodes$  do
18:      if  $node$  stops vertical scan then
19:        remove  $node$  from  $openNodes$ 
20:      end if
21:      if  $node$  stops horizontal segment then
22:        remove  $node$  into  $openSegmentNodes$ 
23:      end if
24:    end for
25:  end for
26: end function

```

Algorithm 9 contains the following methods:

- **CreateNodeConnections**($X, maxTopY, minBottomY$) finds or creates node at position of intersections of segments and is presented in Algorithm 10.

Algorithm 10 Creates the node connections and the intersections of the segments

```

1: function CREATENODECONNECTIONS(currentX, maxTopY, minBottomY)
2:   set openSegmNodes enumeration start to maxTopY and end to minBottomY
3:   for each segmentNode of openSegmNodes do
4:     if it is the first iteration then
5:       nodeTop = GETCURRENTNODE(currentX, maxTopY, openSegmNodes)
6:     end if
7:     lastTopY = Y position of segmentNode
8:     nodeBottom = GetCurrentNode(currentX, lastTopY, openSegmNodes)
9:     set vertical neighbours of segmentNode to nodeTop and nodeBottom
10:    nodeTop = nodeBottom
11:  end for
12: end function

```

Algorithm 10 contains the following methods:

- **GetCurrentNode**(*currentX*, *maxTopY*, *minBottomY*) finds or creates node at position of an intersection of the segments.

4.2.1.3 Pathfinding

Pathfinding is performed by A* algorithm. It iteratively builds partial paths that start from the source node until the target node is reached. Partial paths are nodes in a visibility graph with an assigned parent node (except the source node) which are stored in a open list. The parent node is a predecessor of the partial path. First of all, the source node and the target node are found. These nodes were created at positions of connector ports of source and target vertices and they were assigned to the corresponding edges. Then, the source node is added into the open list. At each step the partial path with the lowest cost is taken from the open list, added to a closed list and expanded. The neighbours of expanded node are placed in the open list. The neighbour node is only added if it was not already removed from the open list. If a neighbour node is already in the open list, and its new cost is lower, its parent node and cost are replaced. The process stops when the target node is in the closed list. The path is then reconstructed from the target node to the source node. The cost associated with each partial path is the length of path so far plus the lower bound estimation of length of path to the target node. The lower bound estimation is the Euclidean distance of the partial path and the target node.

If the position of the neighbour node is not in a horizontal or a vertical line with its two predecessors, the bend of edge is detected. A defined penalty is then added to the cost of neighbour node.

Algorithm 11 Finds paths for each edge through visibility graph

```

1: function PATHFINDING
2:   for each edge in graph do
3:     clear openList
4:     sourceNode = node at position of connection port to source vertex of edge
5:     targetNode = node at position of connection port to target vertex of edge
6:     add sourceNode into the openList
7:     while openList is not empty and targetNode is not in the closedList do
8:       currentNode is a node with the lowest cost from openList
9:       remove currentNode from openList
10:      add currentNode into closedList
11:      if targetNode is not in the closedList then
12:        EXPANDNODE(currentNode)
13:      end if
14:      if targetNode is in the closedList then
15:        CREATEEDGESEGMENTS(edge, endNode)
16:      end if
17:    end while
18:    path of edge is reconstructed path
19:  end for
20: end function

```

Algorithm 11 contains the following methods:

- **ExpandNode** explores all the neighbours of the current node. If the neighbour is not in the *closedList*, it is added into the *openList*. If a neighbour is already in the *openList* and its new cost is lower, its parent node and cost are replaced.
- **CreateEdgeSegments** reconstructs path from the target node to the source node. It also creates edge segments which represents horizontal and vertical segments of current path. These segments are used for post-processing which determines actual edge routes in visualization.

4.2.1.4 Post-Processing

This algorithm determines actual positions of edge routes in visualization. Each edge is divided into segments during reconstruction of path in 4.2.1.3. Each segment represents a horizontal or a vertical part of edge. Segments are ordered according to two criteria. The first criterium is a segment type. The segment type is determined by the input and output direction of the edge route to this segment. This ordering of the segments prefers left and bottom border of the bounding box of the vertex. It can be done in reversed direction. All segment types and their order are presented in Figure 4.13. The order of segment types determines priority of a segment in post-processing. The lower number means higher priority. The second criterium is the number of nodes in each segment. The segment with the lowest number of nodes and the highest priority of a segment type is placed first.

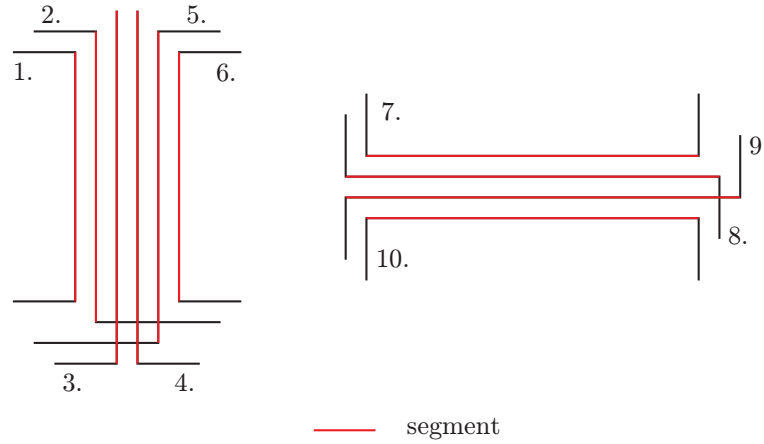


Figure 4.13: Order of the edge segment types

Each segment contains a list of segments which share at least one node. Before the final positions of edge points are determined, a segment level has to be calculated. Segment level represents the final order of shared edge segments. The list of segments is ordered according to this segment level in every step. The purpose of this procedure is to fill all available segment levels. Pseudocode for creation of the final route points is in Algorithm 12. The final path of each edge route is determined in a way that each segment adds to this path only one point. Segment level shifts final point in vertical or horizontal direction. The direction depends on orientation of a segment. The following changes of coordinates are necessary to keep the edge routes orthogonal. If previous segment orientation is horizontal and current segment orientation is vertical, Y coordinate is

replaced with previous segment final point Y coordinate. If previous segment orientation is vertical and current segment orientation is horizontal, X coordinate is replaced with previous segment point X coordinate. Pseudocode of method *CalculateEdgeRoutingPoints* for creation of the final route points is in Algorithm 13.

Algorithm 12 Determines level of the edge segment

```

1: function ORDEREDGESEGMENTS
2:   for each ordered currentSegment do
3:     set level of currentSegment to 0
4:     if type of currentSegment is not directly connected to connection port then
5:       currentSegments = shared segments ordered according to segment level
6:       for each segment in currentSegments do
7:         if level of currentSegment is equal to level of segment then
8:           increment level of currentSegment by 1
9:         end if
10:      end for
11:    end if
12:  end for
13: end function

```

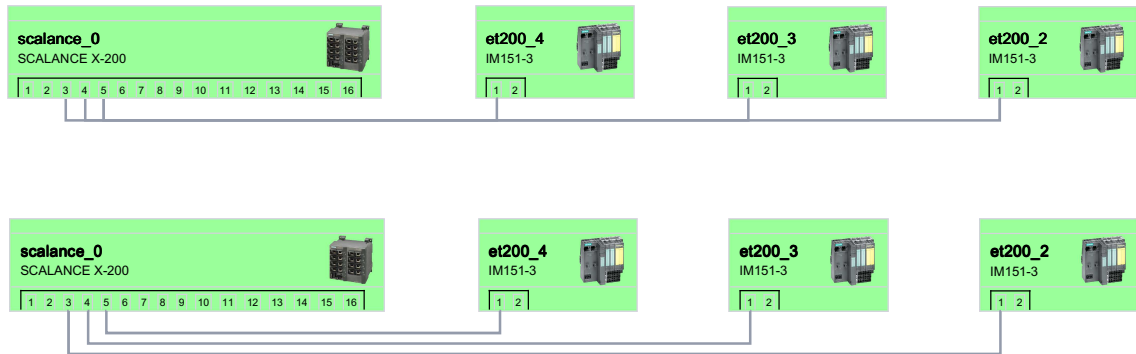


Figure 4.14: The edge routes without and with post-processing

Algorithm 13 Calculates the final position of the edge routing points

```

1: function CALCULATEEDGEROUTINGPOINTS
2:   for each edge in graph do
3:     add position of target node into path
4:     for each currentSegment of edge do
5:       originalX=horizontal position of the first node in currentSegment
6:       originalY=vertical position of the first node in currentSegment
7:       segmentLevel=level of currentSegment
8:       if segment is horizontal then
9:          $Y = originalX + segmentLevel * gap$ 
10:      else
11:         $X = originalY + segmentLevel * gap$ 
12:      end if
13:      if originalX is equal to previousOriginalX then
14:         $Y = previousY$ 
15:      end if
16:      if originalY is equal to previousOriginalY then
17:         $X = previousX$ 
18:      end if
19:      add position with X and Y coordinates to path
20:       $previousOriginalX = originalX$ 
21:       $previousOriginalY = originalY$ 
22:       $previousX = X$ 
23:       $previousY = Y$ 
24:    end for
25:    add position of source node into path
26:    add path to dictionary of edge routing points with edge as a key
27:  end for
28: end function

```

4.2.2 Optimization

In this section several optimization technics are introduced which optimize the appearance of the edges and the performance of the algorithm.

4.2.2.1 Appearance Optimization

In Figure 4.15 we can see that basic post-processing fails if the shared paths are routed around the left or the top border of the vertex. In this case, the edges can overlap the vertices. To avoid this problem it was necessary to improve the post-processing algorithm. In 4.2.1.2 the position of corner nodes of bounding box of each vertex is determined relative to this vertex as a top-left, top-right, bottom-left and bottom-right. If the segment contains at least one of these corner nodes, it is marked according to its orientation as left, right, top or bottom border segment. If the current segment is left or top border, its level is decremented in Algorithm 12. The final position of the current segment is then further from the left or the top border of the vertex.

In Figure 4.15 it is also obvious that some edges are routed differently than others, even if they have similar target position. A* algorithm described in 4.2.1.3 always finds the optimal path for the edge between the vertices. This path can differ if the pathfinding starts from the source or from the target vertex. To eliminate this behaviour, edges have to be ordered to unify edge paths. Edge routes are routed in the same order as they are added into the list of edges in 4.2.1.1.

4.2.2.2 Performance Optimization

Orthogonal visibility graph algorithm is described in 4.2.1.2. We have to find neighbours of each node. To complete this task it is advantageous to use data structure, which can find the previous and the next node in vertical or horizontal direction in the shortest time possible. We implemented data structure called AA Tree for this purpose. AA trees are named after Arne Andersson, their inventor. This data structure keeps all members ordered according to a defined parameter. In our case, this parameter is a vertical or horizontal position of node in the visibility graph. AA tree is a balanced binary tree and its feature is that each node can be find in only a few steps. We also used this data structure in other algorithms, where the objects have to be ordered, to improve performance. In Figure 4.16 and Figure 4.17 we can see visualizations of methods *Skew* and *Split*, which are used for the rebalancing of the AA tree. *Skew* is a right rotation when an insertion or deletion of an AA tree node creates a left horizontal link on the same

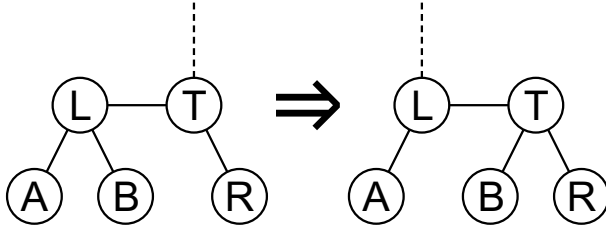


Figure 4.16: AA tree skew method from [17]

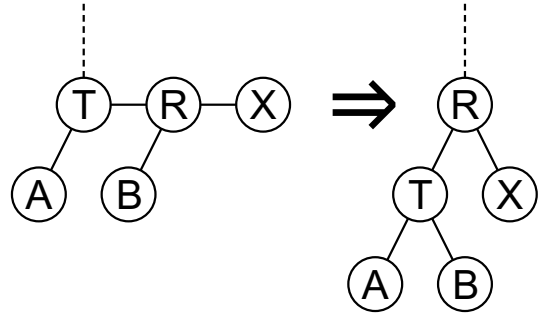


Figure 4.17: AA tree split method from [17]

level. Split is a conditional left rotation when an insertion or deletion of AA tree node creates two horizontal right links on the same level. The simple description is presented in [14,17].

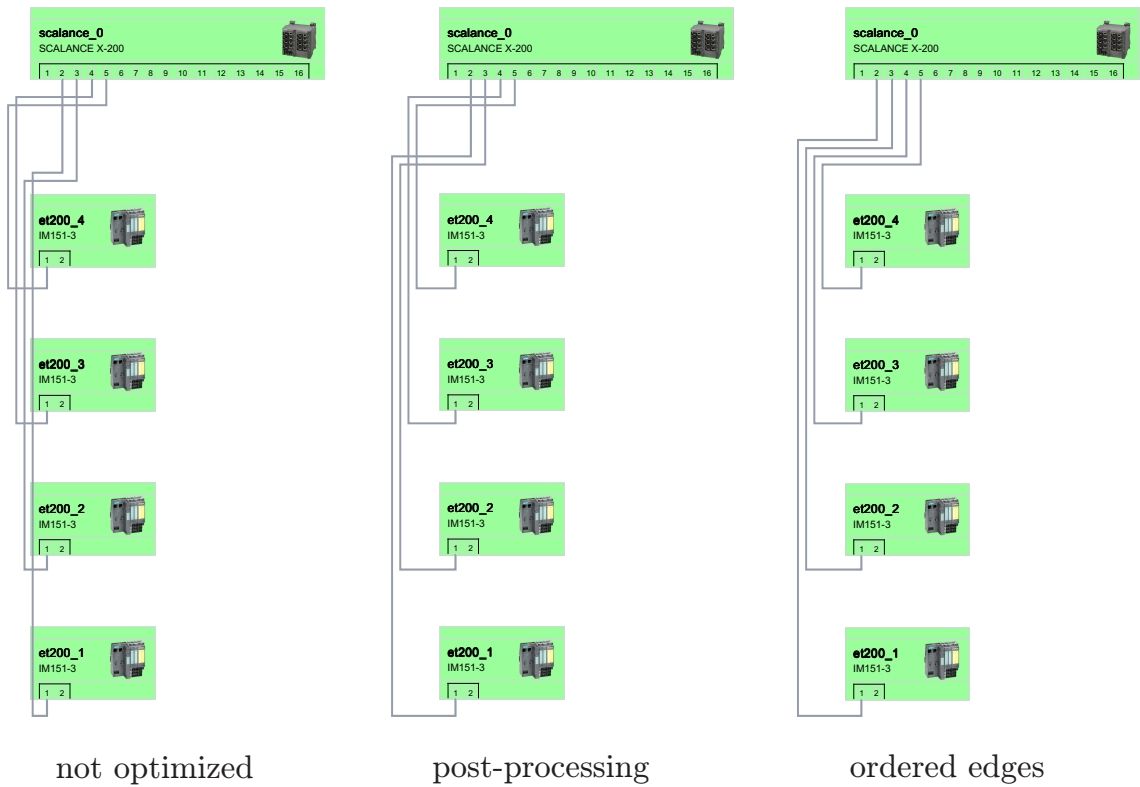


Figure 4.15: Appearance optimization

The most time consuming action in pathfinding algorithm described in 4.2.1.3 is to find the node in the open list with the lowest cost which has to be expanded next. It is not necessary to keep all nodes ordered. The data structure which can be used as a proper open list is a binary heap. The binary heap is a heap data structure created using

a binary tree. All levels of the tree except possibly the last one are fully filled. The heap nodes of the last level of the tree are filled from left to right. Each heap node cost is smaller than or equal to each of its children. The first heap node is always with the lowest cost. In our case, heap node cost is equal to the cost of a node in a visibility graph in 4.2.1.3. This approach significantly improved the performance of pathfinding because the next node for expansion is always the first and searching through all nodes in an open list is not necessary.

Large topologies consist of hundreds of vertices and edges. Also in this case, user interaction with vertices must be comfortable without any delays. The recalculation of all edge routes appeared to be too slow. To solve this problem only the edges in the subgraph of the manipulated vertex are routed, instead of routing all the edges in the topology. The subgraph consists of the manipulated vertex, directly connected vertices and edges between them. The remaining edges are not available for post-processing. When vertex manipulation stops, all edge routes are recalculated again to remove shared segments. This solution provides smooth vertex manipulation even for large topologies. The internal graph is initialized as in the 4.1.1.1. However, in this case only the vertices inside the rectangular area, defined by vertices in the subgraph, are added. In Figure 4.18 we can see a subgraph with routed edges and vertices used for the creation of a visibility graph.

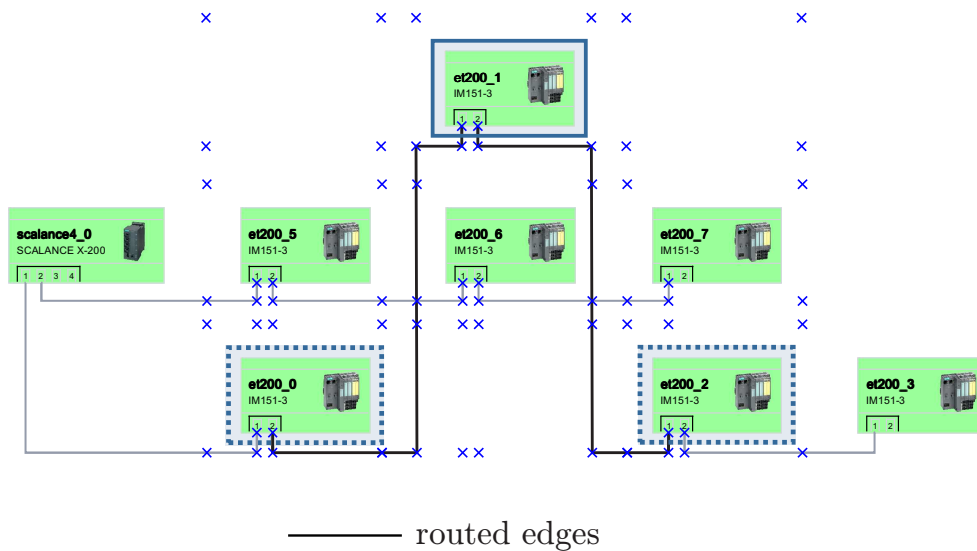


Figure 4.18: Subgraph optimization

Chapter 5

Topology Comparison

In this chapter, the methods for comparison of topologies are described. The first algorithm for topology comparison compares the current state of a network (snapshot) and the previous state of the same network. It creates a merged topology which contains vertices and edges from both. The merged topology is then displayed.

The second algorithm compares the configured topology loaded from the SIMATIC STEP 7 project and the current state of the network topology. The purpose of this algorithm is to find the best matching vertex in the network topology for each vertex in the configured topology, if it is possible.

The information about the devices in the PROFINET networks can be used for topology comparison. We can use semantic similarity components such as the device name or the MAC address to compare the previous and current state of the same network. The second possibility is to find similar subgraphs in the surroundings of the compared vertices.

5.1 Comparison Algorithm for Visualization of Changes

This comparison algorithm is partially based on [18]. The comparison of two topologies is meaningful only if they are sufficiently similar. This decision is up to the user. If the requirement of similarity is met, then the matching vertices can be found easily because each physical device has a unique MAC address.

5.1.1 Matching Vertices

Vertices are compared according to the unique name which consists of the device name and the MAC address. If the project is without any changes, the device names and the MAC addresses in the network should remain the same as in the snapshot. If the device name is not available, only the MAC address is used for the comparison. The physical device in the network might have been replaced because of its malfunction and the comparison of MAC addresses reveals this change. A device without an assigned name has to be marked because it is not in a satisfactory state. Each device in the PROFINET networks must have a unique device name.

5.1.2 Merged Topology

A merged topology is created by combining the two input topologies. First of all, the unique vertices from both topologies have to be added to the merged topology. This can be done in several steps. At first we have to create a data structure which contains lists of the vertices of the same device type for both topologies. The vertex names are then compared only between vertices with the same device type. Then, all vertices from the snapshot topology are added to the merged topology. All matching vertices are marked as existing in both topologies, and all non-matching vertices from the network topology are added to the merged topology. Finally, all vertices from the network topology, with unexplored device types, are added to the merged topology. Pseudocode of the method *CreateVertices*, which perform the tasks already mentioned, is presented in Algorithm 14.

Algorithm 14 Creates new vertices of the merged topology.

```

1: function CREATEVERTICES(networkTypes,snapshotTypes)
2:   for each snapshotTypeList in snapshotTypes do
3:     for each snapshotVertex in snapshotTypes do
4:       ADDVERTEXTODICT(snapshotVertex,snapshot only)
5:     end for
6:     if device type of snapshotTypeList exists in networkTypes then
7:       COMPAREVERTEXNAMES(snapshotTypeList,networkTypeList)
8:     end if
9:   end for
10:  for each networkTypeList in networkTypes do
11:    if device type of networkTypeList does not exist in snapshotTypes then
12:      for each networkVertex in networkTypeList do
13:        ADDVERTEXTODICT(snapshotVertex,network only)
14:      end for
15:    end if
16:  end for
17:  add created vertices to the merged topology
18: end function

```

Algorithm 14 contains the following methods:

- **AddVertexToDict**(*currentVertex*, *state*) creates a new vertex encapsulation of a device for visualization. If the vertex with the same device name and MAC address is already in the vertex dictionary, only its state is changed.
- **CompareVertexNames**(*snapshotTypeList*,*networkTypeList*) compares the names of vertices and marks the matching vertices.

If all vertices are present in the merged topology, algorithm starts the comparison of the edges. At first all the edges from the snapshot topology are added to the merged topology. A suitable key is created for each edge. This key consists of a source vertex name, source port number, target vertex name and a target port number. It also creates a second key with a reversed source and target. These two keys are used for the unique identification of each edge. The first type of key is also created for each edge from the network topology. If the edge with this key was not already used, edge from the network

topology is added to the merged topology. Pseudocode of the method *AddEdges* is presented in Algorithm 15.

Algorithm 15 Adds edges to the merged topology

```

1: function CREATEEDGES(edges)
2:   for each edge in edges do
3:     sourceVertex = sourceVertex of edge
4:     targetVertex = targetVertex of edge
5:     sourceName = GETEDGENAME(sourceVertex)
6:     targetName = GETEDGENAME(targetVertex)
7:     if sourceVertex with sourceName exists in the merged topology then
8:       if targetVertex with targetName exists in the merged topology then
9:         sourcePortIndex = source port number
10:        targetPortIndex = target port number
11:        if sourcePortIndex exists in sourceVertex then
12:          if targetPortIndex exists in targetVertex then
13:            create identification key
14:            create identification reversedkey
15:            if key was not already used then
16:              add new edge to the merged topology
17:              mark key and reversedkey as used
18:            else
19:              mark edge as existing in both
20:            end if
21:          end if
22:        end if
23:      end if
24:    end if
25:  end for
26: end function

```

Algorithm 15 contains the following methods:

- **GetEdgeName**(*currentVertex*) creates the source or the target vertex name as the combination of a device name and a MAC address.

5.2 Comparison Algorithm for Name Assignment

If we want to assign the device name of the device configured in SIMATIC STEP 7 project, we have to find corresponding devices in the network. For each configured device usually exists more then one potential candidates. The purpose of the created comparison algorithm is to find, preferably, the right one or at least prioritize the most similar one. The physical network should be similar enough to the configured one.

As we mentioned in 2, the graph of the network topology can be divided into several disconnected subgraphs. Because of that, the network topology cannot be compared with the configured topology as the one monolithic structure. Moreover, a unique identification using MAC addresses is not possible because they are not present in the configured topology. We had to find some local evaluations of the available matching vertices.

5.2.1 Largest Similar Subgraphs

The first possibility, how to get the proper match, is to search through both topologies and find the largest possible similar subgraphs of the compared vertices. The subgraph is explored from the analyzed vertex until the first difference between the network and the configured topology is found. In the best case this subgraph is the graph of the whole topology. This exploration is similar to the exploration of the edges in the tree layout presented in 4.1.1.3. The vertex in the network topology with the largest subgraph, similar to the subgraph of the configured vertex, is chosen as the best match. For each matching edge, the priority of the matching vertex in the network topology is incremented by one. The search starts simultaneously from the selected configured vertex in the configured topology and from the matching vertex in the network topology. The edges are compared according to the key which consists of the target vertex device type, the source connection port number and the target connection port number. Pseudocode is presented in Algorithm 16.

Algorithm 16 Compares edges until the first difference is found

```

1: function COMPAREEDGES(selectedVertex, matchVertex)
2:   ADDEDGES(configEdges, null, selectedVertex)
3:   ADDEDGES(networkEdges, null, matchVertex)
4:   checkedKey is random number
5:   configuredEdge = first from the configEdges
6:   networkEdge = first from the networkEdges
7:   while configEdges and networkEdges is not empty do
8:     configuredEdge = take first and remove it from configEdges
9:     networkEdge = take first and remove it networkEdges
10:    configuredKey = GETCOMPARISONKEY(configuredEdge)
11:    networkKey = GETCOMPARISONKEY(networkEdge)
12:    if configuredKey is equal to networkKey then
13:      increment matchVertex priority by one
14:      ADDEDGES(configEdges, configuredEdge source, configEdges target)
15:      ADDEDGES(networkEdges, networkEdge source, networkEdge target)
16:    else stop while cycle
17:    end if
18:  end while
19: end function

```

Algorithm 16 contains the following methods:

- **AddEdges** adds edges of the target vertex, which are not connected to the source vertex, to the corresponding list.

5.2.2 Nearest Different

The second approach is based on an observation that the number of vertices to the nearest vertex with the different device type is equal only for a few vertices in the topology. In combination with the previous estimation, the finding of the matching vertex is then even more precise. The line topology is the connection of multiple devices in succession to each other. It is useful even for large topologies with hundreds of vertices. It can eliminate most of the incorrect matches.

This algorithm is based on the modified tree layout algorithm described in 4.1.1.3. It differs in implementation of the method *AddEdges*. If the current vertex device type

and its child vertex device type are different, the number of vertices to the vertex with the different device type has to be updated. It is done for all predecessors of the child vertex with the same device type as the current vertex. The child vertex is the target vertex of an explored edge. Pseudocode of the modified method *AddEdges* is presented in Algorithm 17. In Figure 5.1 we can see an example of the evaluation using this approach.

Algorithm 17 Calculates number of vertices to the nearest different

```

1: function ADDEDGES(parentVertex, currentVertex)
2:   currentType = type of the currentVertex
3:   for each edge of the currentVertex do
4:     if target vertex of the edge is not parentVertex then
5:       if target vertex of edge is not explored then
6:         add edge to the edgeList
7:         target vertex of the edge is now explored
8:       end if
9:       childType = type of the target vertex of the edge
10:      if currentType is not equal to childType then
11:        CALCULATENEARESTDIFFERENT(edge)
12:      end if
13:    end if
14:  end for
15: end function

```

Algorithm 17 contains the following methods:

- **CalculateNearestDifferent**(*edge*) determines the minimal number of vertices to the nearest vertex with a different device type. This is done for each vertex up to the nearest vertex with a different device type.

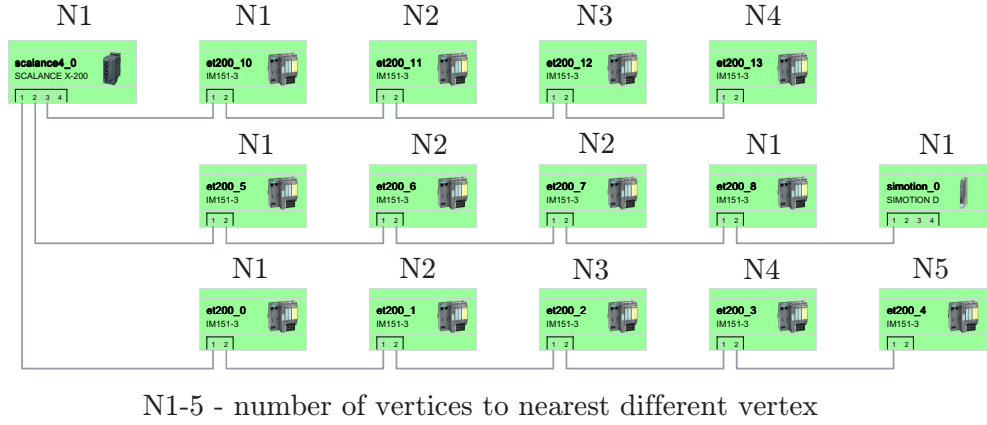


Figure 5.1: An example of the nearest different evaluation

5.2.3 Match priority

The final evaluation of the matching vertices is performed when the user selects the configured vertex in the configured topology and at least one matching vertex exists in the network topology. The matching vertices are vertices with the same device type as the selected configured vertex. The matching vertex priority is determined by the following local criteria:

- Compare edges connected to the vertices according to the source and target vertex connection port numbers and target vertex device type until the first difference is found.
- Compare vertices according to the number of ports. One device type can have several modifications which differs in the number of ports.
- Compare vertices according to the number of vertices to the nearest vertex with different device type.

Pseudocode of the method *DeterminePrioritiesOfMatches* is Algorithm 18.

Algorithm 18 Determines priority of the matching devices

```

1: function DETERMINEPRIORITIESOFMATCHES(selectedVertex, possibleMatchesList)
2:   for each matchVertex in possibleMatchesList do
3:     if selectedVertex is not disconnected then
4:       COMPAREEDGES(selectedVertex, matchVertex)
5:     end if
6:     if number of ports of selectedVertex and matchVertex are equal then
7:       increment matchVertex priority by 1
8:     end if
9:     if nearestDifferent of selectedVertex and matchVertex are equal then
10:      increment matchVertex priority by 1
11:    end if
12:  end for
13:  return ordered possibleMatchesList according to matchVertex priority
14: end function

```

Algorithm 18 contains the following methods:

- **CompareEdges** is described in 5.2.1.

Chapter 6

Component Structure

Structure of the developed component and used technologies are described in this chapter.

6.1 C# and WPF

The component is written in C# and uses WPF for rendering. C# is an object-oriented language that enables developers to build applications that run on .NET framework. This language is described in [20]. Windows Presentation Foundation (WPF) is a presentation system for building Windows applications with the improved visual experience. The core of WPF is a vector based rendering engine that uses hardware acceleration to render graphics. This leads to smoother accelerated visuals. WPF extends this core with development features for creation of GUI that includes Extensible Application Markup Language (XAML), controls, data binding etc.. The data binding is the most important concept of WPF. It serves as data exchange between a control (the binding target) and a data object (the binding source). The presentation layer is then separated from the business layer which is presented in [21]. This is the essential idea of the Model View ViewModel (MVVM) pattern. The view model is responsible for exposing the data objects from the model in such a way that those objects are easily managed and consumed by the View as stated in [22].

6.2 Topology Control

The developed component is called *TopologyControlLibrary*. The main dependencies between *TopologyControlLibrary* and other used libraries are shown in Figure 6.1. All libraries will be described later in this chapter.

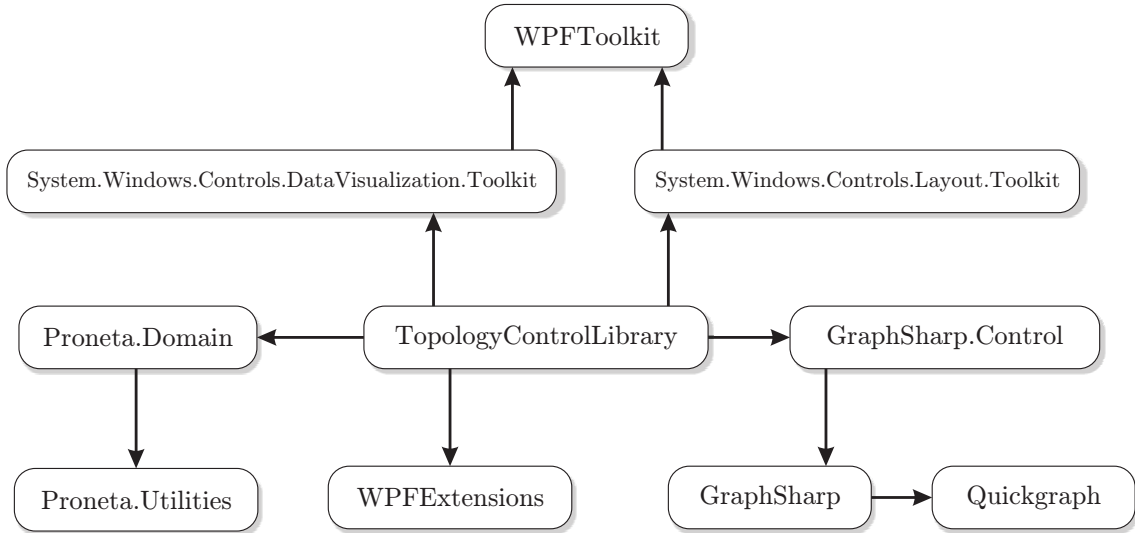


Figure 6.1: Diagram of dependencies between projects and used libraries

The control which is implemented for the visualization of topology is *TopologyControl*. It is a *UserControl* type which can be used in both WPF or Windows Forms.

6.2.1 Structure of Topology Control Library

The *TopologyControl* is the main view. The view model for this view is the class *MainWindowViewModel*. It contains most of the functionality and most of the data structures of the component. It holds all available graphs of the loaded topologies and exposes them to the main view.

The other important parts of the topology visualization are the following:

Converters As stated in [21], the value convertors convert values to reach the data binding compatibility between two incompatible data types. The most important converter in the component is the *EdgeRouteToPathConverter*. It converts the information about the edge route to its visual representation.

DeviceEncapsulations These classes encapsulate the devices and its ports from the topology and adds additional functionality:

- *DeviceData* - The class encapsulates an object which implements *IDevice* interface from the *Proneta.Domain*.
- *PortData* - The class encapsulates an object which implements *IPort* interface from the *Proneta.Domain*.

Helpers These classes provide additional functionality for the *MainWindowViewModel*:

- *TopologyComparison* - Holds both algorithms designed for the comparison of topologies.
- *TopologyPrinting* - Contains methods for saving the topology visualization as an image.

Resources The resources are organized in a way that it is necessary to include only *Resources.xaml* to any XAML file in the project to gain the access to all available images, templates, styles, etc.. The advantage of this approach is that adding new resources to the existing structures is straightforward. It is also possible to use only one of the following groups of resources:

- *DetailedDevices* - Contains XAML data templates for displaying the device details. These templates will be used in the future.
- *Devices* - Holds an internal database of device images.
- *Icons* - Includes icons written in XAML used for indicating the device states.
- *Styles* - Contains all necessary definitions of colored brushes used in the visualization and styles for the main menu and device details.
- *ToolBars* - Holds images used in the main menu and the context menu.

Templates The templates contain styles and data templates which are related to the specific object in the topology visualization such as a vertex or an edge. The following templates are available:

- *ConnectionTemplate* - Defines the drawing of each edge with styles and data templates.
- *DeviceTemplate* - Specifies visual representation of each vertex with styles and data templates.

- *DeviceImageTemplate* - Holds the style for changing the device images.
- *LegendTemplate* - Defines the legend for all available views.
- *PortTemplate* - Determines the representation of ports and their interfaces.

6.3 Graph#

As we mentioned in chapter 3, the basis of the component for topology drawing is Graph#. The Graph# architecture is presented in Figure 6.2.

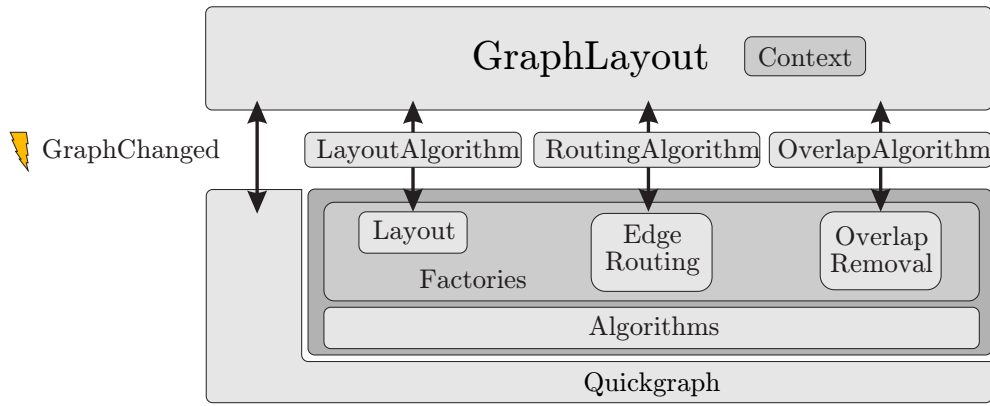


Figure 6.2: The Graph# architecture

The data structures which represents the graphs are based on *Quickgraph* library. The changes of the active graph, such as adding or removing the edges or the vertices, are handled by the *GraphLayout* class in *GraphSharp.Controls* library. The *GraphSharp* library holds all available algorithms for the calculation of layout, edge routing and overlap removal.

6.3.1 GraphSharp

The layout algorithm introduced in 4.1 is integrated in the *Layout* algorithms and its main class is *PronetaLayoutAlgorithm*. The routing algorithm introduced in 4.2 is integrated in the *EdgeRouting* algorithms and its main class is *PronetaEdgeRoutingAlgorithm*.

The graph layout is recalculated whenever a new graph of the topology is loaded or if it is triggered by the user. When the recalculation of the graph layout is needed, the new

instance of the selected algorithm is created by the *StandardLayoutAlgorithmFactory* class. The input consists of the current graph, its vertex positions and sizes.

The edge routing is recalculated whenever the new graph of the topology is loaded or if it is triggered by manipulation with vertices. Whenever the recalculation of a graph layout is needed, *StandardEdgeRoutingAlgorithmFactory* class creates the new instance of the selected algorithm. The input also consists of the current graph, its vertex positions and sizes.

6.3.2 GraphSharp.Controls

The main class in this library is *GraphLayout*. It contains the methods for changing the current state of the visualization of the topology. The recalculation of edges is started by the the method *RouteEdges*. The computed route points are assigned to the edges in method *ChangeEdgesState*. The recalculation of the layout is started by method *Layout*. The computed route points are assigned to the edges using the method *ChangeState*. This library also contains *VertexControl* which is the representation of the device in the visualization and *EdgeControl* which is the representation of the connection in the visualization.

6.4 WPF Extensions

WPF Extensions provides a wide range of extensions for the WPF framework such as controls, attached behaviours, helper classes, etc. that are available at [8]. It is distributed under Microsoft Public License (Ms-PL) in the source code state which allows for additional changes to the source code and commercial uses. One of the available controls is *ZoomControl* class. It is used as an encapsulation of the *GraphLayout* from Graph#. It provides pan and zoom functionality.

The manipulation with objects is implemented in the *DragBehaviour* class. It contains functions for the selection of the object and its dragging of this object. This behaviour can be attached to any object in the visualization.

6.5 WPF Toolkit

It is a collection of WPF features and components which complements a set of WPF controls distributed with .NET Framework available at [7]. The following WPF Toolkit libraries are used in the component:

- *System.Windows.Controls.DataVisualization.Toolkit* - It provides Chart Controls such as Line, Bar, Area, Pie and Column Series. These chart controls will be used for visualization of the network load between devices in the network.
- *System.Windows.Controls.Layout.Toolkit* - It holds the *Accordion* control. It is an animated sliding menu which is used as the main menu in the component.

6.6 Proneta.Domain and PronetaUtilities

Proneta.Domain library is the set of classes and interfaces which represents a topology and its parts in the PRONETA. It is used for smooth integration of the new components to the PRONETA. *PronetaUtilities* library contains useful data structures used in the *Proneta.Domain*.

Chapter 7

Graphical User Interface

A graphical user interface (GUI) consists of two main windows. The main purpose of the first window is to show the network topology overview. It is called Topology View. The second window is the Configuration View which is used for assigning names to the physical devices in the network.

7.1 Overview

Each window consists of one or more instances of *ZoomControl*. The zooming functions are described in table 7.3. We can also zoom by scrolling the mouse.

In Figure 7.5 we can see the vertex which represents the device in the topology visualization. We can identify the device according to the device name and type. Additional information about the devices such as the MAC address or the IP address are included in the tooltip. The device images are currently loaded only from the internal database stored in the component's resources. The database consists of the images of the wide range of the devices produced by Siemens AG. In the future, it will be also possible to load the device images from the GSDML¹ files and support other vendors of the PROFINET devices. We can also see that the device ports are encapsulated into the device interfaces. One device can have several interfaces which will be then separated with a defined distance. There are also the three state icons in the top-left corner. The first icon represents flashing LED² on the physical device which is used for device indication. The two remaining icons are

¹GSDML file is a description of device characteristics at [24].

²LED - light-emitting diode.

visible only in the Configuration View. The second icon is visible when the device name was assigned and the third icon is visible when there exists at least one matching vertex in network topology.






Image	Name	Function
	Show or Hide Menu	It shows or hides the main menu.
	Zoom Box	User can define the area which should be zoomed.
	All	It calculates the zoom according to the dimensions of the topology layout to fill the available area.
	Zoom In	Zooms in to defined higher zoom level.
	Zoom Out	Zooms out to defined lower zoom level.
59 %	Current Zoom	It shows the current zoom in percent form.

Table 7.1: Available zooming functions.

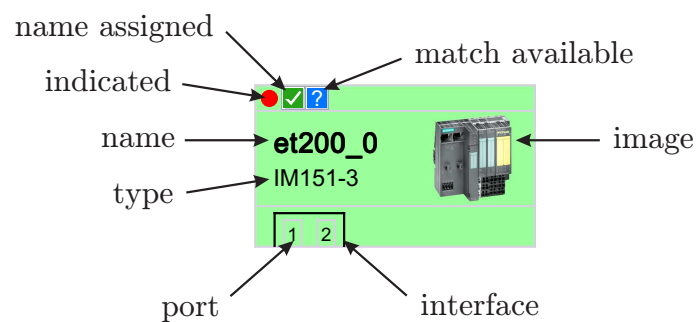


Figure 7.1: Vertex representing a device in visualization.

7.1.1 Topology View

The Topology View shows an overview of the network topology. It provides useful functions for the network diagnostics such as visualization of the network load or the comparison with the saved state of the network (snapshot). In Figure 7.2 we can see an example of the visualization of the network topology in the component.

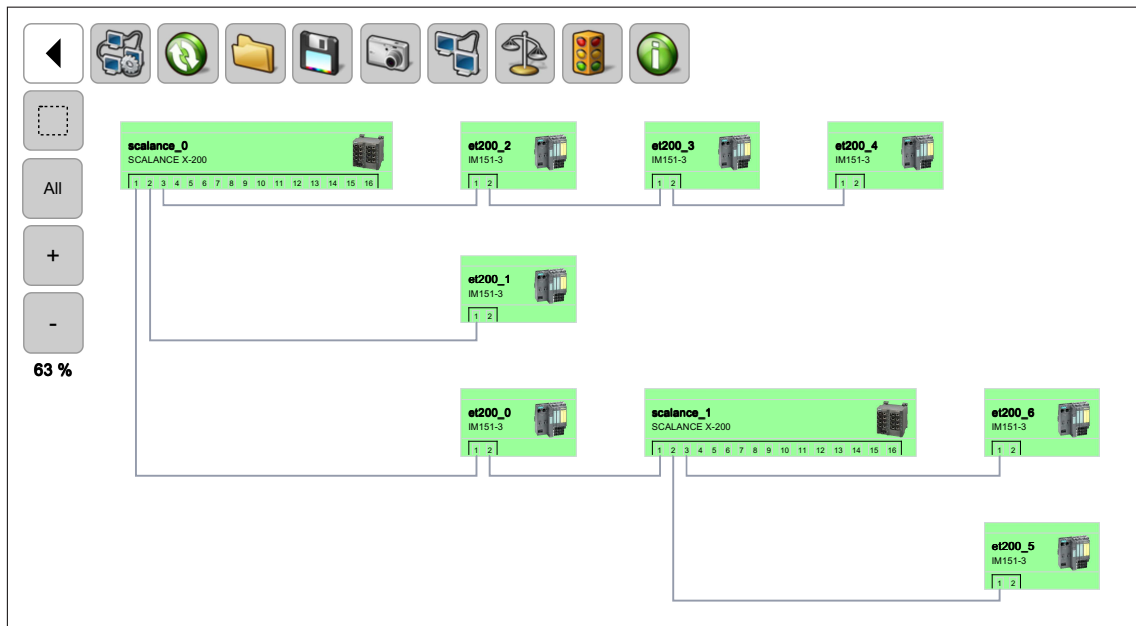


Figure 7.2: Topology View

All the available functions are presented in table 7.2. Now we will describe some details about some of these functions:

- *Network Load* shows communication traffic of connections between the devices. The maximal network load in one direction determines the colour of the edge which is green for the low network load (up to 33%), yellow for the medium network load (between 33% and 66%) and red for the high network load (above 66%).
- The network topology visualization can run in the online or offline mode. In the offline mode, the visualization have to be refreshed to display the recent changes in the network. In the online mode, recent changes in the network are immediately processed and added to the current state of visualization.
- The current state of topology visualization can be saved as an image in bitmapped image formats *.bmp or *.png and in the vector format *.xps. XPS is a XML Paper

Specification. This format provides document appearance similar to PDF.

- The *Legend* is customized for each available view. The colour set used for visualization of the vertices and the edges is the same for all views but the colours have a different meaning.









Image	Name	Function
	Show Configuration View	It change view to Configuration View.
	Refresh	It refreshes online topology and recreates visualization.
	Save Topology	It saves a snapshot of the network in the XML format.
	Save Topology as Image	It saves visualization of topology as image.
	Show Online Topology	It switches between the online and offline mode.
	Show Topology Comparison	It creates and shows the merged topology of the network topology and the snapshot topology.
	Show Network Load	It shows a network load among the physical devices in the network topology.
	Show Legend	The Legend shows information about visualization depending on the current view.

Table 7.2: Available functions in Topology View.

7.1.2 Visualization of Topology Comparison

The creation of the merged topology is described in 5.1. In Figure 7.3 we can see all the possible situations which can occur in the topology comparison. Gray vertices and edges were found only in the snapshot topology. Orange vertices and edges were found only in the network topology. Green vertices and edges were found in both. Red vertex does not have an assigned name.

The edges, which were found in both the network and the snapshot topology and differ only in the connection port numbers, are visualized as they would exist only in the network topology. If the user selects this edge, the edge from the snapshot topology appears in the visualization. This behaviour makes the visualization clearer.

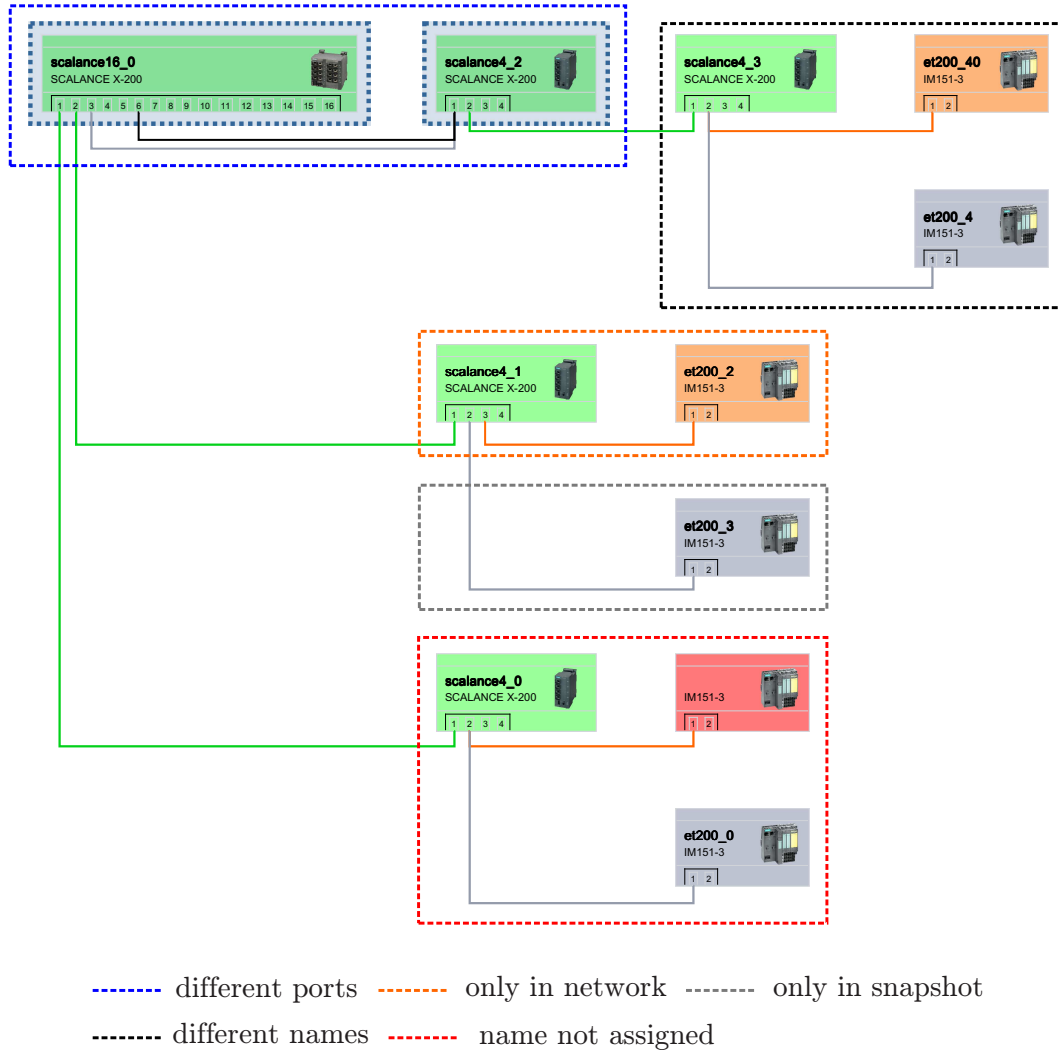


Figure 7.3: Visualization of the merged topology

7.2 Configuration View

As mentioned before, PRONETA can parse the exported SIMATIC STEP 7 project file. The devices and the connections in this project are transformed to the configured topology with the projected device names. This topology is visualized as the template for the network. The Configuration View easily allows the assigning of the configured device names to the physical devices in the network.

7.2.1 Overview

In Figure 7.4 we can see that the window is divided into three parts. The top half of the window shows the configured topology overview. It can also show the network overview. The bottom-left quarter of the window shows a subgraph of the selected vertex in the configured topology. The bottom-right quarter of the window shows a subgraph of matching vertex found in the network topology.

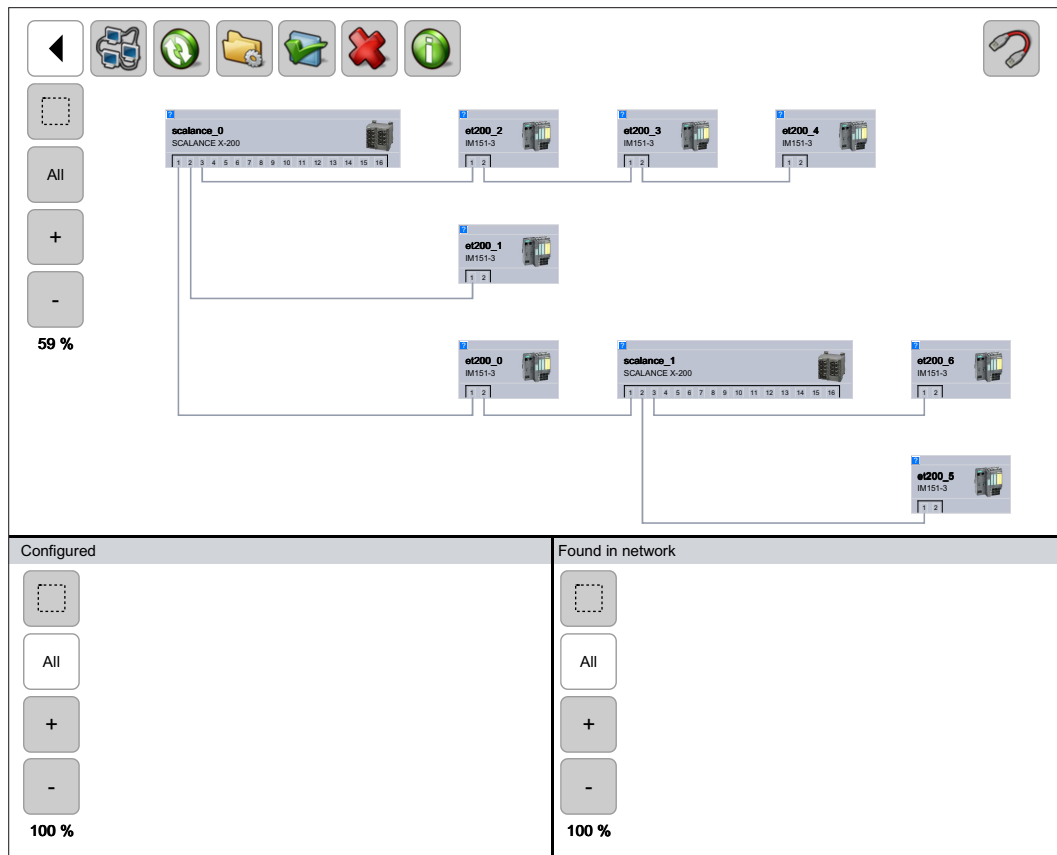


Figure 7.4: Configuration View






Image	Name	Function
	Topology View	It changes the view to Topology View.
	Refresh	It recalculates the layout of the configured topology.
	Set Configuration	It sets the device names from the vertices with an assigned name to the physical devices in the network.
	Reset Configuration	It resets assigned names to the original state.
	Show Network Topology	It shows the network topology in the Configuration View.

Table 7.3: Available functions in Configuration View.

7.2.2 Configuration Selection

The user selects the configured vertex from the configured topology visualized in the top of the window. The subgraph of this vertex is then visualized in the bottom-left corner of the window. This subgraph consists of the selected vertex and the directly connected edges and vertices and it is highlighted in the configured topology. The available matches of the selected vertex are ordered according to the priority calculated in 5.2. The subgraph of the vertex with the highest priority is selected from the network topology and visualized in the bottom-right corner of the window. An example of the configured vertex selection is presented in Figure 7.5. We can see that the proper match was found and marked with the orange bounding box. If the vertex with the highest priority is not the right one, the user can find a proper match manually using a menu of all available matches which is accessible from the bottom-right quarter of the window.

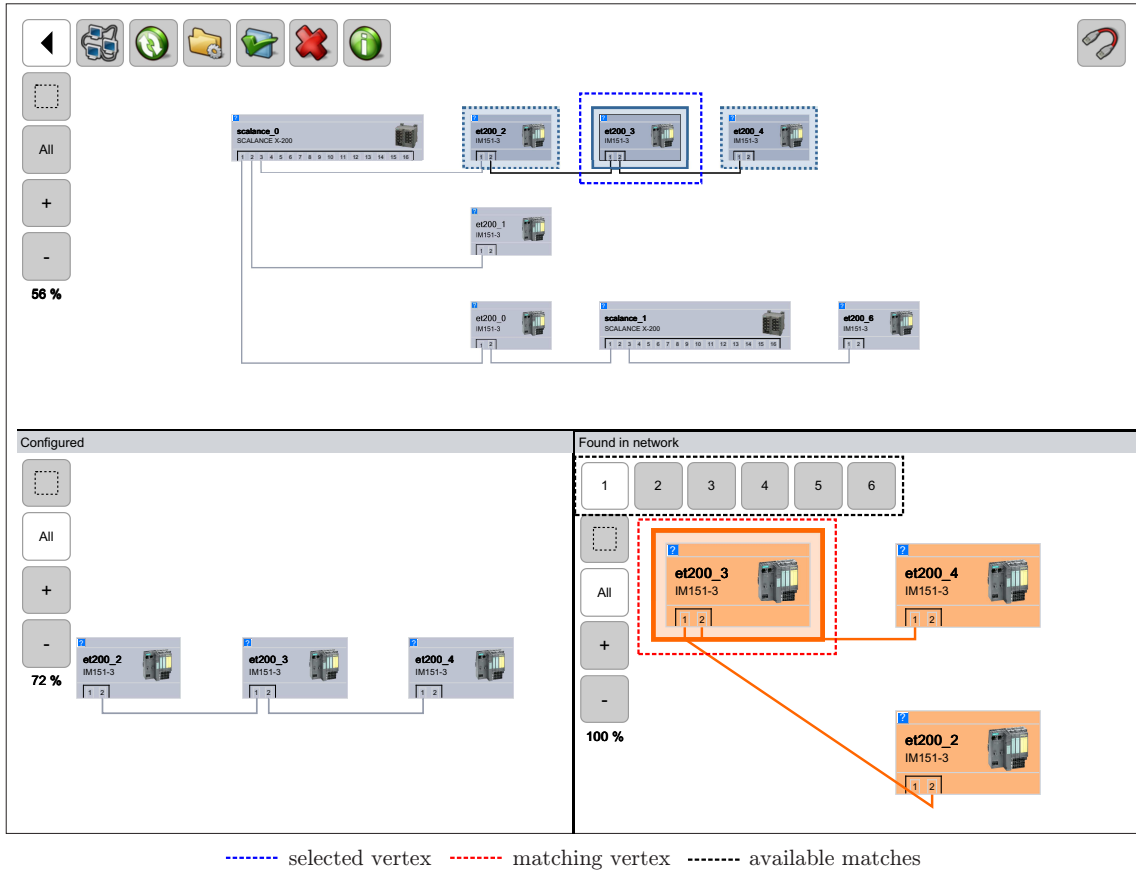


Figure 7.5: Selected configured vertex

7.2.3 Device Name Assignment

When the configured vertex is selected and the proper match in the network topology is found, the user can easily assign a device name by dragging the configured vertex and dropping it on the network vertex in visualization. The user has to click and hold down the left mouse button on the configured vertex in the bottom-left corner of the window. When the slightly transparent vertex has appeared next to the mouse cursor the selected configured vertex is dragged. A device name is assigned when the configured vertex is dropped on the position of the network vertex by releasing the mouse button. An example of the name assignment is presented in Figure 7.5. We can see that vertices in the subgraph in the bottom-left corner of the window are placed at the same positions as there are in the configured topology overview in the top half of the window. Moreover, each vertex with the assigned name in the bottom-right corner of the window is moved to the same position as the corresponding configured vertex. This feature provides a visual

comparison of the vertices in the configured topology and the vertices with the assigned name in the network topology.

Each vertex with the assigned name can be reset to their original state individually using the function *Reset Configuration* accessible from the context menu. To reset all vertices to their original state, one has to use the function *Reset Configuration* accessible from the main menu. Function *SetConfiguration* accessible from main menu set the assigned device names to the physical devices in the network.

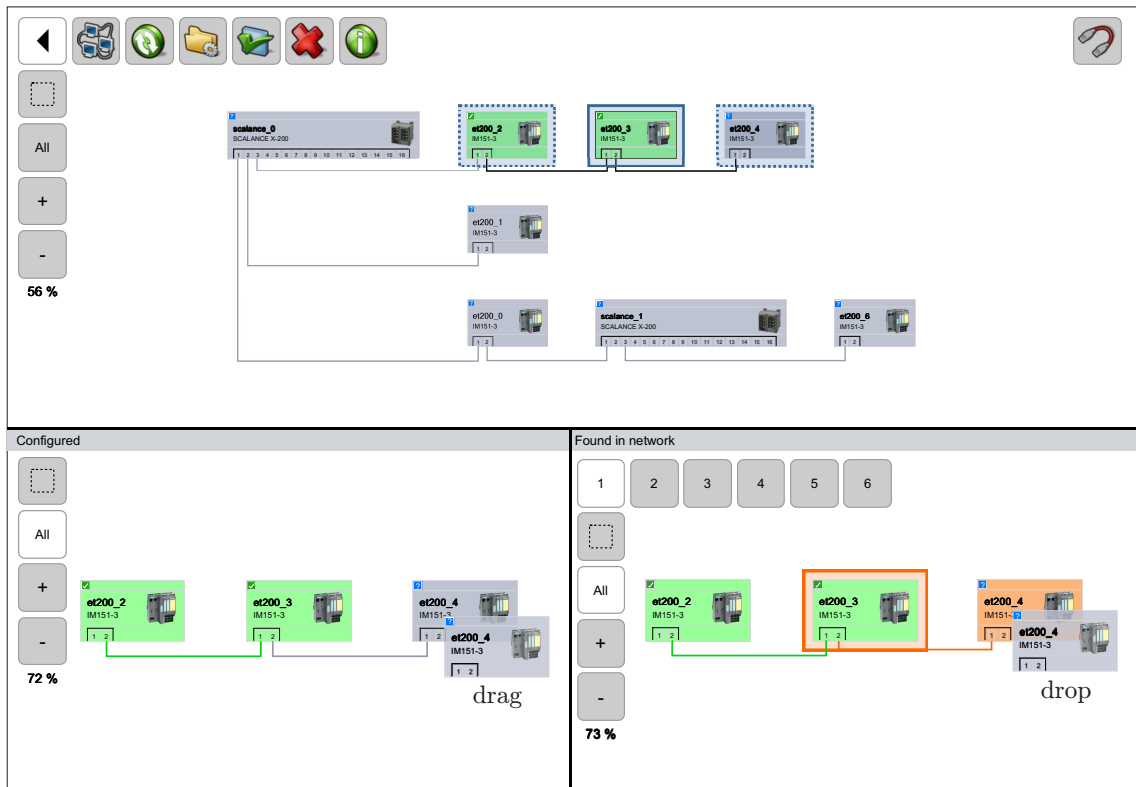


Figure 7.6: Name assignment

7.2.4 Reconstructed switch

As mentioned in chapter 2 the generic Ethernet switch can be reconstructed from for visualization to avoid redundant edges between vertices. In Figure 7.5 we can see the result of this reconstruction. The generic Ethernet switch is shown with a red colour because it can represent one or more generic Ethernet switches connected to each other. We can also see a blue vertex which represents PRONETA PC.

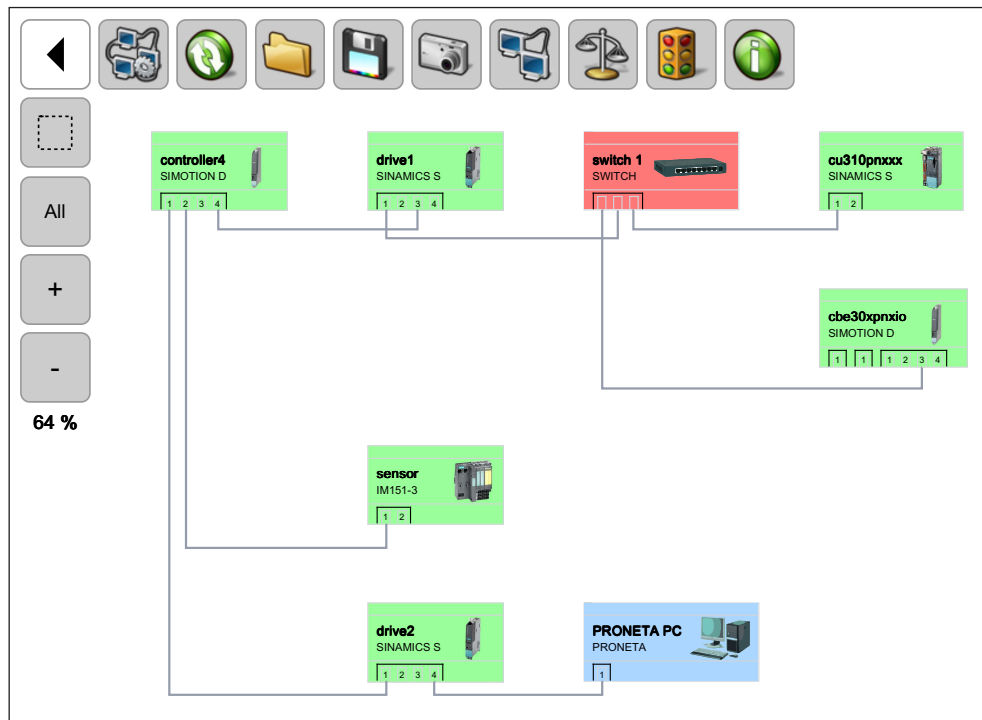


Figure 7.7: Reconstructed generic Ethernet switch in the topology.

Chapter 8

Conclusion

The main purpose of this thesis was to implement a component for topology visualization. After choosing the appropriate framework, it has been implemented in C# using WPF for rendering. Several frameworks for graph visualization were considered as a basis for the developed component. After consideration of all advantages and disadvantages of these frameworks, the Graph# was chosen as the most appropriate. The core of the topology drawing consists of the algorithms for computing vertex layout and edge routes. The layout algorithm is based on determining root vertex, computing modified rooted tree layout and calculating the positions of the disconnected vertices. The approach is fast enough to recalculate the layout several times for each subgraph in the topology and find the optimal root vertex. The optimality is reached by minimization of the size of the layout for each subgraph in topology. The routing algorithm computes the optimal orthogonal edge routings. Optimality is reached by the minimization of the connection length and the number of bends. The approach is based on first computing an orthogonal visibility graph, then an optimal route using an A* search algorithm, followed by post-processing which removes shared edge segments and determines the final position of the edge routes in the visualization. After applying some optimization techniques, this algorithm is fast enough to recalculate the optimal connection routings even during the direct manipulation of vertex position in the visualization. This feature gives instant feedback to the user.

The functionality of the topology visualization component has been extended by two algorithms for topology comparison. The first algorithm creates one merged topology from two similar topologies. The merged topology is then displayed with highlighted differences. This feature is useful for the detection of changes between the current state and the previous state of the same network. The second algorithm determines the priority

of available matches for name assignment. The name assignment is another implemented extension of the component functionality. It is based on drag-and-drop intuitive interface. The device names can be easily assigned from the configured topology to the physical devices in the network.

The developed component was integrated into PRONETA. It was already used for diagnostics of industrial networks. The component is fully functional to provide useful information about analyzed networks. We successfully tested this component to analyze the network with over the six hundred connected devices.

In the future, it is possible to extend this work in the following ways:

- Develop methods allowing incremental changes to the orthogonal visibility graph used in the routing algorithm. The current implementation generates new orthogonal visibility graph whenever the edge routes have to be recalculated. This causes unnecessary overhead if vertex was not added to or removed from the topology.
- Implement better support for the placement of two similar vertices connected with its edges to the same connection ports. This situation occurs in topology comparison. The current layout algorithm is not able to place these two similar vertices under each other to make it clear that they are at the same position in the topology.
- Find additional criteria to make comparisons of the network topology and the configured topology even more accurate to provide more accurate matches.
- Improve layout algorithm for visualization of rings in the topology. The vertices should form an actual ring. It will be more obvious that the ring exists. This feature will be optional.
- Add images from GSDML files to the image database in the component to provide support for a wide range of devices in the visualization.

Bibliography

- [1] BATTISTA, Di Giuseppe; EADES, Peter; TANASSIA, Roberto; TOLLIS, G. Ioannis: Algorithms for the Visualization of Graphs, PRENTICE HALL, 1999. p. 387 ISBN 0-13-301615-3
- [2] NACHMANSON, Lev. MICROSOFT. (2012, Feb. 25). MSAGL - Microsoft Automatic Graph Layout [online]. Available: <http://research.microsoft.com/en-us/projects/msagl/>
- [3] NACHMANSON, Lev. MICROSOFT. (2011, Sep. 25). GLEE - Graph Layout Execution Engine [online]. Available: <http://research.microsoft.com/en-us/downloads/f1303e46-965f-401a-87c3-34e1331d32c5/default.aspx/>
- [4] AT&T LABS RESEARCH. (2012, Feb. 25). Graphviz - Graph Visualization Software [online]. Available: <http://www.graphviz.org/>
- [5] QuickGraph - Graph Data Structures and Algorithms for .NET. (2011, May 13). [online]. Available: <http://quickgraph.codeplex.com/>
- [6] Graph#. (2009, Jun. 1). Graph visualization framework. [online]. Available: <http://graphsharp.codeplex.com/releases/view/28088>
- [7] WPF Toolkit. (2010, Feb. 16) Set of WPF controls for .NET framework [online]. Available: <http://wpf.codeplex.com/releases/view/40535>
- [8] WPF Extensions. (2009, May 1) Set of WPF controls for .NET framework [online]. Available: <http://wpfextensions.codeplex.com/>
- [9] MARRIOTT, Kim; STUCKEY, Peter J.; WYBROW, Michael. Proceedings of 17th International Symposium on Graph Drawing (GD 09): Orthogonal Connector Routing [online]. Springer, 2010. Available: <http://www.csse.monash.edu.au/marriott/publications-web.html>

- [10] MARRIOTT, Kim; STUCKEY, Peter J.; DWYER, Tim. Proceedings of 13th International Symposium on Graph Drawing (GD 05): Fast Node Overlap Removal [online]. Springer, 2006. Available: <http://www.csse.monash.edu.au/~marriott/publications-web.html>
- [11] MARRIOTT, Kim; STUCKEY, Peter J.; DWYER, Tim. Proceedings of 13th International Symposium on Graph Drawing (GD 05): Fast node overlap removal - Addendum [online]. Springer, 2006. Available: <http://www.csse.monash.edu.au/~marriott/publications-web.html>
- [12] NATHAN, Adam. Windows Presentation Foundation Unleashed.: SAMS, 2006. p.638 ISBN 978-0672328916.
- [13] ANDERSSON, Arne. Proceedings of Workshop on Algorithms and Data Structures: Balanced Search Trees Made Simple[online]. Springer Verlag, 1993. Available: <http://user.it.uu.se/~arnea/abs/simp.html>
- [14] WALKER, Julianne. (2007) Andersson Trees [online]. Available: http://eternallyconfuzzled.com/tuts/datastructures/js/tut_andersson.aspx
- [15] DEMA KOV, Aleksey. (2007) Balanced Search Trees Made Simple (In C#) [online]. Available: <http://www.demakov.com/snippets/aatree.html>
- [16] DEMA KOV, Aleksey. (2008, Dec. 12) Implementing a Generic Binary Tree in C# [online]. Available: <http://www.dijksterhuis.org/implementing-a-generic-binary-tree-in-c/>
- [17] WIKIPEDIA. (2011, Jun. 10). AA tree [online]. Available: http://en.wikipedia.org/wiki/AA_tree
- [18] ANDREWS, Keith; WOHLFAHRT, Martin; WURZINGER, Gerhard. (2009, Aug. 4). Proceedings of 13th International Conference Information Visualisation: Visual Graph Comparison [online]. Available: <http://ieeexplore.ieee.org/>
- [19] KLEINEBERG, Oliver; FELSER, Max. (2008, Oct. 3). Network diagnostics for industrial Ethernet [online]. Available: <http://ieeexplore.ieee.org/>
- [20] MICROSOFT. (2012, Apr. 1). Visual C# [online]. Available: <http://msdn.microsoft.com/en-us/vstudio/hh388566/>

- [21] MICROSOFT. (2012, Dec. 1) Windows Presentation Foundation [online]. Available: <http://msdn.microsoft.com/en-us/library/ms754130/>
- [22] SMITH, Josh: MICROSOFT. (2010, Feb. 16). MVVM Foundation [online]. Available: <http://mvvmfoundation.codeplex.com/>
- [23] yWorks. (2011, Sep. 10). Diagramming Components [online]. Available: <http://www.yworks.com/en/index.html>
- [24] Siemens AG. (2012, Apr. 30). Automation [online]. Available: <http://www.automation.siemens.com/>

Appendix A

Content of the Attached CD

To this work is attached CD which contains this work and the whole code of the created component in C#.