Bachelor Thesis

Zhenrui Yue
Matriculation Number: 03674040

Development of a Concept for Visualizing Variation Points in industrial PLC Software

Supervisor:	M.Sc. Safa Bougouffa
Edition:	20.11.2018
Submission:	23.11.2018

Lehrstuhl für
Automatisierung und
Informationssysteme
Prof. Dr.-Ing. B. Vogel-Heuser
Technische Universität München
Boltzmannstraße 15 - Geb. 1
85748 Garching bei München

Telefon 089 / 289 – 164 00
Telefax 089 / 289 – 164 10
http://www.ais.mw.tum.de

Abstract

Automation systems controlled by programmable logic controller (PLC) in standard programming languages IEC61131-3 are very effective and robust systems widely used in automated production facilities. These automation systems, especially the software components of them, are frequently replicated to adapt to changing requirements in different production processes [1]. One of the methods for convenient software modification is called clone-and-own, meaning to create new variants by copying and modifying the current version, which quickly generates various software systems that share similar characteristics and same components.

To manage similar software variations, a tool to analyze. visualize and compare the variation points is very helpful in the practice. This thesis presents a visualization concept for comparing software systems in IEC 61131-3 programming standard formatted in XML files, in which the structure, relationships, attributes as well as comparisons of automation software systems are graphically represented with the family model mining method. Additionally, the software provides a concept for simple configuration possibilities like addition and deletion for PLC systems to quickly comply with different demands. The thesis will implement this concept to compare and visualize scenarios in automation systems and evaluate its potential applicability with case studies. Afterward, conclusion and future work are defined for further development of the visualization concept.

Table of Contents

Abstract II
1. Introduction 1
1.1. Motivation 1
1.2. Background 2
1.2.1 Automation System 2
1.2.2 Programmable Logic Controller 2
1.2.3 The IEC 61131-3 Programming Standard 3
1.2.4 Family Model Mining Approach 4
1.2.5 Software Visualization 4
1.3. Overview of the Thesis 5
2. State of the Art 7
2.1. Software Visualization 7
2.1.1 Software Visualization Aspects 7
2.1.2 Structure and Architecture 9
2.1.3 Relationship 10
2.1.4 Evolution 11
2.1.5 Metric-based Visualization 13
2.2. Visualization of PLC Software System 14
3. Requirements 16
3.1. Software Development Process 16
3.2. Data Input (Data Processing) 18
3.3. Visualization Output (Functional Specification) 19
3.4. Requirements on Programming Standards 21
4. Design of Visualization Concept 23
4.1. Graph Design 23
4.1.1 Design of Tree Diagram 23
4.1.2 Design of Network Graph 24
4.1.3 Design of Tree Map 25
4.2. Design of the Ring Graph 25
4.3. Design of the Bubble Graph 26
5. Implementation of Visualization Concept 28
5.1. Visualization Framework 28
5.1.1 The Prefuse Visualization Toolkit 28
5.1.2 The Java Universal Network/Graph Framework 29
5.1.3 The dom4j XML framework (related library) 30
5.2. Visualization Process 30

5.3. Visualization of Tree Diagram, Network Graph and Tree Map 32
5.3.1 Tree Diagram 33
5.3.2 Network Graph 33
5.3.3 Tree Map 34
5.4. Visualization of Ring Graph and Bubble Graph 35
5.4.1 Ring Graph 35
5.4.2 Bubble Graph 37
6. Evaluation 39
6.1. Evaluation 39
6.1.1 Visualization and Functionality Evaluation 39
6.1.2 Performance Evaluation 40
7. Conclusion 42
7.1. On Visualization 42
7.2. On Implementation 42
7.3. Future Work 43
List of Figures 45
List of Tables 46
References 47
Index 51
Appendix 52
Appendix A: Code of Tree Diagram 52
Appendix B: Survey Questionnaire Demo 1 58
Declaration of Authorship 60

1. Introduction

This thesis presents a visualization concept for visual representation of software components in automation systems. With this concept, multiple PLC software systems and related variants could be quickly analyzed and compared, with its results delivered in visualized form, such as network graph or other visual forms presented in the thesis. The visualization concept simplifies the process to identify, compare and modify different components, it also reduces the workload to fix errors and update certain code in the automation system. For the development of this visualization concept, the thesis will first introduce the problem and motivation of PLC software visualization, then shortly clarify a few relevant definitions in the first chapter.

1.1. Motivation

Automation systems and facilities based on programmable logic controllers (PLCs) are widely used in manufacturing industry and the majority of them are compatible with the IEC 61131-3 programming standard, which is 3rd part of the open international standard IEC 61131 for programmable logic controllers published in February 2013. IEC 61131-3 programming standard defines 5 programming languages including textual and graphical programming languages independent from hardware systems. Today, most PLCs and related products adhere to IEC 61131-3 and it became a widely accepted standard in automation technologies and manufacturing industry [2].

Automation systems are in most cases, fully customized regarding both its hardware and software, while hardware of automation systems is usually dependent from operation scenario, software systems under the IEC 61131-3 standard are the components that could be modified independently from the hardware system. The software of automation systems based on PLC is often changed to meet different requirements in the manufacturing process. Since slight changes are more frequent in the production lifecycle, the software systems usually require only small modifications, and the unaffected components could be directly taken over. One of the methods that are constantly used in this modification process is called clone-and-own, which means copy from a current system and then modify some of its components based on production requirements to create a variant different from the original version.

As number of software system variants increase during the whole manufacturing lifecycle, management of existing software versions of automation systems and sometimes even large amounts of variants due to numerous production adjustments requires well-structured, modularized and documented contents in the software systems, along with huge efforts of the personal during identification and documentation. And since by using the clone-and own method, only minor changes are conducted without any documentation, it could result in numerous software variants only slightly different and thus, great difficulties in the selection, testing and maintenance of the automation system. Also, finding a specific element in the related variants and correction of an error in all possible variants could be extremely challenging without any specific tool to deal with similar software variants.

Therefore, a visualization toolkit for automation systems based on PLC aiming at the solution of these problems became the motivation of this thesis. The toolkit should solve the problem of managing similar software variants under the IEC 61131-3 programming standard by visualizing the similar variants and their components, so that a clear overview of all variants could be graphically demonstrated.

1.2. Background

1.2.1 Automation System

Automation is the process performed by an automation system with no human interference. Automation systems are very common in the modern manufacturing systems such as robots, they consist of various hardware components and a control mechanism (software components) which regulates the behavior of the automation system. Common hardware components are e.g. electric motors and sensors. They are used to carry out certain actions to complete the automation process, such as identifying object properties or transporting object from one to another location.

The control mechanisms mainly implement two methods: feedback control and sequential control. An example of the feedback control mechanism is the proportional-integral-derivative controller (PID controller), which takes the difference between the target value and measured value as an input and calculates the result based on a proportional, an integral and a derivative term, then the PID controller uses the result to correct the output and consequently controls the behavior of the automation system [3]. A sequential control system typically has various states, which define different actions according to the state conditions, and the transition of various states are performed on a sequential basis, or when certain conditions are fulfilled. Sequential control systems are often used, when different actions and their transitions are involved. An example is the start-stop circuit of a motor, which contains two states and two transitions that define when to turn on and when to turn off the motor.

In modern manufacturing industry, computers are used to perform the control mechanism of an automation system. Since computer could be programmed to achieve different control mechanisms, they quickly replaced other controllers. One of the common controllers used in the automation industry is the programmable logic controllers (PLCs), PLCs are industrial digital controllers with microprocessors that could be programmed to perform different control tasks in the manufacturing process. Details of the programmable logic units is introduced in the next subchapter.

1.2.2 Programmable Logic Controller

Programmable logic controllers (PLCs) are industrial computers containing microprocessors that could be programmed to process different parameter as input signals and calculate output accordingly. They are widely used in the manufacturing industry, especially during the production process, where they could control various automated facilities such as industrial robots with high precision and reliability. PLCs are modularized devices that usually consist of multiple electronic components such as central processing unit (CPU), internal memory, I/O unit and power module, which allow PLCs to have excellent computation capacities and

real-time characteristics in program execution process. Therefore, PLCs are frequently adopted for their high reliability and real-time applicability in extreme conditions, which are perfect characters for automation systems [4].

An important feature of PLCs is that they could be programmed anytime to achieve different functionalities and control tasks such as sequential control or feedback control with the same hardware, only the software components in the PLC are modified and uploaded. As a result, PLCs could be mass produced which have much lower prices compared to hardware-specific controllers and they could be customized later with the different PLC programs. In the automation industry, PLCs are used to realize different complex control mechanisms of automation process in mechanical or electrical systems like regenerative thermal oxidizers due to their cost efficiency and high flexibility. Modern PLCs could be programmed in various programming languages including graphical and textual programming languages such as instruction list (IL) regulated in the IEC 61131-3 programming standard. An example product of PLC is the Siemens SIMATIC S7 series and SIMATIC WinAC software PLCs for industrial PCs, which could be programmed and deployed for diverse purposes in the automation technologies.

1.2.3 The IEC 61131-3 Programming Standard

Most PLCs support the programming standard of IEC 61131-3, which is a set of standard programming languages for programmable logic controllers published in February 2013. IEC 61131-3 is the 3rd part of the IEC 61131 international standard for programmable logic controllers (PLCs), which includes five programming languages: function block diagram (FBD), ladder diagram (LD), structured text (ST), instruction list (IL) and sequential function chart (SFC). The five programming languages enable both textual (structured text and instruction list) and graphical (ladder diagram, function block diagram and sequential function chart) programming of PLCs.

In IEC 61131-3 software systems, there are 3 levels of program organization units (POUs): functions, function blocks and programs. Functions are created to perform simple action such as addition and subtraction, they could only call other functions, once a function is created, they could be used repeatedly in the same project or even in other projects. Function blocks are more complicated units to achieve more complex functionalities, they could also call other functions and function blocks to simplify the programming process, so that they are usually highly reusable to perform the same control tasks in the program. Programs are higher levels of programming units that contain more components and could call both functions and function blocks, they could also be seen as a hierarchical structure of sub-elements like functions or function blocks [5]. By using the POUs, IEC 61131-3 software systems are better structured and modularized, with complicated behavior broken down into different small parts, so that every software system could contain numerous components and eventually be represented in a hierarchical structure for overview and possible further development. A typical IEC-61131 program has a hierarchical structure and is made up of numerous POUs of different levels, each containing certain functionalities and possibly further relationships with other POUs, so that a PLC software system is very suitable to be visualized in graphs with visual items and corresponding relations.

1.2.4 Family Model Mining Approach

A family model refers to a 150%-model containing often more than one comparable system, where all possible elements, variation points and their relevant information could be found. It is a superimposition of all related software systems and their components. In a family model, same elements are merged and referred as mandatory elements, different elements could be further classified as alternative elements or optional elements. Alternative elements are elements that exist in involved systems, but containing different information, whereas optional elements represent elements that only exist in one of the involved systems. To create a family model, a family model mining approach of 3 sequential phases is introduced and implemented.

In this approach, all components of involved software systems are processed in three phases: Comparing, Matching and Merging. After the process, a family model with various mandatory, alternative and optional elements is created. In the Comparing Phase, the elements (function blocks etc.) of one variant is compared with every element of a second variants using a metric, which assigns weights on different element properties. After the comparison, we have the similarity value (between 0 and 1) of every possible combination. In the Matching Phase, the elements are matched according to the name or the similarity value. After the Matching Phase, a 150% model is created with all matched elements from the last phase, identical elements will be matched as mandatory elements. Similar elements with similarity value between 0 and 1 will be matched and considered as alternative blocks. The elements with similarity value of 0 are called optional elements, which means they only exist in one of the systems. Afterwards, the Merging Phase creates a modified 150% model with mined information about different properties of the elements, same mandatory elements are simply merged, matched alternative elements are moved into a so-called VariantSubsystem, while optional elements stay preserved and marked as optional [2]. An example of a PLC software family model is shown in Figure 1.

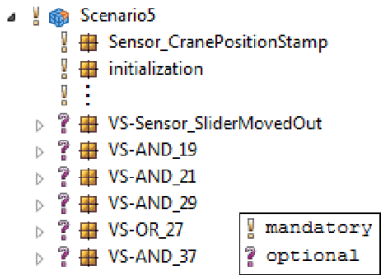

Figure 1 Example of a family model [Holthusen et al., 2014]

The family mining approach could be applied to the PLC software systems, as variant number keeps growing with constantly changing production process. The approach could analyze variation points of different PLC software systems and create a family model that contains all information from the software systems. The objective of the thesis lies in the visualization of the family model (and single PLC software systems), so that the structural information, software components as well as variation points between the software systems could be visually represented.

1.2.5 Software Visualization

Software visualization is the visual representation of software systems and related information such as component relationships, hierarchical structure and detailed implementation. The

software architecture is the most important aspect regarding software visualization, this includes the software organization and various relationships such as inheritances and call dependencies. The visual representation of software systems is mostly in the form of graphs that consist of numerous nodes and edges representing components and relationships, a common example of which is the Unified Modeling Language (UML) class diagram. A UML class diagram describes the software with different classes, their attributes as well as methods in the graph with brief textual information in a text box as nodes, the relationships like inheritance and dependency among the components are represented as the different edge shapes in the graph, which forms a software description containing structural and relational information based on its classes [6].

Software visualization aims at simplifying the related information of a software system, which includes the architecture, organization, components and their attributes and the relationships between these elements. These visualization purposes could be fulfilled with simple graphs such as tree diagrams or network graphs, they could be used to help developers understand the basic construction of a complicated software system and constraints among various components. Besides, software visualization could be used to visualize the behavior during the execution process and even the evolution process of the software, behavior visualization could greatly reduce the efforts to find errors and avoid unexpected behavior in the software systems, whereas the visualization of software evolution process gives a brief introduction to the changes in the history of the software development and the chronological tendencies of different elements to help developers determine directions of further development. Furthermore, there are more methods such as text-based visualization, but due to the constraints of the thesis, we will focus on the certain visualization types, which will be further introduced in chapter 2.

1.3. Overview of the Thesis

Software systems of automated facilities based on the IEC 61131-3 programming standard for programmable logical controllers are comparable to software architectures in other programming languages. PLC programs have a well-structured hierarchical organization based on different levels of Program Organization Unit (POU) mentioned above but could eventually contain numerous components since each POU is designed only to achieve simple functionalities. Also, the relationships in software systems could be rather complicated due to large amounts of inheritance, call relationships and differences of scenarios in the family model. Because of the complexity of the visualization process, the development of the concept is divided into different sections such as visualization analysis, the concept, implementation and evaluation. They will be discussed in separate chapters to achieve the purpose of developing an appropriate visualization concept for variation points in PLC software systems.

In the first chapter, the motivation and background knowledge are shortly introduced. The thesis will focus on the state of the art in software visualization in the next chapter, followed by the specified visualization requirements in chapter 3. Afterwards, we will introduce several visualization concepts for PLC software systems and the implementation of these ideas. After the development, this toolkit will be evaluated regarding different criteria in aspects of visual effects and software functionalities. A conclusion of the thesis is followed with future work

that defines some features to be further developed in the future. The thesis is accordingly divided into the following sections:

- Motivation and background
- Research of the status in PLC software visualization
- Requirements on the visualization toolkit
- Design concepts for PLC software visualization
- Implementation of the visualization concept
- Evaluation of the implemented visualization
- Conclusion and future work of the theses

2. State of the Art

2.1. Software Visualization

Software visualization is defined as the visualization of software systems regarding the related software information such as structure or components. Software visualization methods are frequently used to help understanding the architecture of a software, that is, the structure, components, their attributes and the relations between these elements, which is more common with complicated computer software. Moreover, software visualization could be used to visualize the relationships among components and their behavior during the execution process, which could greatly reduce the efforts to find errors and avoid unexpected behavior in automation software systems such as PLC software, as they are designed to control the behavior of various facilities [7].

As PLC software systems are comparable to software architectures in other programming languages, we will first discuss different visual aspects of software visualization in this chapter, which include many visual forms that are currently used in software visualization technology, we will also introduce the current state of software visualization in these aspects. Afterward, we will concentrate on the status of the PLC software systems and discuss the state of the art in PLC software visualization.

2.1.1 Software Visualization Aspects

The selection of visualization aspects is a critical step before the visualization itself, as it directly affects the visualization information and the visual effects of the outcome. A proper visualization should focus on important information and try to highlight this information with abstract overview of other visual elements. These methods have different advantages and disadvantages in visualizing different software components, so they can't be simply applied to every visualization. With analysis of relevant literature focusing on the software visualization, the visualization aspects could be concluded in the following categories:

- Structure and Architecture: architectures reflect the structural information of the software, which goes from the hierarchical structure of the software project down to class-centered description of software components. To focus on the architecture of a software, it usually requires less information and could create visualizations that quickly transfer a basic understanding of the software. However, visualizations from an architectural aspect often lack textual information and details in components, which result in difficulties to acquire further information of the software. A common way to visualize the hierarchical structure is to use a tree diagram, which represents the hierarchical structure of the software and separates components in different level of depth for structural position [8].

- Relationships: relationships in a software structure reveal its complexity and the importance of different software components, which could sometimes be much harder than visualization of architectural information, because components could have a numerous number of relationships. Relationships could also be divided into different types, call dependency, inheritance and accesses are common relationships of software

components. They reflect the software components and the information flow, certain component properties, e.g. reusability could also be indirectly gained. Relationship visualization loses information such as the software structure and component detail and leads to the same problem in architecture visualization. The expression of relationships in software components is mostly lines (curves) in the form of class diagram or network graphs, which could emphasize different relationship types [9].

- Evolution: evolution focuses the changes of a software in the development or updating process and visualizes the changing structure of the software system as it continues to develop. With evolution of software systems, the number, the size or even the content of components changes in fast paces. The variations are captured in a single visualization with different development phases, which reveals the developing direction of various components in the evolution process. Software evolution could be used to track development process and avoid version control problems. Software evolution could be visualized with stream graph that uses stream width to represent various component attributes in different time scale [10].

- Behavior: software behavior visualization realizes the visualized execution process of a software system, which assist programmers with understanding the runtime behavior of the software. The visualization of the software behavior demands a more abstract description of the software structure and corresponding work flow, which could be very complicated since software systems usually possess numerous components for different functionality and many components could be executed at the same time. Behavior visualization aims at solving problems on runtime behavior of a software and could vastly accelerate the testing and validation process in the software development. It could be represented in flow charts or tree-like graphs, which describes the sequential actions, dividing points and possible loops of the execution process [11].

- Metric-based visualization: metric-centered visualization is frequently used to evaluate a software regarding different criteria such as runtime performance or required system resource. It can quantify certain characteristics of a software and offer different method to describe a complex software system, which could be effectively visualized and evaluated. Common visual techniques include radar charts or bar charts that describe different characteristic values [12].

- Code line: code-based visualization deals with source code visualizations, since source code can’t directly be treated as visual items, lines or instructions such as a while loop are treated as basic visualization unit. Code-based visualization helps programmers with finding errors in the source code or optimizing the code structure, it’s widely used in software implementation process, it also functions to compare different code files or annotate changes in software evolution process [13].

The visualization of the PLC software systems in the thesis has an important task, which is to visualize the variation points of different PLC software versions in the software evolution process. This is important since the rapidly changing production process results in various software versions with evolution process in automation systems, which causes trouble in version control and could raise more problems in future development. To gain an overview and better understand large numbers of PLC software variants, we will concentrate on the architecture of the software systems, evolution visualization, the relationships among different components and the metric-based software visualization for a clear overview of the variation

points and their possible influences, because the visualization of software behavior and code line is not involved in the visual process of the toolkit in this thesis, we will not discuss the current development in software visualization regarding these themes. Before starting to develop the toolkit, the state of the art in these aspects should be researched and introduced in the following.

2.1.2 Structure and Architecture

Software structure and architecture is one of the most important topics in the field of software visualization. The structure and architecture include the hierarchical structure, software components and all related attributes such as the software organization. In this sense, a tree could perfectly fulfill the visual representation of a software structure, with components and limited number of attributes included as tree nodes and hierarchical information as edges in the diagram, on the other hand, trees only visualize structural information and lack textual details such as code and component attributes, which is a disadvantage of this technique. Algorithms and visual methods are developed to represent source code structural information in tree diagram with 2D visualization techniques, which could be further developed for 3D visual effects [14] [15].

Apart from the trees, *Sunburst* charts are also an option to visualize software structure and architecture, *Sunburst* was first proposed by Stasko and Zhang in the year of 2000 [16]. *Sunburst* charts are suitable to display hierarchical structure with rings representing various hierarchical layers. Each ring level of the sunburst chart represents one single hierarchy, which could be divided for different sections of data on the same hierarchical layer. The central circle usually stands for the data root (if exists), with outer layers as sub-branches of the root. The sunburst chart is ideal for visualizing hierarchical data and comparing relative sizes of different elements, they also show great performances in knowledge transfer and software architecture recognition. However, sunburst charts are also limited in the display of complicated structure and great data amounts, which constrains the further possibilities of presenting more detailed information.

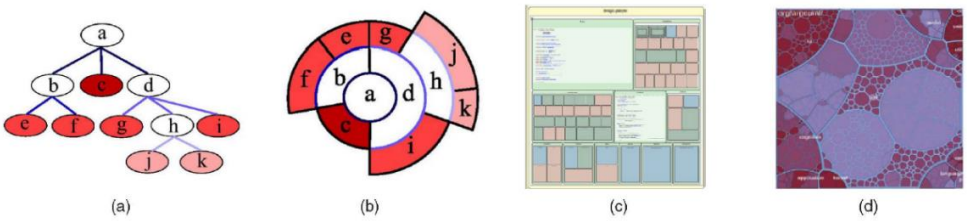

Figure 2 Tree Diagram (a), Sunburst Graph (b), Tree Map in SHriMP (c) and Voroni Treemap (d)
[Caserta and Zendra, 2011]

Tree maps are also common techniques to visualized software structure and architecture, they are graphs to visualize hierarchical data with a set of rectangles and were first introduced by Johnson and Schneiderman [17]. Rectangles of different dimensions are positioned in a way that all elements fill exactly a larger rectangular space, which is the parent component of these elements. Each rectangle represents a data section and could be further divided for tree maps, with rectangle dimensions specify one of the data dimensions. The generation of a tree map is conducted by a recursive algorithm that cut the rectangle into smaller spaces, resulting all elements with separate boxes representing them. Tree maps have great efficiency in terms of space usage and reflect in some way the data dimension and data structure. However, tree maps don't have clear boundaries between elements of different hierarchical level, which

leads to confusion of the ambiguous understanding of the software architecture. Besides, the tree map doesn't provide with much component information, as rectangle are cut into smaller boxes for large number of components, which leads to the reduction of textual information. To minimize of disadvantages of tree maps regarding the structure, the *Voronoi Treemap* is developed to improve the performances of tree maps [18]. The *Voronoi Treemap* introduces more possible shapes to visualize software components such as polygons and allow more possibilities in the visualization. Components in higher hierarchical levels have darker colors while their child elements are blurred to avoid chaotic architecture information. Moreover, the boundaries are also improved as the border between major blocks are highlighted and possess brighter colors, while elements in lower layers only have thinner borders, which largely improves the recognition of different elements in the hierarchical structure. Besides, the Simple Hierarchical Multi-Perspective Tool (*SHriMP*) was introduced to enrich the detail levels of the tree map visualization technique [19]. *SHriMP* is a visualization tool that combines different visualization techniques and represents the visual result in a user-defined frame. For example, *SHriMP* can display a software structure in tree maps, with separate boxes that contains the source code of the selected software component, the text boxes could be manually defined and play different source code, which helps the user to understand details under the macro architecture. *SHriMP* provides a flexible way to visualize software systems without compromising the details in each component, it also offers interactive features zooming, which makes it a good visualization toolkit in the field of software visualization.

2.1.3 Relationship

Relationships are crucial component to understand how a software works, relationships in a software system are inheritances, call dependencies and sometimes access possibilities of different components. To visualize relationships, it is much costlier since the amounts of relationships could be far more than all components together. Relationships reveal the information flow in the software and reduces the efforts to understand the connections between different components. As the concentration lies in the relations, structural information and component attributes are usually not to be found in visualization of this aspect. Network graphs are visual forms that focus on the relationships between different visual components, they consist of nodes and edges, nodes are usually small dots in the graph while edges are curves that connect various items. Network graphs could show different relationships in a software system without much required information. Nevertheless, they could be disorganized if the number of nodes and edges quickly increases, resulting in system resources consumption and challenges in investigation of single relationships.

A simple solution in this dilemma is to transfer the 2D network graphs into 3D space, where the layout could be better managed, and edges have enough space to be well-sorted [20]. Related algorithms and toolkits are proposed, so that 3D network graphs have a better navigation system through different connections and clear visual representation of all relationships. A 3D visualization based on *clustered graph layout* was developed by Balzer and Deussen to display complex software systems [21]. This technique uses clustering characteristic to concentrate individual components that have similarities in terms of structural position and relationships, so that large number of nodes disappear in the graph. Instead, clusters representing groups of nodes are visualized and the relationships could also be described as the connections among different clusters, which greatly simplifies the relationship graphs and form a more abstract but clear structure of the relationships in the software system. Besides, transparent clouds with different colors are introduces to group

different clusters according to the hierarchical relationships, and so the structural information could be partly preserved in this way. The 3D visualization provides a comprehensive overview of the structure and the relationships of its components with good visual effects at the cost of relationship details.

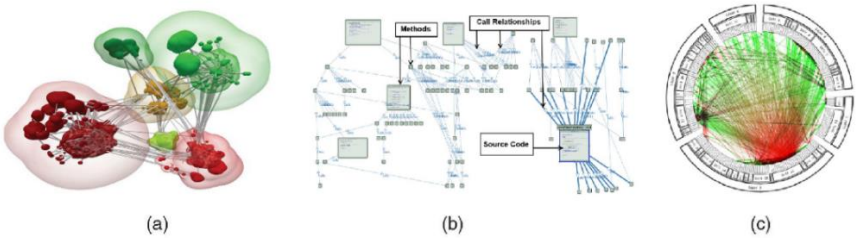

Figure 3 3D clustered graph layout (a), Relationships in SHriMP (b), Hierarchical Edge Bundles (c)
[Caserta and Zendra, 2011]

Alternatively, we have graphs that connect components, which contain much more information such as attributes and characteristics than nodes, such as UML class diagrams. These visualization possibilities offer more details in the components (or classes) than network graphs and contained more information on the functionalities of the edges, but they still cause overload if the software system gets complex with large number of components. To simplify the visualization model on the graphs, a few techniques were used in the visualization with considerations of visual effects and user preferences [6]. The *SHriMP* visualization toolkit was introduced before in the structure and architecture section, the toolkit could also be used to visualize class diagrams with optimized visual outcome for representing relationships. The SHriMP toolkit analyzes the structure and relationships of the software system and visualizes the components with boxes and lines, representing classes (and methods, interfaces) and call relationships, the box sizes could be adjusted to display the source code and other involved attributes. The layout is adjusted to fill the display evenly, in this way, the structure and relationships could reach a balanced compromise.

Another possibility is the hierarchical edge bundles technique that combines the relationships in the software system and the toolkit architecture is to use a ring shape to represent the organization structure and edges in the ring to visualize the relationships between the objects, which was introduces as a feature in the *Extravis* Tool [22]. The ring structure contains all methods (components in lowest level) in the inner circle while the elements in outer ring partially reveals the software architecture. This visualization method emphasizes the call dependencies in the center of the graph with structural information stored in the ring structure, which achieves good visual impacts, but large number of components would cause overload and loss of structural overview, also complicated relationships result in a chaotic overview of the relationships, which impairs the functionality of this visualization technique.

2.1.4 Evolution

Evolution visualization in the software visualization field concentrates the advancements of a software architecture (sometimes relationships) in the dimension of time, it also gains more and more attention as version control systems such as SVN and GIT are becoming standard repository tools. The visualization of software evolution is particularly challenging since the extra dimension of time requires much more data in different development phases. Evolution visualization analyzes information in the development process and offer useful results of the components: which components are continually under development and maintenance, which

functionalities keep growing on size. Such information could be used to improve software modularity and help analyzing the parts that need further development. A stream graph is mostly used in this case, which contains stacked streams as software items, with stream width and its development showing the development process of the elements.

Evolution Spectrograph is a software evolution analysis and visualization tool developed by Wu et al. [23], which analysis the software architecture in the level of components. The software encodes the time dimension in the horizontal axis and the developing process could be easily captured because newly introduced or updated elements are marked with the green color, the colors would gradually turn white as time passes. The software uses the same principle as a stream graph and represents the development process with simple visual elements of boxed and narrow color band between white and green. This saves system resources and accelerates the analysis process of the evolution and the visual scale is proper to be deployed in the complex automation systems. But the toolkit provides no feature to analyze single software component and the visualization lack details such as component name and architectural information, which limits the functionality and usability of the toolkit in further evolution analysis.

Similar visualization techniques with different focus of analysis are also available in visualization frameworks such as the *timeline* visualization technique and *CVSscan* visualization toolkit [24] [25]. The former toolkit focuses on the evolution process of single software component like classes. The visualization uses the code development in different time as the input and analyzes the content, then represent each method with a cube. The stacked bars represent different software versions with the time dimension in direction from left to right, recently modified or new methods will be changed into blue color, then slowly turn back to yellow in next few versions. The *timeline* toolkit offers more useful characters to deal with components instead of the evolution of the whole software structure, it also creates a new perspective to review the constant development of a class file with the time dimension and helps to identify important methods. The class attributes as well as other possible elements in the file are though not included in the visualization, which could cause information loss in the analysis process. The *CVSscan* contrarily, concentrates on the version control functionalities in the aspect of single code files. The toolkit is based on a cushion-based technique and analyzes a multi-version project to generate the visualization in a stream graph. The visualization is version-centered and offers a focus version with the narrowest width which includes no trash code or deleted components. Besides, the toolkit also offers line-based analysis which delivers functionalities in single file such as detection of stable code. This toolkit has useful features and better visual effects, it also takes deletions into consideration. The disadvantage of both techniques is the loss of higher-level abstraction and configuration possibilities.

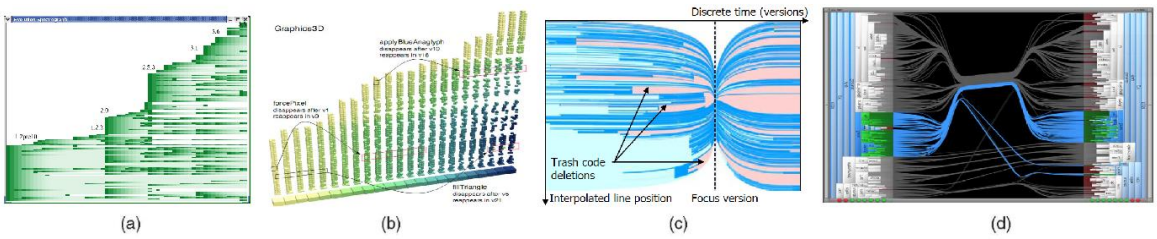

Figure 4 Evolution Spectrograph (a), timeline (b), CVSscan (c), Source code comparison (d) [retrieved and modified from https://hal.inria.fr/inria-00546158v2/document and http://www.cs.rug.nl/~alext/PAPERS/ASCI05/cvsscan]

To compare different versions of a software to determine organizational changes, Holten and Wijk developed a technique for the comparison of software variants [26]. The toolkit shows both versions on two sides of the display, outer elements have higher hierarchical position and inner elements lower, comparable elements are matched and then positioned in same vertical level, then connected with edges, they could be bundled with an edge bundle technique to reduce visual complication. Besides, shade colors define the element in terms of consistency, gray shading indicates that the elements exist in both version while red means that they are only available in current version. The comparison is convenient to investigate differences in different software versions, they could also be used to diagnose errors in new code during development process. Yet the toolkit only supports comparison of two versions and the software construction is not intuitively represented in the graph, which brings problems in understanding the organizational changes.

2.1.5 Metric-based Visualization

Metric-based visualization is a method to evaluate and quantify certain software characteristics and its behavior, it offers information from the view of quality outside the development process. These features allow metric-based analysis and visualization to effectively test and validate complex software systems without efforts to analyze the software architecture. Therefore, it's usually used to describe systems features such as system stability, dependability and complexity. The challenge in metric-based visualization is the visual mapping process from the metrics to the final visualized result [27]. For instance, radar charts are applicable for metric-based visualization, where the values are represented as points on the axis and achieves a simple yet obvious evaluation result.

To quantify various characteristics in one visual graph, a visualization technique called *RelVis* was proposed [28]. This technique uses a radar chart as the basis of the visualization, which could graphically represent numerous metrics together on its axes with lower values nearer from the center point. Values from the software analysis regarding the metrics could generate different values and point positions on each axis, with straight lines connecting the points, a closed surface emerges. The surface area could be used to evaluate the overall behavior pattern or quality index of the software system. An important feature of *RelVis* is that it supports comparisons among different software versions, enabling it to achieve functionalities in software evolution analysis. Different versions and generated surfaces are separated by fill color between the lines, so that the different module values are easier to recognize. The *RelVis* toolkit offers an apparent and intuitive way of describing different key performances or characters of a software, and it also enables user to explore the differences between various versions to analyze features in the development process. It could also be used to explain the relationships of modules by setting the axis value as calls or accesses, but the result is not as easy-understandable as other metric values. Overlapping could also raise problems in the evolution process visualization, which causes information loss.

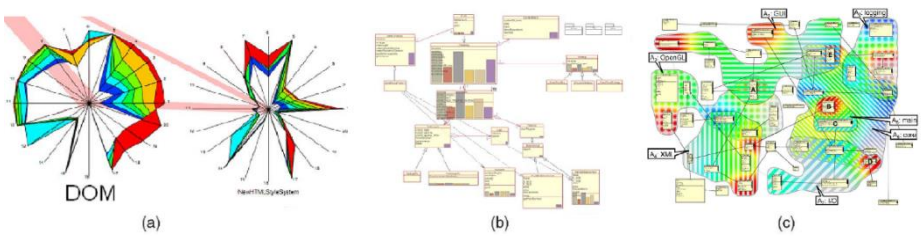

Figure 5 RelVis (a), UML with bar chart (b), UML with heat map (c) [Caserta and Zendra, 2011]

Further visualization possibilities include combinations with other visual graphs. The MetricView was proposed by Termeer et al. [29], which combines the UML elements with pie charts or bar charts to visualize different modules and their metric values. It does the metric-based visualization as simple addition to the UML diagram, but could cause information overload and readability problems in the display. UML diagram could also be combined with heat map to visualize metric values, the method *area of interest* was developed, in which one AOI uses heat map areas that include certain UML components to visualize a metric value [30]. For more metric-based results, several AOIs are also possible at the same time, because they have different filling patterns and create a self-crossing pattern that enables overlapped area is also visible. AOI is a metric-based method that avoids the visual confliction with UML components and relative values could also be easily obtained from the heat map patterns, although the exact value is not directly accessible, and the readability of intercrossed area drops slightly. Metric-based methods could be additionally combined with trees, network graphs, 2D plots or more possible graphs to enrich the visualized information by adding metric-based values as a graphical attribute (such as length or position) of a component.

2.2. Visualization of PLC Software System

In the last subchapter, the current research status on software visualization was discussed and the many different visualization methods were introduced. We will focus on the PLC-oriented software visualization in this subchapter and focus on the state of the PLC application visualization.

Current research and studies on visualization of PLC software systems are very limited, and most of them concentrate on the code visualization of the textual programming languages such as instruction list (IL) in visual forms like flow chart, which is not the topic of the thesis. Apart from the textual programming language visualization, the visualization on the behavior of PLC software systems will be introduced, as behavior is crucial to a PLC program in automation systems. The behavior pattern decides the usability, reliability and efficiency of the automation system; thus, the visualization of the execution process is very helpful to determine the quality of the PLC software. It could also be deployed to simulates possible errors and test the software system regarding its robustness. The software behavior can be visualized in flow charts or tree-like charts which requires no dynamic visual effects but preserves no software architecture, another possible method is the dynamic transition of active software components in the visualization, which is harder to implement but could visualize real-time behavior of the software.

A visualization approach for PLC behavior was proposed by Wirt et al. [31] to analyze the reactive behavior of the program and find possible errors in the PLC software system. The possible execution paths were found out and analyzed upon its transitional behavior and recurring patterns. Afterward, the transition graph, which represents the execution pattern was created based on nodes and edges which is similar to a flow chart. Based on the work, a visualization tool was developed to analyze the behavior from the aspects of code lines, task activity and execution path [32]. This toolkit allows simulation, analysis as well as visualization of PLC software systems and provides plenty useful feature such as task scheduling view and execution path view. The toolkit visualizes PLC software in both code-

based and behavior-based visualization aspects. Differences between two execution could be visualized by background color in the code viewer, values could also be obtained in the code as real-time values returned to the program. Moreover, execution patterns, paths and active tasks could also be directly acquired in the visualization tool. In general, the toolkit offers advanced functionalities and proper visualization concepts in the behavior analysis, it combined many visualization methods and provides powerful features. It could be though further developed for visualizing the relationships and software architecture and the visual effects could also be optimized for behavior analysis.

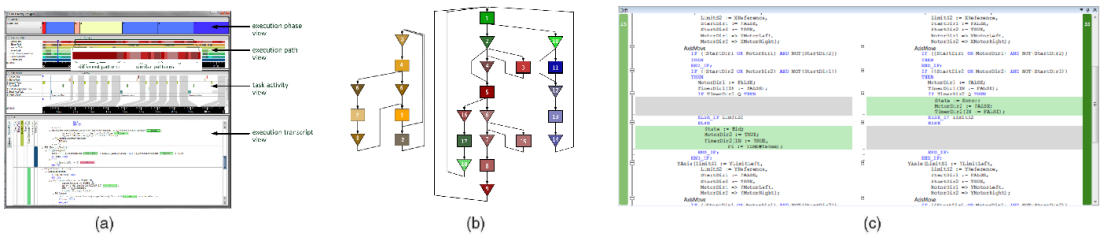

Figure 6 Visualization toolkit (a), Visualized execution (b), Code viewer in different execution path (c) [Wirth et al.]

3. Requirements

Before the development of the visualization toolkit, a few requirements regarding the software and visualization performances should be set to guide the development of the toolkit. Requirements are divided in two subsections: the software requirements and the visual requirements. Software requirements is implementation-oriented and regulates the toolkit performances in terms of functionalities and product quality, so that the software achieve all expected features, is stable and fault-tolerant on the platforms and runs without unexpected failures. As for visual requirements, these are result-oriented and determine the graphical user interface (GUI) as well as the representation of the visualized software systems. Visual requirements set the targets of the visualization and helps guiding the creation and adjustments of the visual styles that are presented as results to the software users.

To achieve the visualization toolkit with full functionalities, while the software runs as expected and generates the wanted visual results, the whole visualization process should be developed under certain requirements, from different phases like data processing, visual mapping to the creation of the software user interface and the testing of the toolkit. They should be the regarded as rules during the implementation phase to guarantee a satisfying outcome after the development. Thus, following requirements in different software components, development phases and quality aspects were made specifically for the implementation of this visualization toolkit.

3.1. Software Development Process

During development of the visualization toolkit, a few requirements regarding the structure of the software and coding process during programming must be followed to create a well-structured and easy-to-understand software system, which could greatly improve the maintainability, reusability and the software quality of the implemented toolkit. These requirements are process-oriented and should be conducted during the whole development for quality and consistency in the toolkit. The requirements on software implementation should ensure the functionalities that were brought up in the last subchapter, on the other hand, they should also ensure that the inner structure and code are well-organized and fully-functional. Thus, the first objective of these requirements is functional requirements tracing, in other words, the functional specification should always be considered during the implementation to guarantee the set goals of the toolkit functionalities.

In this sense, the Waterfall Model is introduced to guide the development process of this toolkit. Waterfall Model is a software development process first used by Herbert D. Benington in the year of 1956 in the development of the software SAGE [33]. It describes the development process with onward project schedule, it contains 5 phases from requirements, implementation to validation and maintenance, which formed a waterfall shape with different layers of abstraction in the vertical direction and time (or development phases) in the horizontal direction. From the first phase requirements to the third phase implementation, the Waterfall Model orientates downward with each phase represents a lower abstraction level in software architecture and requires more specific and detailed information to guide the further process in the development. During implementation, the requirements and design concept are

transformed from natural language into programming language. Afterwards, the Waterfall Model enters the verification process, which checks the software for functionality and performances from components to the software system, then it moves further to the maintenance phase, in which the software is constantly modified and updated to reduces errors and realize more features in its lifecycle.

The Waterfall Model is in this case suitable since the software structure is relatively simple and the functionalities could be realized without many constraints and complicated validation procedure. Also, the Waterfall Model can reduce possible problems in early stages before the implementation process so that many efforts in debugging and testing could be saved. With proper application of the Waterfall Model and unambiguous, detailed requirement definitions, we could focus on the implementation without making mistakes that trace back to the initial phase. In this sense, we apply the Waterfall Model in development of the PLC software visualization toolkit.

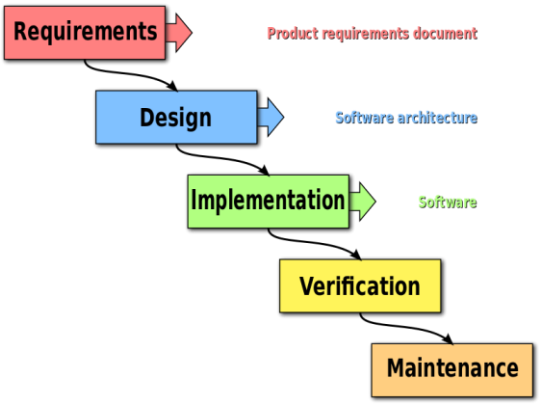

Figure 7 Waterfall Model in Software Development [retrieved from https://en.wikipedia.org/wiki/Waterfall_model]

As the background and motivation of the thesis was explained before, we will then follow the different sequential phases of the Waterfall Model and divide the development process in different phases in the development of the visualization software. We will focus on the requirements on development process of this visualization toolkit here, more detailed and specific requirements are divided into 3 parts in next subchapters: requirements on data input (data processing), requirements on visualization output (functional specification) and requirements on programming standards.

The software development process requirements only segregate the development tasks into multiple phases and regulate the process in these phases to guarantee organized process and properly split workload in each phase. After the requirements are defined, the detailed visualization and functionality design concept will be introduced, which defines how the visualizations exactly look like and what features they should be able to perform to realize different functionalities. Then, the implementation process is carried out from individual visualization components to the integration in the complete toolkit regarding the methods used in the implementation and programming techniques that are important in the realization of the visual effects. After the implementation, this toolkit is tested and verified for the set goals and requirements, we will evaluate the software regarding different criteria such as loading time, we will also conduct a survey for different user groups to investigate the visual effects applied on example PLC software systems, and we could close the verification process with the target attainment of the visualization software regarding visual effects and implementation quality.

However, the maintenance phase of the Waterfall Model will not be discussed in the thesis, because software maintenance is not the focus of the thesis and we couldn't foresee possible maintenance issues shortly after the operation of the visualization toolkit. Instead, the thesis focuses on the conclusion on the visualization toolkit and future work to be done for further features.

3.2. Data Input (Data Processing)

The visualization toolkit should be able to process at least three formats of XML data for the visualization, the supported XML formats are TreeML, GraphML and PLCopen formats, the toolkit can be imported with files in the mentioned formats graphically by the user and automatically process the data, then transform it into internal data structures for further visualization purposes.

TreeML and GraphML formatted files are similar and have features that could be shared [34]. TreeML files are special XML formats that save the structure of a file and relevant information such as name and attributes of a tree in the form of branches und leaves, branch elements possess subcomponents such as further branches or leaves while leave elements are located as the lowest level om the tree. The structure information is directly saved in text as child elements are simply included in the parent element and the size of a single branch element could consequently get very large. TreeML files are created to save the tree structure and the structure is applicable for PLC software systems that share a similar structure. TreeML files contain only necessary information, the structure is also relatively simple and easy-to-process, therefore the TreeML formats provide quick processing time and are mostly used for visualization purposes. GraphML, on the other side, could be split in two parts. One saves all visual components and their information as individual nodes without structural information and the other stores the connections (edges) of different edges as individual items enclosing the source and the target item, so that edges are independent from nodes and have no restrictions on the correlation of elements like in the tree structure, which makes it suitable to store network graphs and complicated dependency relations of a software system. Yet GraphML and TreeML formats are in many ways restricted as a result of its simplicity, as they generally only store important information of the structure, for purposes such as PLC software element modification, these formats are less applicable.

PLCopen XML format is a special XML format created by the PLCopen organization to standardize the saving format of PLC software systems. They are widely used to store and represent the information of the software component in an PLC automation system, PLCopen formats support all different IEC 61131-3 programming languages such as instruction lists (IL), it also saves graphical information like position and size for graphical programming languages such as function block diagrams (FBD). Other relevant information such as layered structure and supplier specific information could also be included in the PLCopen formats. The PLCopen XML formats could literally save all relevant information of a PLC software, which is an important reason that most PLC development software supports the PLCopen format. The PLCopen XML is also extendable to user defined features and files could be checked for its consistency and compatibilities [35]. However, due to various features the PLCopen provides, the PLCopen XML formatted XML files usually have more complicated structure and larger size compared with TreeML or GraphML files. It could also lead to loss

of information during data processing because of the complex structure and the extra information stored in the file. Therefore, the most challenging part of the data processing component is to properly handle the PLCopen formatted files to preserve valuable information for the visualization (e.g. nodes, attributes and code) and abandon redundant data such as position and size of the graphical programming languages.

The toolkit should be able to import the files in the GUI with an obvious button panel and then the toolkit handles the data internally by the data handling component, the processing of data should not miss any structural information or elements and their attributes contained in the files, moreover it should have good processing capabilities regarding the loading time and occupied resources. Afterwards, the processed data will be transformed into an internal object in the program for the visualization process.

3.3. Visualization Output (Functional Specification)

Beside graphical design and visual effects, functionalities in the visualization are also important, since the visualization should deliver more than a graph containing all relevant components. The visualization process should be able to provide more features such as showing details and interactions with user. For the functionalities in visualizing the PLC software systems, the toolkit should be able to perform different actions according to the user demand. An example is the search feature in the visualization, which could be very helpful for engineers in finding repeated software elements or identifying structural position of a function block.

Due to different demands of different users, the following table (Table 1) is created to detail the visualization functionalities and its objectives, followed by an example graph that meet the requirements. The table defines different user scenarios and possible actions / purposes of the user, correspondingly, the required information from the PLC software system is listed and should be treated as input in the visualization. Then, an example of visualization containing required information and features is created to meet the demands of the user, which should be considered as output of the visualization process. Exact visual functionalities are defined in this table, purpose of these utilities in the toolkit is to visualize and help analyzing PLC software systems, which could help users better understand the structure and way of working of automation systems. Functionalities such as code editing and module modification of PLC software systems could be very useful aside from the visualization, and so an extra text editor feature should be considered if needed. Additionally, some important values of the software elements such as reusability or code quality should also be provided in the source data for certain functionalities.

Case	User	Functionality	Required Input	Classification
1	Engineer	Details of Software Element	Involved Element Information (Name, Code, Attributes, Methods etc.)	Elemental Information Acquisition
			Involved Element Properties (Similarity, Group Alignment etc.)	
2	Engineer, Manager	Overview of Software Organization	Involved Organizational Structure	Structural Information Acquisition
			All Element Information (Name, Code, Attributes, Methods etc.)	

3	Engineer, Manager	Information of Software Component Relationships	Involved Organizational Structure	Structural Information Acquisition
			All Element Information (Name, Code, Attributes, Methods etc.)	
			Involved Relationships of the Software System (Dependency etc.)	
4	Engineer, Manager	Overview of Software Family Model	Involved Organizational Structure	Structural Information Acquisition (only family model)
			All Element Information (Name, Code, Attributes, Methods etc.)	
			Involved Element Properties (Similarity, Group Alignment etc.)	
5	Engineer	Comparison of Software Element	Involved Element Information (Name, Code, Attributes, Methods etc.)	Elemental Information Acquisition (Only family model)
			Involved Element Properties (Similarity, Group Alignment etc.)	
6	Engineer, Manager	Overview of Software Element Properties	Involved Organizational Structure	Structural Information Acquisition (only family model)
			All Element Information (Name, Code, Attributes, Methods etc.)	
			Involved Element Properties (Similarity, Group Alignment etc.)	
7	Engineer	Modification of Software Element	All Element Information (Name, Code, Attributes, Methods etc.)	Elemental Modification (only family model)
			Involved Element Properties (Similarity, Group Alignment etc.)	
			Involved Modification Input	
8	Engineer	Modification of Software Structure	Involved Organizational Structure	Structural Modification (only family model)
			All Element Information (Name, Code, Attributes, Methods etc.)	
			All Element Properties (Similarity, Group Alignment etc.)	
			Possible Structural Constraints	
			Involved Modification Input	

Table 1 Requirements of visualization specification

In the table above, certain functionalities and possible purposes are listed with different target users. Also, they are classified as information acquisition and modification with further details on structure or elements such as “Elemental Information Acquisition” or “Elemental Modification” to emphasize the different aspects. Involved Information could be structural information of the PLC software or elemental details of certain components. Element Information include “Element Information” and “Element Properties”, the former concept includes internal element information such as name, code and attributes, while the latter term refers to external definitions such as similarity value in a family model and group alignment (mandatory, alternative or optional). The table sorts all input information generally and assigns requested information to different functionalities, so that the required input for the visualization is available during execution.

Apart from the requirements mentioned in the table above, the visualization process should also fulfill another 3 important features: animation effects, interactive features in the visualization and integrated search function. These 3 features allow users to apply more intuitive methods to interact with the visual outcome and a more effective way to identify objects in the visualization with little efforts. They should be available regardless of the visual

design and different visual forms, with the features, the visualization result will be more accessible and functional as a simple non-interactive graph.

- (**R1**) Animation Effects in the Visualization: the visualization toolkit should be able to perform animated transition if any changes occur in the visualized result, that includes e.g. zooming, highlighting, color changing and other varieties that lead to differences in the visualization. The animation effects should be smooth, have high resolution quality and require only limited system resources. Through animated effects, the visualization should achieve a more intuitive visual effect.

- (**R2**) Interactive Features in the Visualization: the visualization toolkit should be able to generate visualizations that could interact with the user, independent from the visual style and input data. Interaction forms should include at least zooming (zoom in or out the complete display), dragging (change position of a visual item) and hovering (change item color when hovered). Also, selected items should show its path in the software structure in another text box if needed, in case the structural information is not represented in the visualization.

- (**R3**) Integrated Search Function: the visualization toolkit should provide a text in the display that allows user to give input as string to the program, the toolkit should be able to process and search the user input among all visual elements. If matched elements exist, their color should be shown in red. With the integrated search function, the wanted items could easily and precisely stand out and provide satisfactory results to the user.

3.4. Requirements on Programming Standards

Specific requirements on programming (or coding) standards should be considered in the phase of implementation, as implementation in form of coding directly affects the software regarding its structure, maintainability, further development and consequently the quality of the toolkit. The outcome of the toolkit could vary depending on different coding practices, which include documentation, naming conventions, readability etc. Bad coding habits don't directly have consequences in the execution but could largely affect the understanding of the code by other users and indirectly have negative effects on the maintainability, reusability and efficiency on the software project. Some attributes such as modularity and proper algorithms during coding could influence the performances of the outcome and should be kept in mind during the practice. Also, when a project is involving more programmers, the coding style is extremely important as cooperation of a project is only possible, when the code writing could be accessed and understood by others for validation and integration.

As a result, some coding conventions should be treated as requirements during the implementation process to improve the toolkit quality. In general, the following requirements of programming regarding code writing should be implemented:

- (**R1**) Documentation: proper documentation should be added in the code file. Documentation in programming is mostly conducted as comments through the module file. Comments are helpful with the understanding of the code and provide other related information such as author or purpose of the code. Common practice of

documentation includes commenting at the beginning of a code file as well as explanations through the file to separate different steps and their purposes. A brief introduction at the beginning of the code file usually contains the name, author, description and functionality of the module, if needed, additional information such as modifications or licensing information or even work to be done could be included. In case of important calculations or transition between different steps, a short comment explaining the action and its objective should also be added in the line before to improve readability. Good documentation could considerably enhance the code maintainability, usability and extensibility; besides, it makes future work or knowledge transfer much simpler than raw code.

- (**R2**) Naming Convention: naming conventions should be kept during the implementation process. Variable naming in a same style through the whole software implementation is required, so that the variables are easy to recognize and unveil their purposes only with a name. Good naming style could be very helpful with understanding the purpose a variable, the same naming style should also be kept in all modules of the toolkit. Positive examples are iItemColor and fQualityValue, the first letter expresses the data type, followed with two words with each capital letter in the first position, which explains the objective of the variable. Good naming conventions help with understanding important calculations without memorizing each variable, it greatly reduces the efforts to understand a calculation formula and the complete methods in the code during both programming and reading.

Aside from documentation and naming convention, the structure and modularity of the code is also an important attribute to the code quality, which efficiently enhance the reusability and maintainability of the toolkit [36].

- (**R3**) Modularity and Structure: the visualization toolkit should be modularized and well-structured. Good code should be modularized, which means different calculations or important steps should be regarded as separate module instead of integrated in the code. This simplifies the process of further usage of the same function with a simple call and correction procedure of the code if any errors found in the module, as it only needs to be corrected once. Modularity increase the elements in the software structure and could cause trouble when these elements pile up. An optimized structure could classify these elements and create a hierarchical structure that split different elements regarding their functionalities or hierarchical properties. Modularity and Structure could not only helps understanding the modules and their applications, they also improve the maintainability and reusability of the software, because separate modules could help better identify possible bugs, errors and a well-structured program offers clear architecture and description of the toolkit and each module.

The requirements on programming standards is quality-oriented and aims to improve the toolkit in terms of readability, maintainability, reusability and extensibility. Through three individual requirements, the first two requirements should guarantee the code readability and maintainability, while reusability and extensibility are mainly ensured by the third requirement. They ensure the quality of the visualization toolkit and further possibilities in the future development.

4. Design of Visualization Concept

In the last chapter, the requirements on the visualization concept was discussed, different functionalities should be considered when designing the visualization concept. Also, general aesthetic standards as well as visual effects should be considered in the concept phase. To realize the visualization, various visual concepts and corresponding features must be defined before the implementation according to the Waterfall Model [33]. Furthermore, the visual concepts have to be adjusted for PLC software structures to comply with the visualization functionalities and properly demonstrate the properties of different visual elements in the PLC software systems. Important factors in the visualization include e.g. element size (lines), element alignment property (mandatory, alternative or optional) and relationships. To emphasize different aspects of visualized PLC programs, five different visualization concepts specifically for PLC software visualization will be discussed in the following.

4.1. Graph Design

4.1.1 Design of Tree Diagram

Tree diagram is an effective way to represent the structure of a software system. Since a PLC software project is implemented in a hierarchical structure with different Program Organization Units (POUs), which include further functions or function blocks, the tree structure can well reflect the overall dimension of a PLC software. To better adjust this concept to PLC code structure, a modified tree diagram concept is created. This concept owns the same structure as normal tree diagram, connecting elements with hierarchical relations with lines. The root element is located left and its child components on the right, and all elements of the same hierarchical level are vertically aligned. Unlike normal tree diagrams, not all elements are shown in the window, only the selected item and its sub-elements will be expanded. Also, every item will be given a fill color, colors could be used to show different item property, here mostly the element group property (mandatory, alternative, optional) and group affiliation (scenario 1 or scenario 2).

Optionally, the tree diagram could represent the call dependencies in certain hierarchical level with curves on the right side of the graph. However, this could result in confusion of the structure and chaotic visual effects in case of complicated dependencies. In general, the tree diagram concept could help understanding the basic structure of a software project and some basic properties, but for further understanding and features, more advanced and specific visual concepts are necessary. The tree uses text boxes (component name as text) as nodes in the structure and lines as hierarchical bonds between nodes, different node fill colors are designed to outline element properties (such as mandatory). The simple design enables an elegant visual style, but also limits its further functionalities such as comparison and detailed overall description of single elements. The exact design is illustrated in the figure 8, with different color fills in alternative and optional elements, the selected and searched items are also highlighted correspondingly. Moreover, the element size and shape could also represent certain element attributes, but for optimal visualization effects of the tree structure, we disabled this feature for now.

In the tree diagram, the structure could be used to represent not only a PLC software system family model, but also single PLC scenarios in the same way. However, single scenarios in tree diagram will should fewer visual varieties such as color variation, because certain element attributes like alignment property are not applicable with scenarios. So, the tree diagram of single PLC software is basically the simplification of family model tree diagrams. As for functional features, the design concept of the tree diagram allows an overview of the PLC software structure and attributes of software components, specific element properties in family model as well as the interactive features and search function should be provided to the user.

4.1.2 Design of Network Graph

Network graphs are used to model relationships among all software elements. Unlike tree structure, network graphs could graphically display much more complicated object relationships than pure hierarchical relations, which makes it applicable for dependencies of the complete PLC software system. Network graphs also have more available visual options to describe element attribute such as element size, color, grouping. Thus, network graphs could reflect more element details and possible relationships than a tree diagram at the price of PLC program organizational structure. But in case of large amounts of elements in a single frame, a network graph could be very crowded which directly leads to difficulties in picking up information.

In this concept, the network graph could represent either the hierarchical structure (which makes it exactly a tree) or the dependency structure of a family model (or a scenario). The element text represents the element name and different colors show an item property (element group alignment), which is same as the tree diagram. Furthermore, the size of a node can represent certain value or property of the object, which could be the code line quantity, the similarity value or other meaningful values. A surface area could also be optionally introduced in the graph to highlight all elements of the same group alignment, so that all objects of the same group could be surrounded in this surface area, while other elements fade away, forming a sharp contrast for emphasis purpose.

The concept of network graph could be useful to acquire further basic information of a PLC software system than the tree diagram. It offers interaction between the user and the grapy, with which certain objects could be selected and their related elements highlighted. The concept has more details, more advanced features such as configuration possibilities of size or group criterium. But all elements in one display cause trouble finding and understanding the structure, especially with large software systems. Functionalities of a network graph are despite its focus on relationships mostly identical to a tree diagram, which basically includes element details, but the PLC program organizational structure will not be graphically presented, instead the relationships are focused. With the force simulation feature in the visualization concept, dragging of a visual item results in the movement of neighboring elements. If the relationships of an element should be researched, it could be dragged further to frame edge to find connected items. Nevertheless, numerous edges and nodes with overlapping could potentially add to the complexity of the visualized PLC program and cause difficulties in understanding.

4.1.3 Design of Tree Map

A tree map is a set of nested rectangles arranged in specific order. A huge advantage of tree maps is that it makes good use of space. It also represents the comparison of elements with the sizing, bordering and coloring according manually defined criteria. So, the tree map concept is very flexible and has better performance in comparing the objects.

In the tree map concept, every rectangle equals a basic element of the scenario, position and sizing are the most important factor in a tree map and could be defined by the user. The elements from higher hierarchical levels are more detailed and could reflect more characters by organizing corresponding sub components in its rectangular area. For small sized objects and their child elements, names and other attributes will not appear in the frame. Dividing lines are also different for different hierarchical levels in terms of color and thickness. Tree maps don't have particular advantages in PLC visualization, but it produces a visualized structure with clear dividing boundaries and a clear organizational structure of all components. Dividing edges are thicker and brighter for bigger size rectangles (elements of higher hierarchical position, possibly with names displayed). While other software components are generally visualized as boxes with dark fill color, the alternative and optional elements however, have different colors.

Concerning the functionalities of the tree map concept, it is not different from the tree diagram concept. But the focus of a tree map has minor differences in the display of visual items, it points out elements and structure in a less textual way and emphasizes their characteristics by sizing and positioning. The tree map concept could generate and overview without overlapping of information and disorganized visual effects. Additionally, this concept visually produces an element-oriented overview of PLC software system, which can be very helpful in finding out the relevant position of certain elements (such as frequently called objects).

Three basic visualization concepts were discussed, but for more detailed and reasonable visual styles, minor modifications on current visual methods are not good enough. Two whole new visual concepts will be introduced in the next sub-chapter.

4.2. Design of the Ring Graph

The design of the ring graph concept uses some visual techniques in data visualization and combines their advantages to make the graph suitable for PLC software visualization. The ring graph is a new visual concept for PLC visualization consisting of two visual elements, the rings and the edges.

The rings represent the family model or single scenarios, the outer ring is the family model here with all possible objects, meanwhile the inner rings are all scenarios contained in the family model with possibly different composition. Each ring has a color pattern, the color depth could be used to show element property and be defined by the user, components that don't exist in one scenario will simply be shown in gray as missing element. Fat lines divide different sections of the ring, which are grouped by hierarchical structure (or other manually defined criteria). Moreover, the angular width of each element also serves as a property visualizer, which relatively reflects an attribute value, and in most cases, it visualizes the

quantity of code lines in the element. The edges could reflect the relations among elements, mostly call dependencies of the family model. Different line types are also possible in some cases. If needed, this area could also be blank or used for data legends.

Functional features of the ring graph include the overview of a PLC program with its different components, otherwise it also supports the visualization of the relationships among these elements, the overview of a PLC family model and the comparison of all elements regarding certain attribute value. The advantages of the ring graph concept is the combination of structural and relational information of a software system, which are separately processed by the rings and the edges. Through the separation of these two visual components and customizable possibilities of the graph element, the ring graph could achieve numerous combinations of visualized information and fulfill the requirements of information acquisition by most criteria.

The ring graph concept is visually optimized for a clear overview of the structure and objects in a family model, it also brings details of single elements in a subtle way such as color, width and lines. More importantly, most of these visual criteria could be changed by the user to make certain properties stand out. The finding and understanding of information are also simplified with the search function, related items will be shown in red instead of its original color. This visual concept offers an effective way to acquire information on the whole family model and its scenarios regarding a wide range of selection criteria. These criteria are visualized in different aspects of the graph with no interferences. It gets though more problematic with increasing number of elements in one family model, in which the acquisition of object details is more demanding.

4.3. Design of the Bubble Graph

The bubble graph is another innovative design concept for PLC software with more freedom and possibilities to realize the visualization of different PLC software system with individual requirements. It contains a frame of X and Y axes as regarding value of the element position, and positioned bubbles in the frame are visualized to simulate the elements with different X and Y attribute values.

The bubble graph uses bubbles to symbolize the various objects in a PLC family model such as functions and function blocks, the visualization is consequently a collection of many bubbles. Bubbles could possess different sizes and colors, which speak for element values such as similarity and call frequency, and they are positioned in the frame according to the X and Y values. X and Y axes are brought up in this visualization concept, the definition and value interval of X and Y axes could be manually defined, with each definition belonging to one of the possible element attributes contained in the element or the family model. These two properties set the X and Y position of all visual elements in the frame and can be changed any time, and the scaling of the axes are completely automatic so that bubbles are visible in the picture. The bubbles are positioned according to its X and Y attribute values such as code quantity and group affiliation, which are analyzed during the data processing. In addition, a surface could appear if an object is selected, and all elements with the same group affiliation will be included in this surface and form a colored area, a little window will also appear next to the selected object to display relevant element information.

Respecting the functionalities of the bubble graph concept, in should be able to implement an overview of a PLC program with its different components without hierarchical information, it should also visualize the relationships among these elements if needed, the overview of a PLC family model and the comparison of all elements regarding certain attribute value are same guaranteed by the customizability of this graph. The biggest advantages of the bubble graph concept are information to be visualized of a software system, which can be fully defined by the user and reaches great applicability in different user cases. By using the axes as the visualization standard, the graph is defined by all possible values of a visual element, which allows a comparison of all elements in a 2-dimensional space. Unlike normal 2D-plots, the bubble graph adopts bubbles with different sizes and colors not only to offer a better visual impact, but also to include key information such as group alignment. In this way, the bubble graph concept could visualize more related information of a PLC software system than other visual designs.

The bubble graph concept is more customizable and flexible in terms of the visual layout. The elements will be positioned exactly according to the X and Y properties, which can help identifying valuable objects such as reusable elements with high call frequency and code quality. Moreover, the bubble itself and the surface of similar elements and the data label can add extra useful features to this visualization. With the bubble graph, users can sort the elements with desired criteria and quickly find valuable objects. This visualization could also identify elements that are supposed to be further developed or deleted, which could be a huge burden for the update and maintenance process. The disadvantage of the bubble graph stands in the missing structure information, which will have to be compensated with other visual concepts.

5. Implementation of Visualization Concept

To implement the visualization concepts mentioned in the last chapter, the prefuse visualization framework was used to support our visual models, other relevant libraries and toolkits are also included in the chapter. Besides, the implementation process of the visualization is be discussed and a basic pattern of the process is first introduced, followed by the programming details and a short summary in each design concept.

5.1. Visualization Framework

5.1.1 The Prefuse Visualization Toolkit

Prefuse is a software toolkit to help with creating data visualization, it provides a visualization framework for the Java programming language and can be imported in the eclipse IDE. Prefuse supports variable functions for data extraction and processing, visualization as well as interaction. The prefuse visualization Toolkit also provides data structures, graphic layouts, animation and database connectivity (see next page) [37].

The Prefuse Visualization Toolkit was first released in 2006. It was developed by the University of California, Berkley under the BSD license. Prefuse is written in Java and could be accessed for both commercial and non-commercial purposes. It reduces great amount of work in visualization compared with traditional methods, some of its main features are:

- Support different data structures
- Separate components for layout, color, size and shape
- Support interactive and dynamic visualization
- Support animation through scheduling
- Integrated search function
- Support SQL database

The prefuse visualization toolkit is based on the *information visualization reference model*, which could divide the visualization process in a few discrete actions from the raw data to the final view [prefuse-toolkit structure]. First, the collected source data must be formatted in a *data table* constructed internally in the toolkit. Then, a *visual abstraction* is created, which is a data model containing visual features like layout, color, shape etc. The last step of the visualization is the so-called *view transformation*, which transform the *visual abstraction* to a view with rendering components and represent the final visualization. Besides, the prefuse toolkit supports interaction with the user after the view is generated, e.g. dragging or zooming.

In this process, the most important components in the prefuse visualization toolkit are **prefuse.data**, **prefuse.action**, **prefuse.control**, **prefuse.visual**, **prefuse.render** and **prefuse.display**. To build an application with visualization functionality, the loaded data has to be transformed in *data table* in the created **Visualization**, then a **RendererFactory** is created to specify the visual features (color, shape etc.). **Actions** are responsible for the visualization operations which include animation, location and other interaction possibilities. **Controls** could be added to the **Display**, enabling the specific functions such as zoom and search utility. A demo included in the source code of the prefuse visualization toolkit shows how the visualization of a simple network graph looks like.

The prefuse visualization toolkit provides various visual layouts including tree diagram, network graph, tree maps, charts etc. The visual elements and different layouts of the toolkit have great visualization and animation effects. The toolkit has good support in the processing of data, it is structured and well-documented. The default visual style is impressive, and users don't have to spend much time on adjusting the visual details and interactions and can focus on the programming, which is relatively complicated. Consequently, modification or extension of the toolkit are also feasible and need only some efforts finding the right way.

However, some features of visual items (such as node shape) couldn't be directly modified, and the variability of the visual layouts is very limited. Thus, certain changes must be made to achieve the ideal visualization effects for PLC Code. Due to the separate design of visual components (nodes, edges, colors etc.), it is relatively quick to find and modify the source code, the well documented code also helps with understanding and modifying this element. Another disadvantage of the prefuse visualization toolkit stands in its outdated version, the latest stable version was released in 2007, with constant development of the visualization technology, the toolkit is outdated in some way and could result in compatibility problems.

It is generally simple and enjoyable to work with prefuse visualization toolkit, the performance of the visualization, the elegant appearance of the display and adaptability of toolkit provide good applicability in the PLC visualization project. But due to limited (partly old-fashioned) visual styles and outdated code of prefuse, certain amount of efforts to adapt and modify the code is necessary. Most of the visualization in this thesis is also conducted using the prefuse visualization toolkit.

5.1.2 The Java Universal Network/Graph Framework

The Java Universal Network/Graph Framework (JUNG) is a software visualization library which offers a simple way for analyzing and visualizing data in a network graph. The JUNG toolkit was developed by 3 PhD students of the University of California, Irvine under the BSD license [38].

JUNG focuses on the visualization based on nodes, edges, joints etc. (Networks, hypergraph etc.), which represent the relationships among different visual elements. It has two basic graph types, namely graph and hypergraph, both of which have further sub-layouts. The visual elements such as vertex and edges could also be divided into many different types and sub elements.

To create a visualization instance, data must be loaded from a file which contains the basic information of a network graph (usually GraphML and Pajek). A layout of the network is to be defined afterwards, the toolkit provides a few layouts depending on different algorithm

(clustering, distance, equivalence etc.). Then the visualization must be initiated by creating nodes individually and specify edges of the graph, in this process of the visualization is the most important step, as different options of the edges and nodes have direct impact on the final visual effects (node shape, color, text, edge types), painting and labeling are also options to improve the graph quality.

Additionally, JUNG supports interactive graph with **DefaultModalGraphMouse** class, it also supports interactively creating, modifying or deleting nodes directly on the graph with extra class of **EditingModalGraphMouse**. An individual class could also be created to specify different modes interacting with the graph. Positioning, rotating and zooming of the graph is also possible using the **GraphMouse** class built in the toolkit.

JUNG has very powerful features for a network graph, the code that would achieve the visualization is incredibly simple. It also provides advanced function for setting up the details of a graph. But the processing of data and the support of other visual forms are literally not practicable. The toolkit version is also relatively old, with the latest version released in the year of 2010.

5.1.3 The dom4j XML framework (related library)

The dom4j XML framework is an open source library for XML, XPath and XSLT documentation for the Java programming language using the Java Collections Framework and with full support for DOM, SAX and JAXP standards. It provides functions for parsing, creating, editing and transforming XML data, which is widely used in the automation systems and the visualization toolkit presented in this thesis. The library is distributed under an BSD-style license and can be used for commercial and non-commercial purposes [39].

Dom4j library is used as part of the data processing unit during the implementation. As the data input of a PLC program is mostly PLCopen XML formatted files, the visualization has to transform the XML file into data tables, so that the program could start the visual mapping process and generates the final visualization. The data transformation process mainly analyzes the XML file and captures information that is necessary in the visualization process. The information is then transformed into a formatted table and appended to the origin data table in the visualization.

5.2. Visualization Process

The visualization process with the prefuse visualization toolkit is relatively complicated but offers more visual options and good visual effects. The data processing of prefuse supports GraphML- and TreeML-files, with the visual objects and their attributes saved as a data table- Prefuse framework doesn’t support direct edition or mapping of the visual data, thus the data editing after the visualization is hard to achieve and needs constant system resources. Therefore, the thesis processes the data before the visualization process and focuses on the visual adjustments after the visualization.

The general approach of visualization with prefuse goes like the picture below:

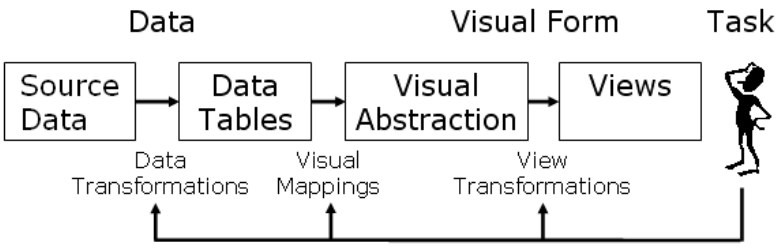

Figure 8 Prefuse visualization model [Retrieved from http://prefuse.org/doc/manual/introduction/structure/]

Data Transformation:

The first step of the visualization is data transformation. Since the PLC code are saved in PLCopen XML format, the structure of the PLC code could be easily transformed into TreeML-file with the structure section of the XML file, selected values of these objects in the PLCopen-file could also be saved in the file as attributes. After this process, a data table is created with the **TreeMLReader, GraphMLReader or AbstractGraphReader** built inside the framework, which could be visualized as network graph or tree diagram without any further data processing.

Alternatively, we could directly create an empty data table and fill data to the table in the software (since the comparison of different PLC code is not directly given in XML format, the data handling has to be processed in the visualization in prefuse), in which we add objects to the empty data table.

Visual Mapping:

After the data table is created, the raw data is transformed to visual objects containing the structural information of the PLC code, they are saved as **VisualItem** in the data table and we could add attributes to the data table before the next visualization step.

The most efficient way is to use an iteration process of the data and add important attributes to the visual object in the table. During the iteration we could also add are a few columns in the data table to save extra values which could be used in the filtering or adjusting process of the visual effects, such as lines, dependencies etc. The iteration goes through all visual objects and add related values to it, and then the step of data transformation is done.

View Transformation:

The view transformation process is the most complicated step in the whole process and handle all the details that are relevant to the final visualization. For fine visual effects, the visual mapping could usually be divided into the following subsections:

1) Render: for different visual groups, various renderer should be created, which defines the details of the corresponding visual groups. Mostly used are the **LabelRenderer, EdgeRenderer and ShapeRenderer**, they are used separately for the nodes (vertexes), edges (lines), and shapes in the visualization. In this process, the rendering type, alignment, shape and other options could be defined in the renderers, after which all renderers are added to a **DefaultRendererfactory** and set the rendering process.

2) Layout: for different visualization layouts such as the Ring Graph and Bubble Graph, a lot of modification and changes should be made to the default layouts. But the prefuse toolkit contains many layouts that could be directly used such as **TreeLayout and ForceDirectedLayout**, which provide optimized layouts for tree diagrams and network graphs and could be further modified if needed. User defined layouts are usually extra saved as Java classes and the thesis will not go into the details of the layouts.

3) Visual Action: this part defines the possible visual effects that could be useful in the visualization such as animation. Common visual **ActionList** includes filtering and animation, in which series single actions such as **FontAction** (defines text font) and Layout are combined, registered and run in the display, they could have different length according to their functions (e.g. Animation has INFINITY as activity duration). Coloring actions also have to be defined and registered, since the color of each element could be different, color actions are usually saved in extra class, where the visual item and its attributes will be analyzed and then given different color.

4) Control Actions: some interactive actions such as dragging items (**DragControl**) and zooming (**ZoomControl**) are available in the prefuse toolkit, they only have to be added to the visualization as **ControlListener.** Other useful control actions are **HoverControl** (acts to hovered item), **PinControl** (pins the display) and **ZoomToFitControl** (fits the graph in display). More advanced functions such as adding and editing nodes in the display are not included in prefuse and must be created by the user.

5) Initialization: now that the most details of the visualization are already defined, we could initialize it with a simple **Visualization**, which is contained in **Display** class. It only has to be integrated in a panel to get the final visual results.

The five steps are only a brief description of the visualization process, more features such as filtering panel or integrated search function are also important for the visualization. They also have to be considered as part of the display if desired, meanwhile the details of above-mentioned actions such as position, movements, font and color are not included in this description. We will discuss these details and other related features in the next subchapter.

5.3. Visualization of Tree Diagram, Network Graph and Tree Map

The tree diagram, network graph and tree map concepts will be discussed here, because these layouts are pre-defined in the prefuse toolkit, which means the visualization only requires proper data processing before visual mapping and details in the view transformation. The rest could be done according to the prefuse visualization model, each with individually designed visual elements and optical effects. Moreover, these three concepts have similar shape, color and font actions, the features of them are also similar. Thus, they will be taken a closer look together in this subchapter.

5.3.1 Tree Diagram

The data transformation of the tree diagram is very simple, we import the data with TreeML file or in the program with the reader in prefuse, then we transform it into a **Tree** file and add it to the display. We could also build an empty data table and add PLC objects as well as important attributes such as size and group property to it.

Now, the tree nodes will share a renderer that defines the shape (round-corner rectangle) and text (name). The coloring actions, which are written in an extra class to realize interactive features, will also be added to the tree nodes. After setting the renderers, we set the layout as **NodeLinkTreeLayout** and start registering actions of filter and animation. The filter action list includes visual actions such as renderer, layout and coloring, while the animation action list only contains animator of tree nodes and related animation effects (slow in slow out, quality control etc.). Afterwards, we add control actions like zooming and panning for interaction with users, and finally, the visualization could be initialized.

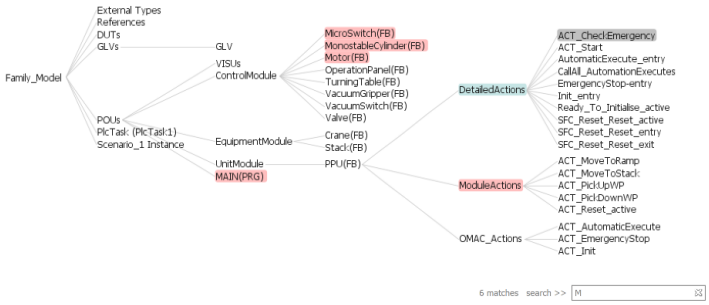

Figure 9 Implemented Tree Diagram

Here, the node colors are white, with selected items in grey and related items in light green. Searched items will be shown in red and fonts of the text are Tahoma. Edges connecting objects with hierarchical relations are in a gray and straight lines. The initialized tree diagram will only show tree hierarchical layers until the user click on any object of the lowest layer in display.

The initialized tree diagram is shown in the figure above. The lower layers in display can be expanded when their parent element is selected, visible objects could also be dragged, zoomed and pinned, while the search function also integrated under the display, creating a sequence of visual items that contains the entered string and causing them to turn into red.

5.3.2 Network Graph

Networks don't have strict rules on object correlation as tree structure, thus a tree diagram could be easily transformed into network, but the reverse process is not always possible. For network graphs, common graph formats such as GraphML and TreeML could be quickly created as the source data of the visualization. The network graph has similar and smaller sized nodes and edges to the tree diagram. Coloring is different with the root element (or the highest layer) always in blue and others grey according to their depth level. Layout is the predefined **ForceDirectedLayout**, with filter and animation action lists that are similar to the tree diagram, only the animation list contains the layout, all elements in the filter action list as well as high quality animators.

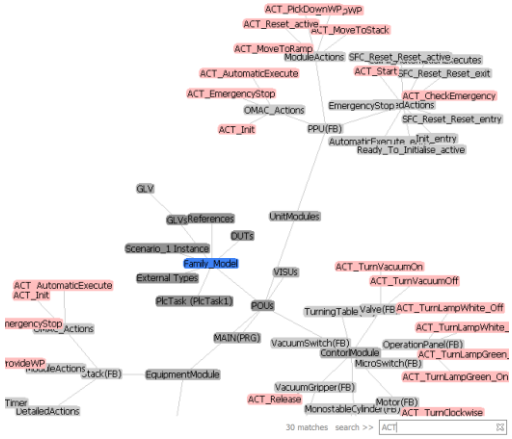

Figure 10 Implemented Network Graph

After initializing the network graph, we have an interactive network showing the structure of the PLC code, the root element marked blue and nodes in gray, searched elements in red. Most importantly, the graph objects could be dragged anywhere in the display, results in forces for connected elements and causing movement of these elements. This process runs very smooth due to the outstanding performance of the force computing, but too many visual items in one display could still lead to lags.

The created network graph is presented above, the hierarchical structure of the PLC program is used in this case. The root element could be easily found, and the related elements are connected through gray edges. Visible objects could also be dragged, which result in force-oriented movements of other connected elements, zooming and panning features are also available in this visualization.

5.3.3 Tree Map

Tree Map is visually different from tree diagram and network graphs, but the data structure is similar. The data structure of a tree map is also hierarchical, with the whole frame as the root elements and the biggest rectangles as its child elements, each rectangle could be divided into further sub-elements representing its child elements.

After importing the data table, the node renderer is created, and the edges can be disabled and set as invisible, since no correlation is shown with lines. The coloring scheme of nodes in the tree map depends on the element depth level, with higher elements lighter and lower elements darker, which also roughly represents the element position in the PLC code structure. An important difference here is the edge color (border color), which is defined by the element hierarchy, and they are brighter if the element has higher positions in the PLC program in for highlight purpose. The layout is **SquarifiedTreeMapLayout** in the prefuse layout class with no further modification. After setting the filtering and animation action lists, the visualization could be started and has a view shown in figure below. Interactive features are mostly not applicable in this concept since visual items in a tree map are once analyzed, visualized and then fixed.

The tree map is visualized as numerous boxes in the frame. Since the number of components is limited and the hierarchical structure of the PLC program is relatively simple, the advantages of a tree map is not fully extended in the case. An overview of the program structure is visualized with all possible components, and it represents the hierarchical position

by color gradients. The search feature could well demonstrate the advantages of a tree map, namely the overview of the software and positioning in a quick search.

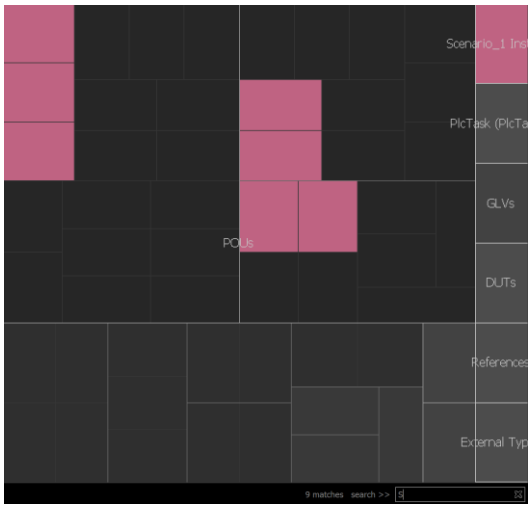

Figure 11 Implemented Tree Map

5.4. Visualization of Ring Graph and Bubble Graph

The ring graph and bubble graph are different from any layouts or combinations that are included in the prefuse toolkit. A simple solution is a visualization containing large number of visual elements could achieve these views. But for consistent and optimal visual effects of all visualizations, the better approach is to create a user defined layout class that contains different features used in the two concepts. And the exact implementation of them will be discussed in this subchapter.

5.4.1 Ring Graph

The ring graph mainly consists of two parts, the rings and the edges. The basic features of connecting edges could be achieved with the **EdgeRenderer** and could represent the correlations of visual items in straight lines or curves, but visual styles and customizability of the edges are limited, so that a customized renderer for edges should be created to realize more visual variations. The other part of the ring graph – the segmented rings are not available in any layout and must be created as a class separately. This class takes a few important features of PLC software systems such as component structure into consideration, it also realizes a ring consisting of various different arcs, which could be individually defined in terms of width, angular width, color and position. In this way, we could manage the ring graph concept with two basic elements, different rings for family model or scenarios and the edges connecting corresponding elements.

In this sense, a **RingLayout** was created as a child class of the **TreeLayout**, since the software organizational structure of a PLC program is comparable to a tree structure. A few important arc element values such as the inner radius, thickness as well as starting angle position are predefined in the class in order to compute the position and size of the arc element, so that a complete, closed ring could be created without overlapping or

mispositioning. The default value used to compute the angular width is the element size and the zero angular position starts from the right side of a vertical line crossing the circle center, followed by other elements rotating clockwise. After defining possible features of a single ring, possible multi rings should be considered in a family model. All rings should have similar cutting positions, the only visual differences lie in the radius and color, so that only small changes should be made to realize the other rings.

Aside from the layout, a child class of **AbstractShapeRenderer** named **ArcRenderer** was also built to define the shape the node element in the form of an arc. This class defines the shape of all possible components in the software structure in the same way. For instance, we need to create an arc shape with inner radius, outer radius, a starting angular position and an ending angular position. The shape could be drawn regarding a circle center point and only the fill color should be defined further. Here a **ColorAction** class could be defined to regulate the fill color of visual elements, an attribute value from the source data could be applied here, for instance similarity value. Every ring has a basic color, which is defined by the RGB color space that applies to elements in this ring. The final color output is however, defined by RGBA color space, which means the transparency of the corresponding color could be manipulated by the attribute value and achieves color shading feature. For elements that don't exist in certain ring, the similarity value equals zero and will be given complete transparency in this case.

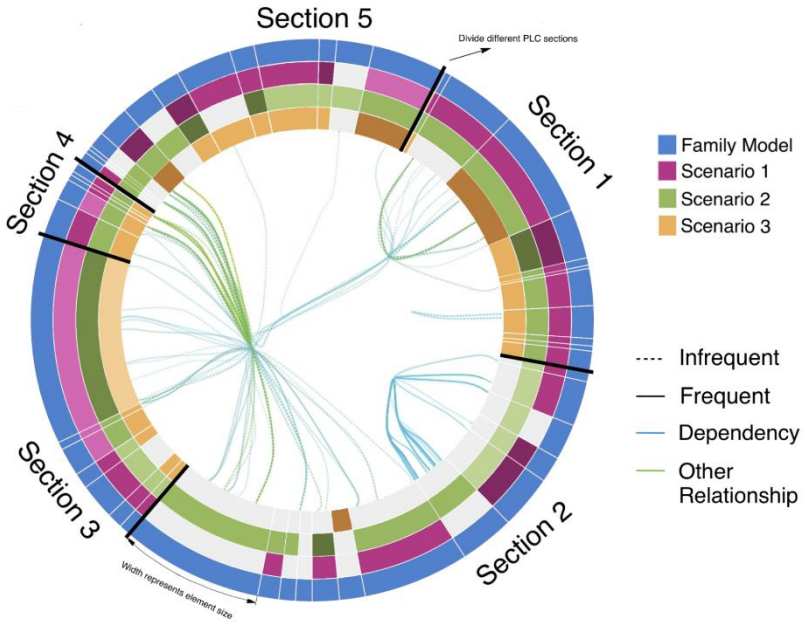

Figure 12 Implemented ring graph

The ring graph has a few problematics regarding the data structure, since the family model is given in a file structure like trees. This input data can't be directly used in the ring graph since the graph has different structure, and so the data processing is different from other visual concepts. A modified tree structure is adopted, with multiple layers of identical elements, each layer represents a single scenario and the root element set invisible in the graph, in this way, the data processing workload could be greatly reduced. The ring graph visualizes all visual elements as arc elements and scenarios as rings, their interconnections are represented by the edges in the middle of the graph. The ring graph offers great variability and an overview of the family model as well as relationships among components, which allow users to have a deeper understanding of the software architecture.

5.4.2 Bubble Graph

The bubble graph contains all elements in the display as colorful bubbles in a plane coordinate system, their size, position and coloring could vary depending on their attributes. These attributes lead to direct visualization of the bubble graph and each can be defined by the user. Thus, the visualization possibilities of a bubble graph could be very different using various dimensions as input of the data table. This feature of bubble graph enables high customizability and much more user options during the visualization process.
The concept basically contains 2 parts: the coordination system and the software elements. The coordination system is responsible for positioning the elements regarding their attribute values, while the components are captured as bubbles defined by its size and color. The combination of them forms a solid comparison of all elements in a 2-dimensional perspective, which could be used to evaluate different element characteristics.

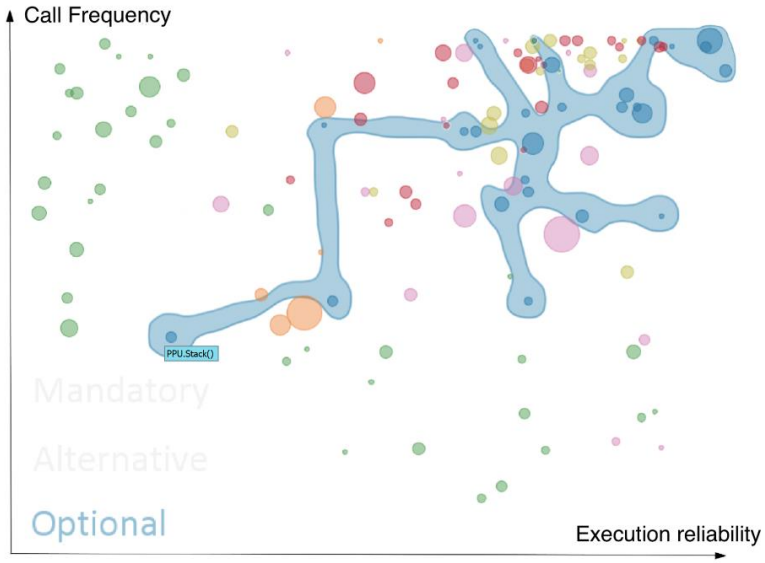

Figure 13 Implemented bubble graph

To achieve this special layout, we only need to create a layout that position the visual items according to the input values (coordinates, size, color). In this sense, a layout called **BubbleLayout** as child class of Layout and a renderer called **BubbleRenderer** as child class of **AbstractShapeRenderer** are created. The layout basically defines the coordination system, which contains value ranges in the vertical direction and horizontal direction, with the user-defined attribute in the two directions, a primitive coordination system is established. The **BubbleRenderer** is then responsible to render the round shape of a software component, it's defined by its size and color, the size dimension could be defined by user, while the color usually stands for mandatory, alternative, optional or its group alignment. With the layout and the renderer extra, we can now follow the visualization process and create a bubble graph by defining the relevant dimensions.

The bubble graph has an extra visual feature of group surfacing. This feature is activated when an element is selected, and the element belongs to an element group (e.g. mandatory elements), then a surface is created and covers all elements in the same group (or with same color). The surface feature is realized by a shape renderer that analyzes all necessary positions, then create an area containing all these points as small as possible. Afterwards, the area border is optimized and rounded with consideration of smooth edges, so that the surface stays thin and round-edged with only few elements visually affected by the area. This feature

allows user to find and locate all similar elements quickly and then conduct a quick comparison among these elements.

The bubble graph could visualize all possible attributes in the data table as an important dimension in the visualization. It allows maximal flexibility and customizability by allowing users to position visual elements in orders they can define. It could be very helpful when one or multiple characteristics of software components should be considered, the visualization offers an intuitive and straightforward result in a coordination system. A major disadvantage of the bubble graph is the loss of structural information and incapability of representing the software architecture.

6. Evaluation

In the last chapter, the implementation process of the visualization concepts was discussed, the implementation process was also conducted under constant review on fulfillment of the requirements. Since the visualized models of PLC software programs are available now, we could briefly evaluate the visualization results regarding different aspects of a software and draw a conclusion on the implemented concepts for this thesis. The evaluation process will be conducted in two parts, the first part includes a survey for possible visualization users such as students, researchers and engineers regarding visual effects, functionalities and other related dimensions. The second part focuses on the behavior of the implemented toolkit, it tests the toolkit on various different PLC programs and retrieves performance results and possible error information to evaluate the robustness of the visualization tool.

6.1. Evaluation

6.1.1 Visualization and Functionality Evaluation

Questionnaires are a common form to evaluate a product and have been widely used in software engineering to assess different software aspects and improve the product accordingly [40]. To evaluate the visualization concepts and implemented tool proposed in this thesis, a survey was conducted for different groups to get feedback and insight from the users. The test participants were shortly introduced to the background knowledge concerning the PLC software systems, software evolution and visualization as well as the purpose of this visualization concept with family model, then they spent about 20 minutes to explore different features and visual effects of 5 demos implemented in the last chapter. These 5 concepts: the tree diagram, network graph, tree map, ring graph and bubble graph will be then separately rated in the survey.

Participants were asked to fill a questionnaire for each demo concerning various aspects of the toolkit (visual effect, functionality and user friendly), participants were supposed to rate these key features of each demo based on a 0 to 10 points scale. Questions in the survey are mostly generalized and are neutrally formulated such as “How likely would you recommend this tool to a friend or family?”, “How satisfied are you with the visualization effects of the tool” and “How user-friendly is our tool interface”. Also, all questions are divided into 3 basic section regarding the aspects mentioned above, an overall grading of a single aspect will be calculated as average result of all related question results. The calculated values in the 3 different aspects will be shown in the survey results.

In the survey, 5 participants in total were involved, 3 from whom without background knowledge of PLC automation system and software visualization and were shortly introduces to the thesis. All participants have fully understood the intention of the implemented tool after the short introduction. Then participants freely explored the software and tried to evaluate the software. After that, the questionnaires were filled, and question answers gathered for final evaluation results. The results of this evaluation are summarized in the figure below as rounded average values from all participants:

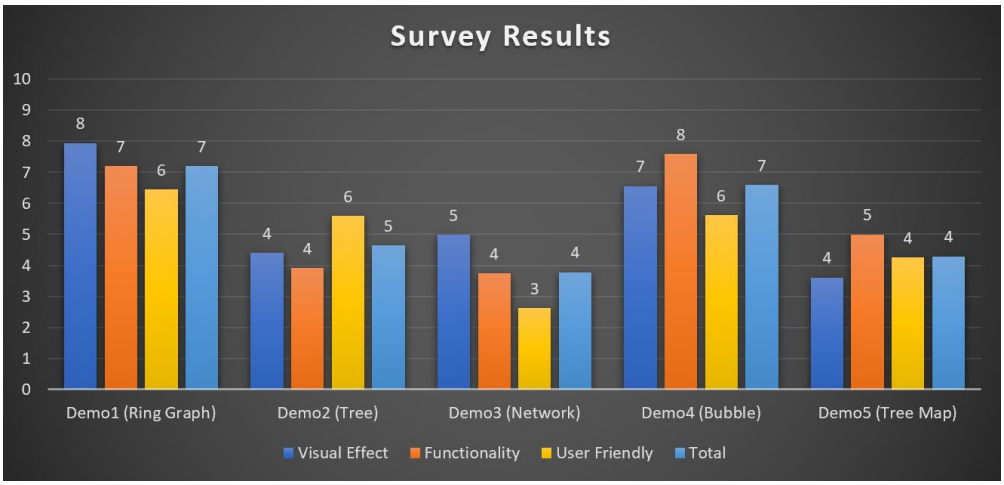

Figure 14 Survey results on visualization

All demos are presented in a random order, and they were separately based on tree diagram, network graph, tree map, ring graph and bubble graph. As we can see from the results, all demos are averagely evaluated in the middle of the rating scale, with visual effects and functionality rated higher than the aspect of user friendly, this could be due to the fact that the user interface of the implemented tool was not carefully designed and resulted in difficulties to start with the tool. Also, the concepts could be compared according to their total result, which generates a ranking (from high to low):

- Ring Graph and Bubble Graph (Total: 7)
- Tree Diagram (Total: 5)
- Network Graph and Tree Map (Total: 4)

Worth mentioning is that the demo 1 (ring graph) and demo 4 (bubble graph) are rated above average in almost all terms. Especially the visual effects and the functionalities are highly rated among all demos. The network graph and tree map have relatively low ratings mainly due to two reasons. On the one hand, all of them lack useful functionalities (such as details of each visual element) or user-defined visualization possibilities. On the other hand, the demos don't possess good user interface features and unique advantages in the visualization that could level up the total rating.

6.1.2 Performance Evaluation

To validate the implemented visualization software, the implemented demos were used to visualize 20 different PLC automation system scenarios, these PLC scenarios were used for different behaviors for a pick and place unit in the Institute of Automation and Information Systems at Technical University of Munich. These files were imported in the visualization program as XML file input in random order, then the program executed data processing and visualized the related information. The execution behavior and other key performance factors of these visualization tools were documented, they are loading time, frame per second (FPS) and warning & error information.

For assessment, the visualization was evaluated only with valid documented parameters: loading time of the display from file import (time gap between file input and display initialization), average framerate (from **getFrameRate()** method in **Display** class, first 1000 framerate entries after display initialization documented and average value accordingly

calculated) and error information during the whole visualization process (in Eclipse IDE console panel), the results of the evaluation are summarized in the figure below:

Table 2 Results on software performance

Demos	Loading Time	FPS	Warning & Error
Tree	0,02798	115,76	1
Network	0,03973	57,59	1
Tree Map	0,04952	7,53	1
Ring	0,03052	14,58	3
Bubble	0,02937	46,45	5

As we could draw from the table, the data loading and transformation process only requires little time. However, in animation and repaint (tree map and ring graph don't contain animation actions and their framerates only rise in case of initialization and user input), due to the large amounts of visual items in bubble graph and bubble graph, the framerates drop rapidly in initialization and animation process, which should be further improved in the future. Besides, all demos have at least 1 warning message, but none quitted during execution, the first warning message was caused by the prefuse I/O reader, beside this warning, the ring graph and bubble graph show a few errors (mostly **NullPointerException**) from time to time, but the program didn't collapse and run further fully functional. Generally, the implemented tools achieved stable performances during the evaluation process, only a few animation problems and program errors should be removed.

7. Conclusion

This thesis starts with an introduction of related fields such as PLC automation systems and the problem definition, then discussed different possibilities of visualization concept to resolve the issue. After the concept design and implementation phase, the visualization tools were tested and evaluated with generally satisfying results. Now, we will analyze the whole process and implemented concepts to draw a conclusion for the thesis, afterwards we will define the future work of the PLC code visualization tools for more functionalities and better user experiences.

7.1. On Visualization

The visualization reaches the most set goals in the design of visual concepts, they basically follow the pattern in the design concepts and deliver appropriate visual effects for the PLC automation systems. Due to development conditions and the restrictions of other projects, some functionalities in the visual requirements are not implemented in this PLC visualization software:

- Fully Support of PLCopen Format
- Editor Feature for Code Modification
- Configurator of Element Modification
- Visualization Concept for Elemental Comparison

Aside from these features in the software, the software toolkit fulfills the requirements for PLC visualization. In some respects, such as interactive qualities and visual effects, the software deliver satisfying results for the user. The main concern in the visualization aspect lies in lack of functionalities such as editing and modification in the graph, which impairs the practicability of the visualized models.

7.2. On Implementation

The software that we implemented functions well and generates visualization with interactive possibilities. With the basic utilities of PLC software visualizetion achieved, the toolkit is to be evaluated after a code review in five general aspects and their relevant requirements, which we defined in the first chapter.

- (**E1**) Data Process: the visualization toolkit can directly process XML files in TreeML, GraphML. For PLCopen formatted data, the software will analyze the data contents and create a data table for the further visualization process. However, files in PLCopen format could lead to loss of information, since not all relevant attributes and information are considered in the data transformation process. Thus, the thesis

basically achieves the set goals for the data process with a few flaws regarding the file parsing and handling.

- (**E2**) Visualization and Functionalities: the visualization in different forms are achieved partly due to the integrated features of the prefuse visualization toolkit. Additionally, we implemented two visual styles based on the prefuse toolkit, but with different layouts and renderers programmed with the Java language. However, functionalities such as code editing and PLC modification are not achieved due to limited time and resources in this project. In general, the visualization of the software achieves the objectives.

- (**E3**) Interaction: the visualization can react to user and fulfill functions such as zooming and dragging. Interactive elements could also be found in the filtering panel and search box, in which the visual effects can be manually adjusted, and certain visual items could be searched. Nonetheless, some useful functionalities like adding or deleting elements in the display are not implemented due to difficulties in data process and display refresh, which could be defined as future work and further developed.

- (**E4**) Maintainability and Reusability: the visualization toolkit can be reused and maintained, the implemented code in Java programming language is also documented and understandable. Most modules are separately implemented and has clear functions and documentation. The prefuse visualization toolkit also provides a well-documented framework which is easy to work on. For these reasons the maintainability and reusability could be positive evaluated.

- (**E5**) Quality: the toolkit is programmed with the prefuse visualization toolkit, which provides a framework including modularized renderer, layout, visual actions and other components. The software deals with most XML files with fair results, but large data could result in slow responds and lags in the visualization. The code quality of prefuse and the implemented visualization software is comparable and offers stable performances in the visualization.

7.3. Future Work

Work to be done in the future primarily includes testing, debugging and further evaluation of the software. Moreover, functionalities regarding two aspects should be further developed, they will be explained in the following points:

- Data Process: for PLCopen formatted data, the software can analyze the data but result in loss of information, because only important objects or attributes are considered in the data transformation process. Also, the PLCopen formatted XML files have a complicated structure and requires specific parsing and data handling elements to preserve all information in the files. Thus, the data process of the PLCopen file remains an important aspect that needs future work.

- Visualization and Functionalities: the visualization in several forms are achieved in the software. But most visual elements, layouts and styles are predefined, which couldn’t be adjusted by the user. Additionally, features such as modification and

creation of files are not available, they could largely increase the functionalities of the software and simplify the process of PLC development. In this way the visualization and its functionalities stay one of the most critical work in the future.

Besides, cooperation with engineers and technical managers in the automation industry could be helpful to improve software quality and usability. Continuous maintenance, updates and new features could also be very helpful for the automation process and meet more requirements in this industry.

List of Figures

Figure 1 Example of a family model [Holthusen et al., 2014] 4
Figure 2 Tree Diagram (a), Sunburst Graph (b), Tree Map in SHriMP (c) and Voroni Treemap (d) [Caserta and Zendra, 2011] 9
Figure 3 3D clustered graph layout (a), Relationships in SHriMP (b), Hierarchical Edge Bundles (c) [Caserta and Zendra, 2011] 11
Figure 4 Evolution Spectrograph (a), timeline (b), CVSscan (c), Source code comparison (d) [retrieved and modified from https://hal.inria.fr/inria-00546158v2/document and http://www.cs.rug.nl/~alext/PAPERS/ASCI05/cvsscan] 12
Figure 5 RelVis (a), UML with bar chart (b), UML with heat map (c) [Caserta and Zendra, 2011] 13
Figure 6 Visualization toolkit (a), Visualized execution (b), Code viewer in different execution path (c) [Wirth et al.] 15
Figure 7 Waterfall Model in Software Development [retrieved from https://en.wikipedia.org/wiki/Waterfall_model] 17
Figure 8 Prefuse visualization model [Retrieved from http://prefuse.org/doc/manual/introduction/structure/] 31
Figure 9 Implemented Tree Diagram 33
Figure 10 Implemented Network Graph 34
Figure 11 Implemented Tree Map 35
Figure 12 Implemented ring graph 36
Figure 13 Implemented bubble graph 37
Figure 14 Survey results on visualization 40

List of Tables

Table 1 Requirements of visualization specification 20
Table 2 Results on software performance 41

References

[1] HOLTHUSEN, S., WILLE, D., LEGAT, C., BEDDIG, S., SCHAEFER, I., VOGEL-HEUSER, B.; *Family Model Mining for Function Block Diagrams in Automation Software;* Technische Universität München, 2014.

[2] BOLTEN, W.; *Programmable Logic Controllers;* (5th Edition).

[3] BENNETT, S.; *A Brief History of Automatic Control;* IEEE Control Systems Magazine, 1996.

[4] PARR, E.A.; *Industrial Control Handbook;* Industrial Press Inc., 2018, ISBN: 978-87-4300-241-3.

[5] ANTONSEN, T.M.; *PLC Controls with Structured Text (ST): IEC 61131-3 and best practice ST programming*; BooksOnDemand, 2018, ISBN 978-87-4300-241-3.

[6] SUN, D., WONG, K.; *On Evaluating the Layout of UML Class Diagrams for Program Comprehension*; Proceedings of 13th International Workshop Program Comprehension, 2005.

[7] WIRTH, C., PRÄHOFER, H., SCHATZ, R.; *A Multi-level Approach for Visualization and Exploration of Reactive Program Behavior*; Johannes Kepler University, 2011.

[8] CASERTA, P., ZENDRA, O.; *Visualization of the Static Aspects of Software: A Survey*; IEEE Transactions on Visualization and Computer Graphics, 2011.

[9] HERMAN, I., MARSHALL, M.; *Graph Visualization and Navigation in Information Visualization: A Survey*; IEEE Transactions on Visualization and Computer Graphics, 2000.

[10] DIEHL, S.; *Software Visualization — Visualizing the Structure, Behavior, and Evolution of Software*; Springer, 2007, ISBN 978-3-540-46504-1.

[11] PRÄHOFER, H., WIRTH, C., BERGER, R.; *Reverse Engineering and Visualization of the Reactive Behavior of PLC Applications*; Johannes Kepler University, 2013.

[12] FENTON, N.; *Software Metrics: A Rigorous Approach*; Chapman & Hall, Ltd., 1991.

[13] EICK, S., STEFFEN, J., SUMNER, E.; *Seesoft—A Tool for Visualizing Line Oriented Software Statistics*; IEEE Transactions on Software Engineering, 1992.

[14] KOSCHKE, R.; *Software Visualization in Software Maintenance, Reverse Engineering, and Re-Engineering: A Research Survey*; Software Maintenance and Evolution: Research and Practice, 2003.

[15] WETHERELL, C., SHANNON, A., SUMNER, E.; *Tidy Drawings of Trees*; IEEE Transactions on Software Engineering, 1979.

[16] STASKO, J., ZHANG, E.; *Focus + Context Display and Navigation Techniques for Enhancing Radial, Space-Filling Hierarchy Visualizations*; IEEE Symposium on Information Visualization, 2000.

[17] JOHNSON, B., SCHNEIDERMAN, B.; *Tree-Maps: A Space-Filling Approach to the Visualization of Hierarchical Information Structures*; Proceedings of the Second IEEE Conference on Visualization, 1991.

[18] BALZER, M., DEUSSEN, O., LEWERENTZ, C.; *Voronoi Treemaps for the Visualization of Software Metrics*; Proceedings of ACM Symposium on Software Visualization, 2005.

[19] STOREY, M., BEST, C., MICHAND, J.; *SHriMP Views: An Interactive Environment for Exploring Java Programs*; Proceedings of Ninth International Workshop Program Comprehension, 2001.

[20] STAPLES, M., BIEMAN, J.; *3D Visualization of Software Structure*; Advances in Computers, 1999.

[21] BALZER, M., DEUSSEN, O.; *Level-of-Detail Visualization of Clustered Graph Layouts*; Proceedings of Asia-Pacific Symposium on Visualization, 2007.

[22] HOLTEN, D.; *Hierarchical Edge Bundles: Visualization of Adjacency Relations in Hierarchical Data*; IEEE Transactions on Visualization and Computer Graphics, 2006.

[23] WU, J., SPITZER, C., HASSAN, A., HOLT, R.; *Evolution spectrographs: Visualizing punctuated change in software evolution*; Proceedings of the 7th International Workshop on Principles of Software Evolution, 2004.

[24] WETTEL, R., LANZA, M.; *Visual Exploration of Large-Scale System Evolution*; Proceedings of the 15th IEEE Working Conference on Reverse Engineering, 2008.

[25] VIONEA, S.L., TELEA, A., VAN WIJK, J.; *A Line Based Visualization of Code Evolution*; Technische Universiteit Eindhoven

[26] HOLTEN, D., VAN WIJK, J.; *Visual Comparison of Hierarchically Organized Data*; Computer Graphics Forum, 2008.

[27] IRWIN, W., CHURCHER, N.; *Object-Oriented Metrics: Precision Tools and Configurable Visualizations*; Proceedings of 9th IEEE International Software Metrics Symposium, 2003.

[28] PINZGER, M., GALL, H., FISCHER, M., LANZA, M.; *Visualizing Multiple Evolution Metrics*; Proceedings of Symposium on Software Visualization, 2005.

[29] TERMEER, M., LANGE, C., TELEA, A., CHAUDRON, M.; *Visual Exploration of Combined Architectural and Metric Information*; Proceedings of Third IEEE International Workshop Visualizing Software for Understanding and Analysis, 2005.

[30] BYELAS, H., TELEA, A.; *Visualizing Metrics on Areas of Interest in Software Architecture Diagrams*; Proceedings of Pacific Visualization Symposium, 2009.

[31] PRÄHOFER, H., WIRTH, C., BERGER, R.; *Reverse Engineering and Visualization of the Reactive Behavior of PLC Applications*; Johannes Kepler University.

[32] WIRTH, C., PRÄHOFER, H., SCHATZ, R.; *A Multi-level Approach for Visualization and Exploration of Reactive Program Behavior*; Johannes Kepler University.

[33] BENINGTON, H.C.; *Production of Large Computer Programs*; IEEE Annals of the History of Computing, IEEE Educational Activities Department.

[34] BRANDES, U., EIGLSPERGER, M., HERMAN, I., HIMSOLT, M., MARSHALL, M.S.; *GraphML Progress Report*; Proceedings of 9th International Symposium on Graph Drawing, Springer-Verlag, 2002.

[35] PLCOPEN; *PLCopen adds independent schemes to IEC 61131-3*; Retrieved from http://www.plcopen.org/pages/tc6_xml/xml_intro/index.htm, October 10, 2018.

[36] SOMMERVILLE, I.; *Software Engineering*; (7th Edition), Pearson, pp. 12–13.

[37] HEER, J., CARD, S.K., LANDAY, J.A.; *prefuse: a toolkit for interactive information visualization*; Proceedings of the SIGCHI conference on Human factors in computing systems, 2005.

[38] MADADHAIN, J., FISHER, D., SMYTH, S., WHITE, S., BOEY, Y.B.; *Analysis and visualization of network data using JUNG*; Journal of Statistical Software, 2005.

[39] DOM4J; *Flexible XML framework for Java*; Retrieved from https://dom4j.github.io/, October 10, 2018.

[40] GROVES, R.M., FOWLER, F. J., COUPER, M.P., LEPKOWSKI, J.M., SINGER, E., TOURANGEAU, R.; *Survey Methodology*; New Jersey: John Wiley & Sons, ISBN 978-1-118-21134-2.

Index

C

clone-and-own 1

D

dom4j 30

F

family model 4

G

GraphML 18

I

IEC 61131-3 1

P

PLCopen 18

Prefuse 28

programmable logic controllers 1

S

Software visualization 4

T

TreeML 18

W

Waterfall Model 16

X

XML 18

Appendix

Appendix A: Code of Tree Diagram

```
package TreeMap;

import java.awt.BorderLayout;
import java.awt.Color;
import java.awt.Dimension;
import java.awt.Font;
import java.awt.Shape;
import java.awt.event.MouseEvent;
import java.awt.geom.Rectangle2D;
import java.util.Iterator;
import javax.swing.BorderFactory;
import javax.swing.Box;
import javax.swing.BoxLayout;
import javax.swing.JComponent;
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.SwingConstants;
import resultView.MainWindow;
import resultView.TextArea;
import prefuse.Display;
import prefuse.Visualization;
import prefuse.action.ActionList;
import prefuse.action.RepaintAction;
import prefuse.action.animate.ColorAnimator;
import prefuse.action.assignment.ColorAction;
import prefuse.action.layout.Layout;
import prefuse.action.layout.graph.SquarifiedTreeMapLayout;
import prefuse.controls.ControlAdapter;
import prefuse.data.Schema;
import prefuse.data.Tree;
import prefuse.data.expression.Predicate;
import prefuse.data.expression.parser.ExpressionParser;
import prefuse.data.io.GraphMLReader;
import prefuse.data.io.TreeMLReader;
import prefuse.data.query.SearchQueryBinding;
import prefuse.render.AbstractShapeRenderer;
import prefuse.render.DefaultRendererFactory;
import prefuse.render.LabelRenderer;
import prefuse.util.ColorLib;
import prefuse.util.ColorMap;
import prefuse.util.FontLib;
import prefuse.util.PrefuseLib;
import prefuse.util.UpdateListener;
import prefuse.util.ui.JFastLabel;
import prefuse.util.ui.JSearchPanel;
import prefuse.util.ui.UILib;
import prefuse.visual.DecoratorItem;
import prefuse.visual.NodeItem;
import prefuse.visual.VisualItem;
import prefuse.visual.VisualTree;
import prefuse.visual.expression.InGroupPredicate;
import prefuse.visual.sort.TreeDepthItemSorter;
```

```
public class MapDemo extends Display {
	private static final long serialVersionUID = 1L;

	public static final String TREE_CHI = "data/scenarios/scenario_test.xml";

    private static final Schema LABEL_SCHEMA = PrefuseLib.getVisualItemSchema();
    static {
        LABEL_SCHEMA.setDefault(VisualItem.INTERACTIVE, false);
        LABEL_SCHEMA.setDefault(VisualItem.TEXTCOLOR, ColorLib.gray(200));
        LABEL_SCHEMA.setDefault(VisualItem.FONT, FontLib.getFont("Tahoma",16));
    }

    private static final String tree = "tree";
    private static final String treeNodes = "tree.nodes";
    private static final String treeEdges = "tree.edges";
    private static final String labels = "labels";

    private SearchQueryBinding searchQ;

    public MapDemo(Tree t, String label) {
        super(new Visualization());

        VisualTree vt = m_vis.addTree(tree, t);
        m_vis.setVisible(treeEdges, null, false); //Function?

        //Ensure that only leaf nodes are interactive
        Predicate noLeaf = (Predicate)ExpressionParser.parse("childcount()>0");
        m_vis.setInteractive(treeNodes, noLeaf, false);

        //Add decorator to top-level nodes
        Predicate labelP = (Predicate)ExpressionParser.parse("treedepth()=1");
        m_vis.addDecorators(labels, treeNodes, labelP, LABEL_SCHEMA);

        //Set up the renderer, for node and for label
        DefaultRendererFactory rf = new DefaultRendererFactory();
        rf.add(new InGroupPredicate(treeNodes), new NodeRenderer());
        rf.add(new InGroupPredicate(labels), new LabelRenderer(label));
        m_vis.setRendererFactory(rf);

        //Node colors: fill and border
        final ColorAction borderColor = new BorderColorAction(treeNodes);
        final ColorAction fillColor = new FillColorAction(treeNodes);

        //Set color action
        ActionList colors = new ActionList();
        colors.add(fillColor);
        colors.add(borderColor);
        m_vis.putAction("colors", colors);

        //Set animate action
        ActionList animatePaint = new ActionList(400);
        animatePaint.add(new ColorAnimator(treeNodes));
        animatePaint.add(new RepaintAction());
        m_vis.putAction("animatePaint", animatePaint);

        //Set the single filtering and layout action list
        ActionList layout = new ActionList();
        layout.add(new SquarifiedTreeMapLayout(tree));
        layout.add(new LabelLayout(labels));
        layout.add(colors);
```

```
        layout.add(new RepaintAction());
        m_vis.putAction("layout", layout);

        //Initialize our display
        setSize(690, 630);
        setItemSorter(new TreeDepthItemSorter());

        addControlListener(new ControlAdapter() {
            public void itemEntered(VisualItem item, MouseEvent e) {
                item.setStrokeColor(borderColor.getColor(item));
                item.getVisualization().repaint();
            }
            public void itemExited(VisualItem item, MouseEvent e) {
                item.setStrokeColor(item.getEndStrokeColor());
                item.getVisualization().repaint();
            }
        });

        searchQ = new SearchQueryBinding(vt.getNodeTable(), label);
        m_vis.addFocusGroup(Visualization.SEARCH_ITEMS, searchQ.getSearchSet());
        searchQ.getPredicate().addExpressionListener(new UpdateListener() {
            public void update(Object src) {
                m_vis.cancel("animatePaint");
                m_vis.run("colors");
                m_vis.run("animatePaint");
            }
        });

        m_vis.run("layout");
    }

    public SearchQueryBinding getSearchQuery() {
        return searchQ;
    }

    //Main function
    public static void main(String argv[]) {
        String infile = TREE_CHI;
        String label = "name";
        if ( argv.length > 1 ) {
            infile = argv[0];
            label = argv[1];
        }
        JComponent treemap = getMapDemo(infile, label);

        JFrame frame = new JFrame("Tree Map");
        frame.setDefaultCloseOperation(JFrame.EXIT_ON_CLOSE);
        frame.setContentPane(treemap);
        frame.pack();
        frame.setVisible(true);
    }

    public static JComponent getMapDemo() {
        return getMapDemo(TREE_CHI, "name");
    }

    public static JComponent getMapDemo(String datafile, final String label) {
        Tree t = null;
        try {
                    t = (Tree)new TreeMLReader().readGraph(datafile);
             } catch (Exception e1) {
```



```
                try {
                        t = (Tree)new GraphMLReader().readGraph(datafile);
                } catch (Exception e2) {
                        e2.printStackTrace();
                }
            }

        final MapDemo treemap = new MapDemo(t, label);

        JSearchPanel search = treemap.getSearchQuery().createSearchPanel();
        search.setShowResultCount(true);
        search.setBorder(BorderFactory.createEmptyBorder(5,5,4,0));
        search.setFont(FontLib.getFont("Tahoma", Font.PLAIN, 11));

        final JFastLabel title = new JFastLabel("                 ");
        title.setPreferredSize(new Dimension(350, 20));
        title.setVerticalAlignment(SwingConstants.BOTTOM);
        title.setBorder(BorderFactory.createEmptyBorder(3,0,0,0));
        title.setFont(FontLib.getFont("Tahoma", Font.PLAIN, 16));

        //Set Text next to search box and function for opening overview of
variable and code
        treemap.addControlListener(new ControlAdapter() {
            public void itemEntered(VisualItem item, MouseEvent e) {
             NodeItem nitem = (NodeItem)item;
             int depth = nitem.getDepth();
             StringBuffer path = new StringBuffer(nitem.getString(label));
             for(int i= 0; i < depth; i++) {
                    nitem = (NodeItem)nitem.getParent();
                    String temp = nitem.getString(label);
                    path.insert(0, temp + "/");
             }
             title.setText(path.toString());
            }
            public void itemExited(VisualItem item, MouseEvent e) {
                title.setText(null);
            }
            public void itemClicked(VisualItem item, MouseEvent e) {
             int clickTimes = e.getClickCount();
             if(clickTimes >= 2) {
                    MainWindow.addVariable(new TextArea(), item.getString(label),
null);
                    MainWindow.addCode(new TextArea(), item.getString(label),
null);
             }
             //TODO Show the code and variable
            }
        });

        Box box = new Box(BoxLayout.X_AXIS);
        box.add(Box.createHorizontalStrut(10));
        box.add(title);
        box.add(Box.createHorizontalGlue());
        box.add(search);
        box.add(Box.createHorizontalStrut(3));

        JPanel panel = new JPanel(new BorderLayout());
        panel.add(treemap, BorderLayout.CENTER);
        panel.add(box, BorderLayout.SOUTH);
        UILib.setColor(panel, Color.BLACK, Color.GRAY);
        return panel;
```

```
    }

    public static class BorderColorAction extends ColorAction {
        public BorderColorAction(String group) {
            super(group, VisualItem.STROKECOLOR);
        }

        public int getColor(VisualItem item) {
            NodeItem nitem = (NodeItem)item;
            int depth = nitem.getDepth();
            if (item.isHover())
                return ColorLib.rgb(99,130,191);
            else if (depth == 1)
                return ColorLib.gray(205);
            else if (depth == 2)
                return ColorLib.gray(155);
            else if (depth == 3)
                return ColorLib.gray(105);
            else
                return ColorLib.gray(55);
        }
    }

    public static class FillColorAction extends ColorAction {
        private ColorMap cmap = new ColorMap(ColorLib.getInterpolatedPalette(10,
ColorLib.rgb(85,85,85), ColorLib.rgb(0,0,0)), 0, 9);

        public FillColorAction(String group) {
            super(group, VisualItem.FILLCOLOR);
        }

        public int getColor(VisualItem item) {
            if (item instanceof NodeItem) {
                NodeItem nitem = (NodeItem)item;
                if (nitem.getChildCount() > 0) {
                    return 0; // no fill for parent nodes
                } else {
                    if (m_vis.isInGroup(item, Visualization.SEARCH_ITEMS))
                        return ColorLib.rgb(191,99,130);
                    else
                        return cmap.getColor(nitem.getDepth());
                }
            } else {
                return cmap.getColor(0);
            }
        }
    }

    public static class LabelLayout extends Layout {
        public LabelLayout(String group) {
            super(group);
        }
        public void run(double frac) {
            Iterator<?> iter = m_vis.items(m_group);
            while ( iter.hasNext() ) {
                DecoratorItem item = (DecoratorItem)iter.next();
                VisualItem node = item.getDecoratedItem();
                Rectangle2D bounds = node.getBounds();
                setX(item, null, bounds.getCenterX());
                setY(item, null, bounds.getCenterY());
            }
```



```
        }
    }

    public static class NodeRenderer extends AbstractShapeRenderer {
        private Rectangle2D m_bounds = new Rectangle2D.Double();

        public NodeRenderer() {
            m_manageBounds = false;
        }
        protected Shape getRawShape(VisualItem item) {
            m_bounds.setRect(item.getBounds());
            return m_bounds;
        }
    }
}
```

Appendix B: Survey Questionnaire Demo 1

Survey PLC Visualization (Demo 1)

Please answer the following questions based on your experience of the Demo 1, feel free to share any comment or suggestions on the demo in the next page. Thank you!

How would you rate the visual layout of the demo?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

Disappointing Exceptional

How would you rate the visual effects (color, shape, animation etc.) in the demo?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

Disappointing Exceptional

How satisfied are you with the demo's overall look and feel?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

Disappointing Very Satisfied

How satisfied are you with the demo's features and abilities?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

Disappointing Very Satisfied

Are there extra functional features you expect from this demo?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

Too Many Very Few

How would you rate the demo's overall functionality?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

Disappointing Exceptional

How user-friendly is the demo software to you?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

Disappointing Very User-Friendly

How satisfied are you with the demo's ease of use?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

Disappointing Very Satisfied

How likely is it for you to recommend this visualization concept to your colleague or family?

☐ 1 ☐ 2 ☐ 3 ☐ 4 ☐ 5 ☐ 6 ☐ 7 ☐ 8 ☐ 9 ☐ 10

Not At All Very Likely

Please share any additional comments or suggestions in this page.

Declaration of Authorship

I, Zhenrui Yue, born on August 14th,1995, hereby declare that this thesis with the title "Development of a Concept for Visualizing Variation Points in industrial PLC Software" is my own unaided work. I have used no sources other than those indicated in the thesis, all direct or indirect sources used are acknowledged as references.

This paper was not previously presented to another faculty or scientific institution in the same or adapted form to attain academic qualifications and has not been published domestically or abroad.

This thesis was developed under the scientific guidance of my supervisor Safa Bougouffa. The main concept and relevant solutions are developed under joint efforts from both sides.

Munich, August 20th, 2018