Multi-Edge Graph Visualizations for Fostering Software Comprehension

Vom Fachbereich Informatik
der Technischen Universität Kaiserslautern
zur Verleihung des akademischen Grades

Doktor-Ingenieur (Dr.-Ing.)

genehmigte Dissertation
von

Ala Ali Hamel Abuthawabeh

Datum der wissenschaftlichen Aussprache: 22. Januar 2016

D 386

Acknowledgements

I want to express my appreciation to my supervisor Prof. Hans Hagen, the head of IRTG group, for providing his support to perform this work. I would like to thank the DFG's International Research Training Group (IRTG) for the funding scholarship at the University of Kaiserslautern (http://www.irtg.uni-kl.de). I am also very grateful to Dr. Dirk Zeckzer for guiding me during my doctoral study over the years to accomplish this work and for sharing expertise. I appreciate his positive feedback and constructive suggestions to develop and to enhance this work. In addition, I want to thank him for proofreading my dissertation and his fruitful comments. I would like to acknowledge Prof. Stephan Diehl, Dr. Fabian Beck, and Mr. Barbaros Can Konar for their cooperation in the user study (Chapter 5). I am thankful to Prof. Gerik Scheuermann for letting me stay in his department in Leipzig University last two years. My thanks are also extended to Mrs. Yasmin Al-Zokari for introducing me to the group of computer graphics and HCI in the University of Kaiserslautern. I greatly appreciate her help and support. Many thanks also go to my colleagues and friends within the University of Kaiserslautern for all the good times we've had. I thank the computer graphics and HCI group for offering many scientific activities and giving the opportunity to meet many experts from around the world. Furthermore, I want to thank International School for Graduate Studies (ISGS), International Club (IntClub), and Muslimische Studierenden Gruppe Kaiserslautern (MSG) for their activities and events in the University of Kaiserslautern. I am deeply indebted to my parents for their encouragements. Above all, I wholeheartedly thank Allah for everything. Danke schön für alles.

Abstract

Typically software engineers implement their software according to the design of the software structure. Relations between classes and interfaces such as method-call relations and inheritance relations are essential parts of a software structure. Accordingly, analyzing several types of relations will benefit the static analysis process of the software structure. The tasks of this analysis include but not limited to: understanding of (legacy) software, checking guidelines, improving product lines, finding structure, or re-engineering of existing software. Graphs with multi-type edges are possible representation for these relations considering them as edges, while nodes represent classes and interfaces of software. Then, this multiple type edges graph can be mapped to visualizations. However, the visualizations should deal with the multiplicity of relations types and scalability, and they should enable the software engineers to recognize visual patterns at the same time.

To advance the usage of visualizations for analyzing the static structure of software systems, I tracked different development phases of the interactive multi-matrix visualization (IMMV) showing an extended user study at the end. Visual structures were determined and classified systematically using IMMV compared to PNLV in the extended user study as four categories: *High degree, Within-package edges, Cross-package edges, No edges.* In addition to these structures that were found in these handy tools, other structures that look interesting for software engineers such as cycles and hierarchical structures need additional visualizations to display them and to investigate them. Therefore, an extended approach for graph layout was presented that improves the quality of the decomposition and the drawing of directed graphs according to their topology based on rigorous definitions. The extension involves describing and analyzing the algorithms for decomposition and drawing in detail giving polynomial time complexity and space complexity. Finally, I handled visualizing graphs with multi-type edges using small-multiples, where each tile is dedicated to one edge-type utilizing the topological graph layout to highlight non-trivial cycles, trees, and DAGs for showing and analyzing the static structure of software. Finally, I applied this approach to four software systems to show its usefulness.

Contents

Chapter 1

Introduction

With the rapid development of computer hardware and software systems in the last decade, a large amount of information immerses the experts of many application domains. To benefit from this information, the experts need advanced analysis tools that support them in performing their tasks. In software engineering, the software structure keeps growing. Therefore, managing and analyzing the ever increasing amount of related information gets more and more difficult. To understand, develop, and maintain software systems, analyzing the static structure of these software systems is one very important task. This task is supported by tools that represent this software structure visually. Usually there are differences between the real structure of the implementation and the designed one. To examine if the real structure of the implementation conforms to the designed one, the real structure is induced from the source code and visualization is used to show the structure making it possible to find deviations and to resolve them.

Approches for displaying software architectures were described in the Software Visualization book of Diehl [42] who defines *software visualization* as "the visualization of artifacts related to software and its development process". Moreover, a classification for visualization approaches that display software static characteristics was provided by the survey of Caserta et al. [40]. The static software structure is commonly modeled using graphs with nodes representing artifacts such as classes or interfaces, and with edges representing relations such as method-call and inheritance. Taking into consideration several types of relations such as inheritance and method-call will support the software engineers in their analysis tasks. Therefore, I focus in this thesis on graphs that represent the static structure of software. Especially, graphs are considered whose nodes are packages, classes, interfaces, and whose edges can be of multiple different types.

Although visualization can show a considerable amount of information, issues such as diversity of relation types and scalability need to be addressed when visualizing the software structure for analysis. This motivates me to enhance the essential visualizations that can deal with these issues. For example, the initial design study of Zeckzer [101] and the approach of Abuthawabeh [7] visualized this kind of graph using a multi-matrix visualization. The visualization presented different visual structures that can help software engineers in their analysis tasks. In addition, node-link visualizations were employed for visualizing this kind of graph in the literature showing different visual structures. Determining the visual structures in current visualizations and how they can help software engineers in their analysis tasks still needs more exploration as few attempts were made in this direction. Showing additional structures that may not be determined using the current visualizations requires introducing new visualizations considering the previously raised issues.

In this thesis, I follow an exploartive approach similar the one followed by van Ham in his thesis [53] where he explored representative solutions considering graph data and general application areas. However, I focus on specific graphs, namely those with multiple edge types and I target the software engineering application area.

The goals of this thesis are:

1. Supporting the analysis of software structure by providing advanced visual analysis tools
2. Expanding the use of visualizations of multiple type relations for analyzing the static structure of software

The research question addressed in this thesis is stated as follows: *"Given a multiple edge types graph modeling several types of relations between classes and interfaces of the static structure of software, how to advance the usages of visualizations of multiple type relations for analyzing the static structure of software?"*

The main contributions of this thesis are:

- enhancing the interactive multi-matrix visualization (IMMV)
- presenting an extended user study that determines the visual patterns for this approach in comparison to the approach of Beck et al. [26]
- presenting algorithms for the topological decomposition of directed graphs together with a complexity analysis of these algorithms
- presenting algorithms for drawing directed graphs based on this topological decomposition
- showing additional four use cases using the small-multiples node-link approach (SMNLV) for visualizing the relations of the software structure based on their topological drawing

This thesis consists of four parts: Part I discusses different development phases of the interactive multi-matrix visualization (IMMV) showing an extended user study at the end (Chapter 5) for determining visual patterns using this tool. Appendix A describes two pretests that were performed in this study. The new approach for visualizing graphs with multiple edges, is based on a new graph layout for the node-link representation of graphs for the sake of displaying additional visual structures that are interesting for software engineers. The new graph layout is presented in Part II. It improves the decomposition and drawing process for achieving an optimal topological visualization of directed graphs. Combining this graph layout with small-multiples provides the small-multiples node-link visualization SMNLV presented in Part III. Finally, Part IV concludes the thesis. Figure 1.1 shows an overview of this thesis.

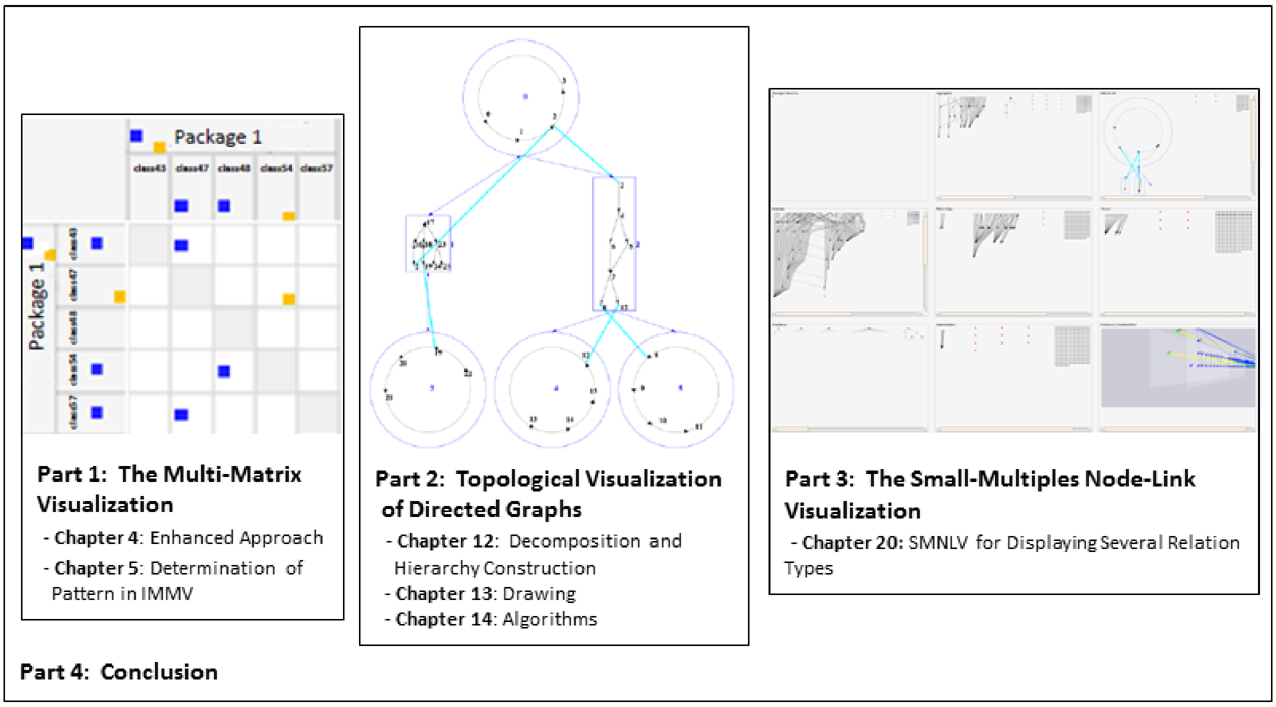

Package 1
Package 1
Part 1: The Multi-Matrix Visualization
- Chapter 4: Enhanced Approach
- Chapter 5: Determination of Pattern in IMMV
Part 2: Topological Visualization of Directed Graphs
- Chapter 12: Decomposition and Hierarchy Construction
- Chapter 13: Drawing
- Chapter 14: Algorithms
Part 3: The Small-Multiples Node-Link Visualization
- Chapter 20: SMNLV for Displaying Several Relation Types
Part 4: Conclusion

Figure 1.1: Thesis Overview

Part I

The Multi-Matrix Visualization

Chapter 2

Introduction

This part discusses different development phases of the interactive multi-matrix visualization (IMMV) showing an extended user study at the end of Chapter 5 for determining visual patterns using this tool. The enhanced approach, called IMMV describes a tool that was built by Abuthawabeh and Zeckzer [9] based on the initial design study of Zeckzer [101] and the approach of Abuthawabeh [7] that added visual elements and interaction. These initial approaches are described in Chapter 3 and form the basis of the enhanced approach (Chapter 4), which is in turn the basis for the pattern determination (Chapter 5).

Improving the quality of software systems requires analyzing their structures. For analyzing the structure of software system, this structure is modeled as a graph by considering classes (and interfaces) as nodes and relations between classes as edges. Considering several types of relations such as inheritance and method-call and different types of nodes such as classes and interfaces contribute to the analysis process. Using visual representations of the graph for the analysis, different issues such as multiple different types of relations and scalability should be considered beforehand. The initial approach by Zeckzer [101] introduced the multi-matrix visualization (Chapter 3), which can represent multiple relations types of software systems in a compact way allows visually comparing the relations while being scalable to large systems.

After this proof of concept, Abuthawabeh developed a prototype adding interaction and further visual elements to the multi-matrix visualization [7] (Chapter 3). To foster the analysis of the static structure of software systems, the IMMV tool was derived from the two approaches afterward as an extended version of Abuthawabeh [7] enhancing it with additional features including source code modifications, enhancements of the visualization, and enhancement of the interaction (Chapter 4).

In addition to the multi-matrix approach of Zeckzer [101], graphs with multiple edge types were represented using the parallel node-link visualization (PNLV) of Beck et al. [26]. Using IMMV and PNLV for visually analyzing and comparing relations between classes (and interfaces), different visual patterns can be found. This leads to the questions, which pattern can be found, what do the pattern represent, and how can the pattern be used by software engineers, e.g., to understand or to improve the structure of software systems. In order to determine the visual patterns that can be shown using IMMV in contrast with the parallel node-link visualization (PNLV) [26], an extended version of the explorative user study [8] is presented in Chapter 5. PNLV and IMMV have been used to visualize these relationships for several software systems. A counterbalanced within-subject design was used to evaluate these two visualizations.

Chapter 3

Initial Approaches

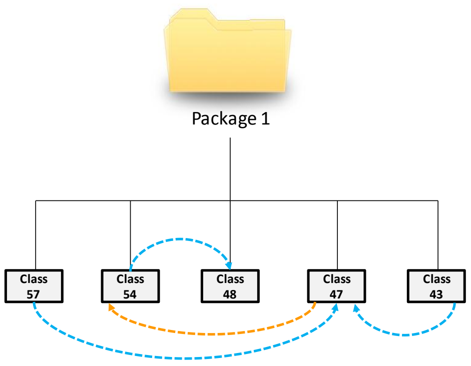
3.1 Introduction

Figuring out the structure of software systems is essential for determining its quality. *"The entities of these systems–classes and interfaces–have many different types of relations. Therefore, discerning multiple types of edges is important for analyzing the graph having the entities as nodes and the relations as edges."* [9] In this chapter, the two multi-matrix visualization approaches that were proposed by Dirk Zeckzer [101] and by Ala Abuthawabeh [7] are described following the descriptions given in [8, 9]. In these approaches, matrix visualization is applied to the software engineering domain to support analyzing multiple types of relations between the classes and interfaces of a system. These classes and interfaces have *"relations such as inheritance, aggregation, implementation, method-call, parameter, return-Type, and throws."* These *"are complementary to each other, revealing multiple aspects of the same software system. Hence, approaches for analyzing graphs benefit from discerning instead of aggregating different types of edges."* [8]

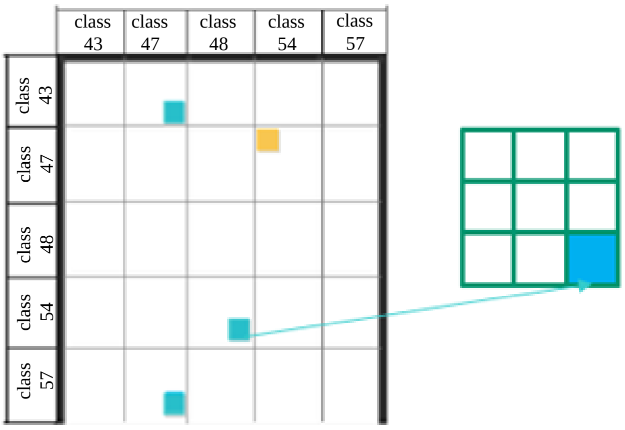
3.2 Multi-Matrix Visualization First Approach

In the first multi-matrix visualization (MMV) approach [101] as shown in Figure 3.1(b), Dirk Zeckzer *"visualizes the classes and interfaces of the software system and the relations between them using an adjacency matrix representation of the underlying graph."* [9] *"Every cell of the matrix is divided into sub-cells, each sub-cell representing a different edge type. A colored sub-cell appears in the matrix if an edge of the respective type from the vertex in the current row to the vertex in the current column exits. To be able to distinguish the types more clearly, a different color is used for each type."* [9] A simple color legend is shown in Table 3.1. *"As an illustrating example, Figure 3.1 presents a small sample software system consisting of one package that includes five classes. The classes are connected by two types of coupling: inheritance and method-call. The system is modeled as a graph: the classes (and interfaces) of the system form the vertices and the different types of couplings are mapped to different types of edges. Figure 3.1 shows two variants of visualizing this sample dataset: Figure 3.1(a) depicts a standard node-link representation and Figure 3.1(b) represents the dataset in MMV."* [8] With regard to visualizing the package hierarchy of the classes, he used and explored three ideas for that. These ideas are:

- using line styles
- using line thickness
- using the border of the matrix to add the full name of packages or to add bar charts and using the length of the bar to represent the depth of the class in the hierarchy



(a) Standard node-link representation adapted from [8]



(b) The MMV [7]

Figure 3.1: An example of MMV showing different relations represented as edges; yellow: inheritance relation; cyan: method-call relation.

Relation	Row	Column	Color
Parameter	1	1	yellow
Return-Type	1	2	red
Throws	1	3	purple
Method-Call	2	1	blue
Aggregation	2	3	brown
Implementation	3	2	green
Inheritance	3	3	cyan

Table 3.1: The color legend [101]

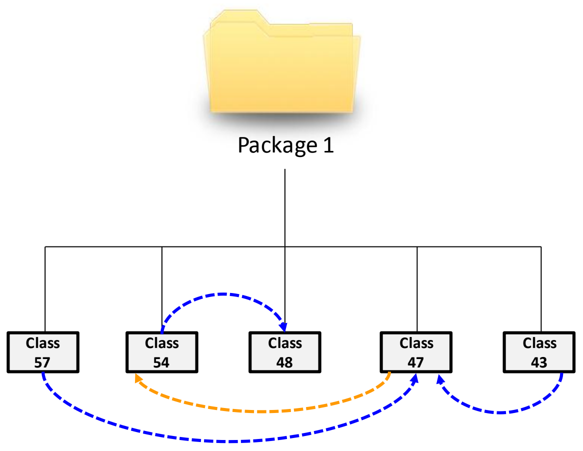
3.3 Multi-Matrix Visualization Second Approach

In the second multi-matrix visualization (MMV) approach [7] as shown in Figure 3.2(b), Ala Abuthawabeh extended the matrix visualization approach of Dirk Zeckzer [101] for increasing scalability in visually discerning multi-type edges in graphs.

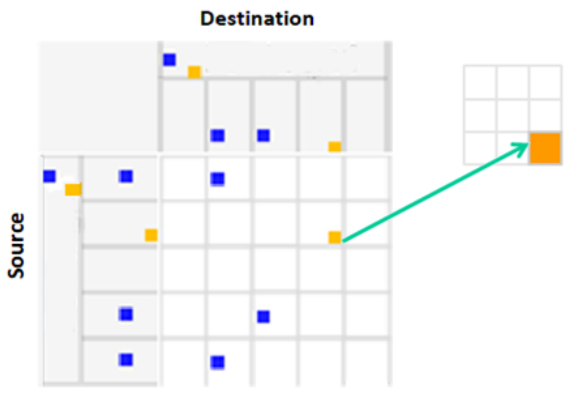
In the visualization part of (MMV) [7] as shown in Figure 3.2(b), *" two icicle plots are attached to the left (source) and the top (destination) of the matrix visualization to display the package hierarchies."* [9] *"Relation summaries were added to the icicle plots to provide an overview over the relations. The relation summaries of the rows are placed inside the leaf cells of the left icicle plot, while the relation summaries of the columns are placed inside the leaf cells of the top icicle plot. The cells representing the packages contain relation summaries of their children."* [9] A simple color legend (see Figure 3.2(c)) was placed on the right side of the matrix.

As a demonstrating example, Figure 3.2(a) depicts a standard node-link representation and Figure 3.2(b) represents the dataset in MMV using the same example graph as in Figure 3.1. The new color legend is shown in Figure 3.2(c). The interaction was enhanced. The names of packages, classes, and interfaces can be retrieved on demand as tooltips when selecting an element of the icicle plot (Figure 3.3).

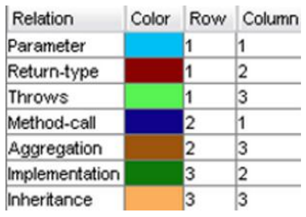
"Scanning selected rows, columns and cells can be performed easily using highlighting. A row in the matrix visualization is highlighted, if a class/interface or a package in the left icicle plot is selected. All sides of the selected cell of a class/interface or a package in the left icicle plot and the row in the matrix visualization are painted with black. A column in the matrix visualization



(a) Standard node-link representation [8]



(b) The MMV adapted from [8]



Relation	Color	Row	Column
Parameter	🟦	1	1
Return-type	🟥	1	2
Throws	🟩	1	3
Method-call	🟦	2	1
Aggregation	🟫	2	3
Implementation	🟩	3	2
Inheritance	🟧	3	3

(c) The color legend [7]

Figure 3.2: Demonstrating example for visualizing a software system with several types of relations formed as a graph with multi-type edges; orange: inheritance relations; blue: method-call relations.

is highlighted analogously. Both row and column are highlighted with a black border, if a cell in the matrix visualization is selected." [9]

The highlighting is also reflected on the coordinate tree view in the same way. Scrolling is also possible when the matrix is large. Finally, folding and unfolding interactions are used to interact with the package hierarchies and the matrix visualization. Folding implies aggregating all relations in the matrix subcells after selecting a package in the icicle plot, while unfolding restores the relations. To discern the cell of a folded package from the unfolded packages' cells, the color saturation of the cell in the icicle plot is changed (Figure 3.4).

Definition 1. *Let $G = \{N, E\}$ be a graph with a set of nodes N and a set of edges E. Let $n = |N|$ denote the number of nodes and let $e = |E|$ denote the number of edges. Further, let d be the depth of the package hierarchy and c be the number of edge types.*

The time and space complexity of the folding and unfolding algorithms is given in Table 3.2.

Algorithm	Time complexity	Space complexity
Folding algorithm	$O(d \cdot c \cdot n^2)$	$O(c \cdot n^2)$
Unfolding algorithm	$O(d^2 \cdot c \cdot n^2 + d \cdot e)$	$O(c \cdot n^2 + e)$

Table 3.2: Time and space complexity analysis for the folding and unfolding algorithms [7]

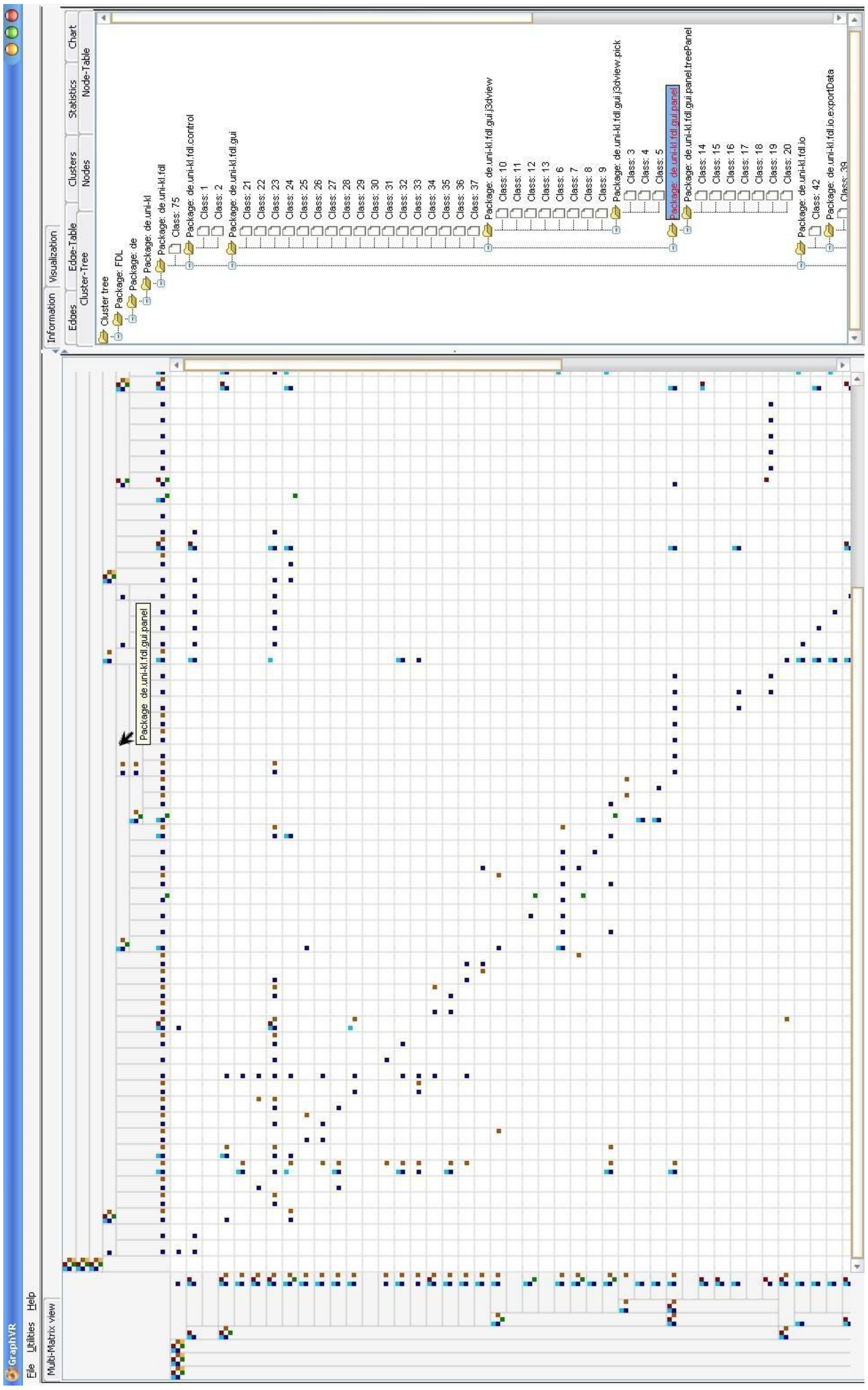
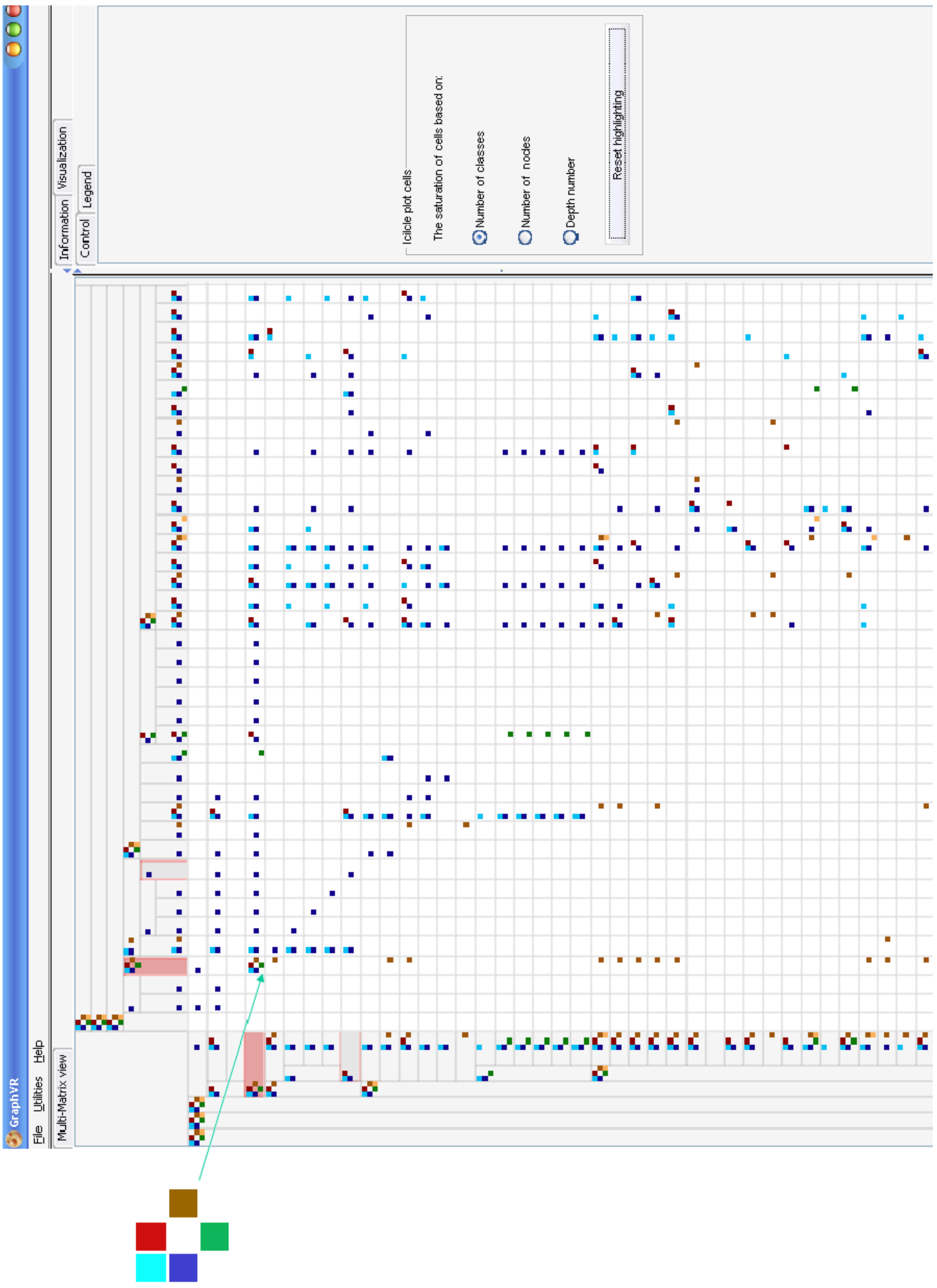


Figure 3.3: Showing the names associated with packages, classes, and interfaces in icicle plots utilizing tooltips [7]



Icicle plot cells

The saturation of cells based on:

[✓ Number of classes] Number of nodes Depth number

Figure 3.4: Using the color saturation to discern the cell of the collapsed package from the expanded packages' cells. [7]

Chapter 4

Enhanced Approach

4.1 Introduction

The initial approaches were applied to different software systems, presented at conferences, and discussed with other researchers. Thereby, several useful extensions were identified. This resulted in an enhanced version of MMV called IMMV, with the following modifications and enhancements compared to MMV:

- MMV's Source Code modifications
  - Importing and parsing graphml files (Section 4.2.1)
  - Compressing tree paths (Section 4.2.2)
  - Using a bipartite graph for the underlying data structure (Section 4.2.3)
  - Involving additional relation types (Section 4.2.4)
- Enhancements of the Visualization
  - Showing labels partially in the icicle plots (Section 4.3.1)
  - Improving the color legend (Section 4.3.2)
  - Showing levels of icicle plots (Section 4.3.3)
- Enhancement of the Interaction
  - Additional Folding and Unfolding Options (Section 4.4.1)
  - Opening data source of classes (Section 4.4.2)
  - Interacting with the enhanced color legend (Section 4.4.3)

First, I give two definitions.

Definition 2. *A Bipartite Graph is "a graph whose vertex-set V can be portioned into two subsets U and W, such that each edge of G has one endpoint in U and one endpoint in W. The pair U, W is called a (vertex) bipartition of G, and U and W are called the bipartition subsets". [60]*

Definition 3. *An inner node is "any node except the root and the leaf nodes." [7]*

Algorithm 4.1 Parsing GraphML and preprocessing data Algorithm

```
1: Parse GraphML file data to tree and graph{Adapted from [101]}
2: Call graph.compressGraph1(root){Path compression (Algorithm 4.2)}
3: Call graph.addDestinationClusters(){Duplicating the nodes in the graph (Algorithm 4.3)}
4: Call graph.transformGraphToBipartit(){Identifying the new source and destination nodes
   for each edge in the graph (Algorithm 4.4)}
```

4.2 MMV's Source Code modifications

To involve more features, the Java source code from [101, 7] was extended. *"A GraphML data importer and three preprocessing steps were added"* [9] as shown by Algorithm 4.1 (Sections 4.2.1- 4.2.3). Further, additional relation types were added (Section 4.2.4).

4.2.1 Importing and parsing GraphML files

"A new importer was added for reading and parsing GraphML [5] files and for inspecting its attributes." [9]

4.2.2 Compressing tree paths

Package structures might contain sub-paths, where each package contains exactly one sub-package and not classes, especially at the beginning of the path. This part can be combined, such that the number of levels in the icicle plot gets reduced. This gives more space to the remaining levels and thus improves the representation of the package hierarchy. This reduction is achieved by applying the path compression Algorithm 4.2. The compressing is stopped, if the node is a leaf node (Line 2). If the node has exactly one child, there are two cases: if the node is the root node, call compress with the child node (Lines 3-5), otherwise if the node is an inner node with only one child, the node is removed from the path as follows: the child's grandfather is set as the parent of the child. Then, this child is added to its grandfather's children and the child is deleted from its parent's children. After that, the node is removed from the graph. Finally, compressing is called with the child node (Line 6-12). If the node has more than one child, compressing is called with all child nodes (Line 13-16).

4.2.3 Using a bipartite graph for the underlying data structure

The initial graph is turned into a bipartite graph by replicating the graph nodes (Algorithm 4.3) to determine the origin and target nodes for each edge (Algorithm 4.4, Figure 4.1). A hashtable was added to the graph class, namely: *allDestinationClusters*. To duplicate the nodes, the duplicating algorithm (Algorithm 4.3) is performed. All the nodes in the initial graph are copied and stored in the *allDestinationClusters* hashtable (Line 1-3). The old nodes have become the source nodes and the new copied nodes have become the destination nodes. For each non root node in the initial graph (the node has a parent), its copied node is added to the children list of its copied parent and the copied parent is set to this node as parent (Line 5-7). If the node in the initial graph has no parent, its copied node is set as root (Line 9).

The identifying algorithm (Algorithm 4.4) is implemented to identify the source node and new destination node for each edge in the graph. To change the end node for each outgoing edge of each source node, a new end node is set for the outgoing edge from the destination nodes, the edge is added to the incoming edges of the destination node, the graph is updated by replacing the old outgoing edge with the modified outgoing edge in the graph (Line 2-6). The incoming edges list of the source node is cleared (Line 7). To change the end node for each self-referencing

Algorithm 4.2 Path Compression Algorithm

```
1: if the node is leaf node then
2:     Stop
3: else if the node has #children = 1 then
4:     if the node is root then
5:         Call compressGraph1(child)
6:     else
7:         Set the parent of this child node its grandfather
8:         Add this child to its grandfather'children
9:         Delete the node from its parent'children
10:        Remove the node from the graph
11:        Call compressGraph1(child)
12:    end if
13: else
14:    for all children of the node do
15:        Call compressGraph1(child)
16:    end for
17: end if
```

Algorithm 4.3 Duplicating the nodes in the graph Algorithm

```
1: for all the nodes in the graph do
2:     Add a copy of the node in allDestinationClusters hashtable {The old nodes became the
   source nodes and the new copied nodes became the destination nodes.}
3: end for
4: for all the nodes in the graph do
5:     if node has parent then
6:         Add the copied node to the children list of its copied parent in allDestinationClusters
7:         Set the copied parent to the copied node as a parent
8:     else
9:         Set the copied node in allDestinationClusters as root
10:    end if
11: end for
```

edge of each source node, a new end node is set for the edge from the destination nodes, the edge is added to the outgoing edges of the source node, the edge is added to the incoming edges of the destination node, the graph is updated by replacing the old self-referencing edge with the modified edge in the graph (Line 8-13). The self-referencing edges list of the source node is cleared (Line 14). Finally, the folding and unfolding algorithms from [7] were adapted to deal with the bipartite graph by taking into account these new changes.

Algorithm 4.4 Identifying the new source and destination nodes for each edges in the graph Algorithm

```
1:  for all the source nodes in the initial graph do
2:      for all outgoing edges of the current source node do
3:          Replace the end node of the current outgoing edge with its copied node from $allDestinationClusters$ hashtable updating edge $id$
4:          Add the modified outgoing edge as a new incoming edge of the copied node
5:          Replace the old outgoing edge with the modified outgoing edge in the graph{Update the graph}
6:      end for
7:      Clear the incoming edges list of the source node
8:      for all self-referencing edges of the node do
9:          Replace the end node of the current edge with its copied node from $allDestinationClusters$ hashtable updating edge $id$
10:         Add the edge to the outgoing edges of the source node
11:         Add the edge to the incoming edges of the new end node
12:         Replace the self-referencing edge with the modified edge in graph{Update the graph}
13:     end for
14:     Clear the self-referencing edges list of the source node
15: end for
```

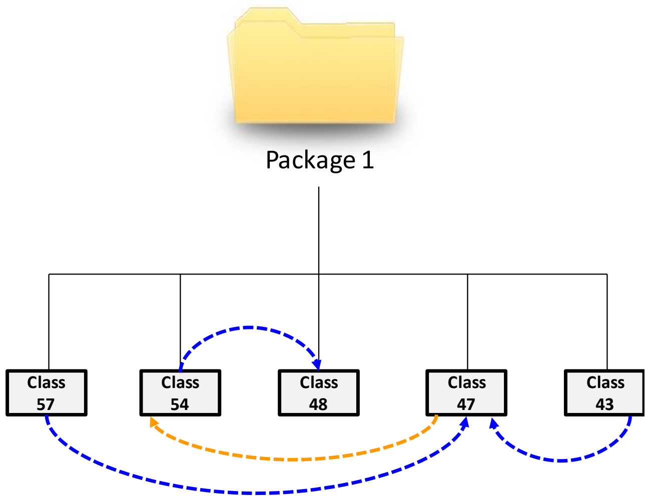
4.2.4 Involving additional relation types

Additional relation types were added using the list of code couplings introduced by Beck and Diehl [29].

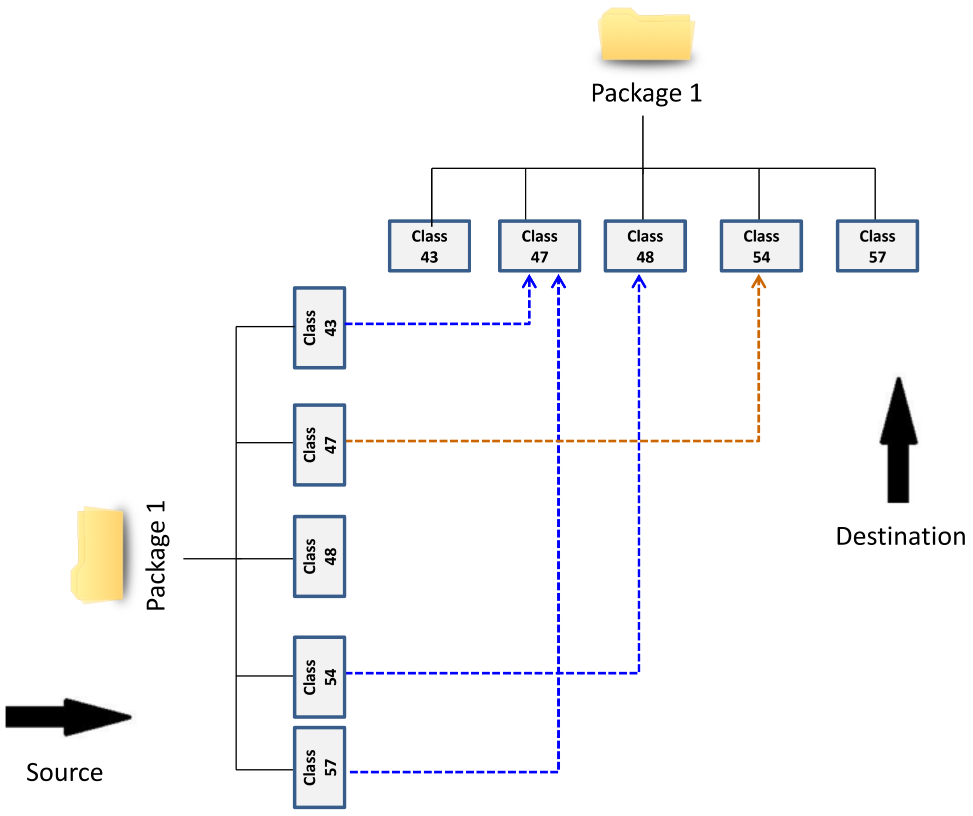
Definition 4. *Code Coupling: a relationship between classes in software systems.*

The code couplings used and implemented are [8]:

- *"**inheritance** (extend and implement dependencies)"*
- *"**aggregation** (usage of another type as a class attribute)"*
- *"**usage** (structural dependencies on method level)"*
- *"**evolutionary coupling** (files that were changed together frequently in the past)"*
- *"**code clones** (files that are reasonably covered by the same code clones)"*
- *"**semantic similarity** (similar vocabulary used in identifiers and comments)"*



(a) Graph



(b) Bipartite Graph

Figure 4.1: Turning a graph into bipartite graph

4.3 Enhancements of the Visualization

The updated visualization and the modified graphical interface were applied to JHotDraw [62]. Figure 4.2 shows the structure of JHotDraw using IMMV. The enhancements include highlighting the main diagonal using a gray background of the cells. Further, labels are partially shown in the icicle plots, the color legend was improved, and levels in the icicle plots are shown.

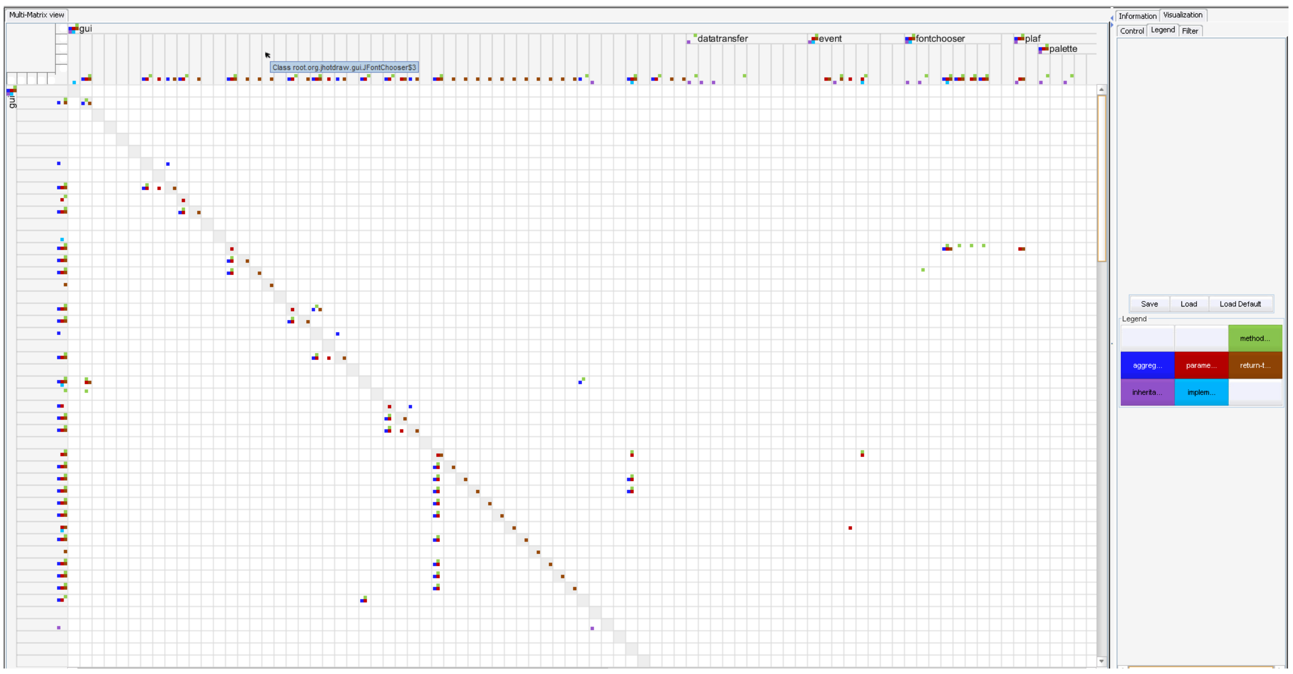


Figure 4.2: The improved MMV, which visualizes relations from the JHotDraw software system.

4.3.1 Labels

"Inside the icicle plots, nodes have labels, if enough screen space is available." Additionally, the user can use tool tips to show the label by moving the mouse over the cell (Figure 4.2). [9]

4.3.2 Enhanced Color Legend

"The color legend is used to show the color mapping of the code couplings and their positions inside the matrix cells. The default color mapping assigns the bright colors to the corner subcells and the darker colors to the other subcells (Figure 4.2)." [9]

4.3.3 Levels in Icicle Plots

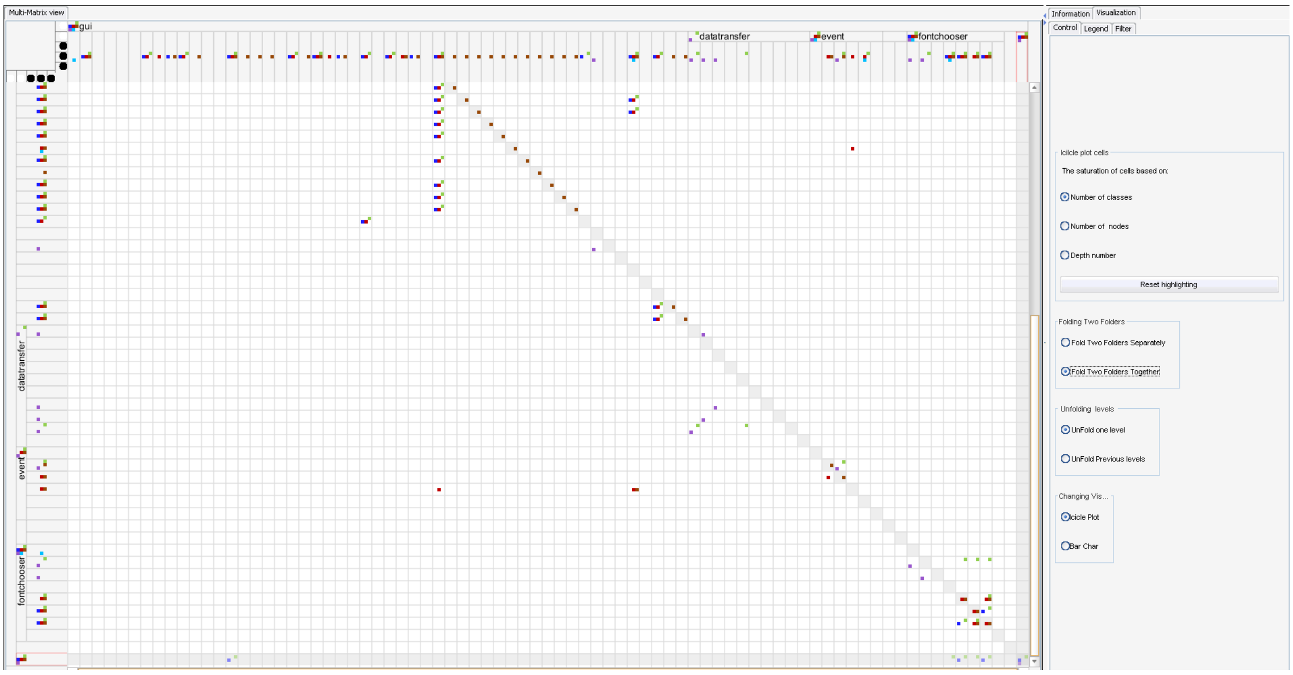
"Each level in the top and in the left icicle plot is represented with a white circle in the upper left corner (Figure 4.2)." [9]

4.4 Enhancement of the Interaction

4.4.1 Additional Folding and Unfolding Options

Many additional folding and unfolding options were added:

- *"In the matrix, the background color of the folded package column and row are highlighted with gray."* [9]
- Packages in the top icicle and in the left icicle can be folded separately or together by selecting the radio button on the right (Figure 4.3).
- The folded packages can be unfolded for one level or to the previous state by selecting the radio button on the right (Figure 4.3).



Icicle plot cells
The saturation of cells based on:
[✓ Number of classes] Number of nodes Depth number

Folding Two Folders
Fold Two Folders Separately [✓ Fold Two Folders Together]

Unfolding levels
[✓ UnFold one level] UnFold Previous levels

Changing Vis...
[✓ Icicle Plot] Bar Char

Figure 4.3: Using radio buttons (right) to choose whether to fold source and destination packages separately or together

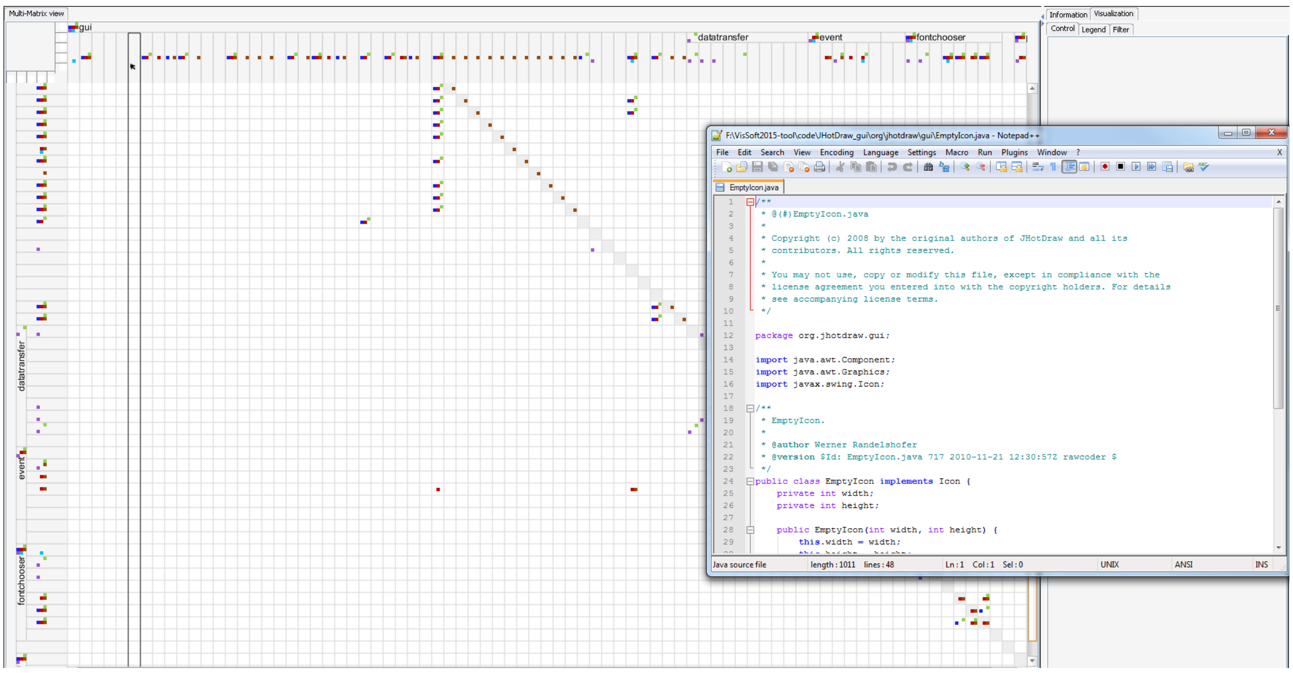
- Each level in the icicle plot can be folded by mouse double clicking on its respective white circle in the upper left corner. After that, the folded level can be unfolded by mouse double clicking on its respective black circle in the upper left corner (Figure 4.3).

4.4.2 Opening Data Source

The user can open an editor with the source code of a class by double clicking on the cell representing the class in the icicle plot. This enables to directly explore and verify the underlying source code behind the discovered visual structures (Figure 4.4).

4.4.3 Color Legend Interaction

Using the enhanced color legend, the user can change the relations color mapping in the matrix cells by choosing different colors. Also, the sub-cell positions of the couplings can be changed by selecting the coupling from the list (Figure 4.5).



```
/**
 * @(#)EmptyIcon.java
 *
 * Copyright (c) 2008 by the original authors of JHotDraw and all its
 * contributors. All rights reserved.
 *
 * You may not use, copy or modify this file, except in compliance with the
 * license agreement you entered into with the copyright holders. For details
 * see accompanying license terms.
 */

package org.jhotdraw.gui;

import java.awt.Component;
import java.awt.Graphics;
import javax.swing.Icon;

/**
 * EmptyIcon.
 *
 * @author Werner Randelshofer
 * @version $Id: EmptyIcon.java 717 2010-11-21 12:30:57Z rawcoder $
 */
public class EmptyIcon implements Icon {
    private int width;
    private int height;

    public EmptyIcon(int width, int height) {
        this.width = width;
```

Figure 4.4: Using the data source editor

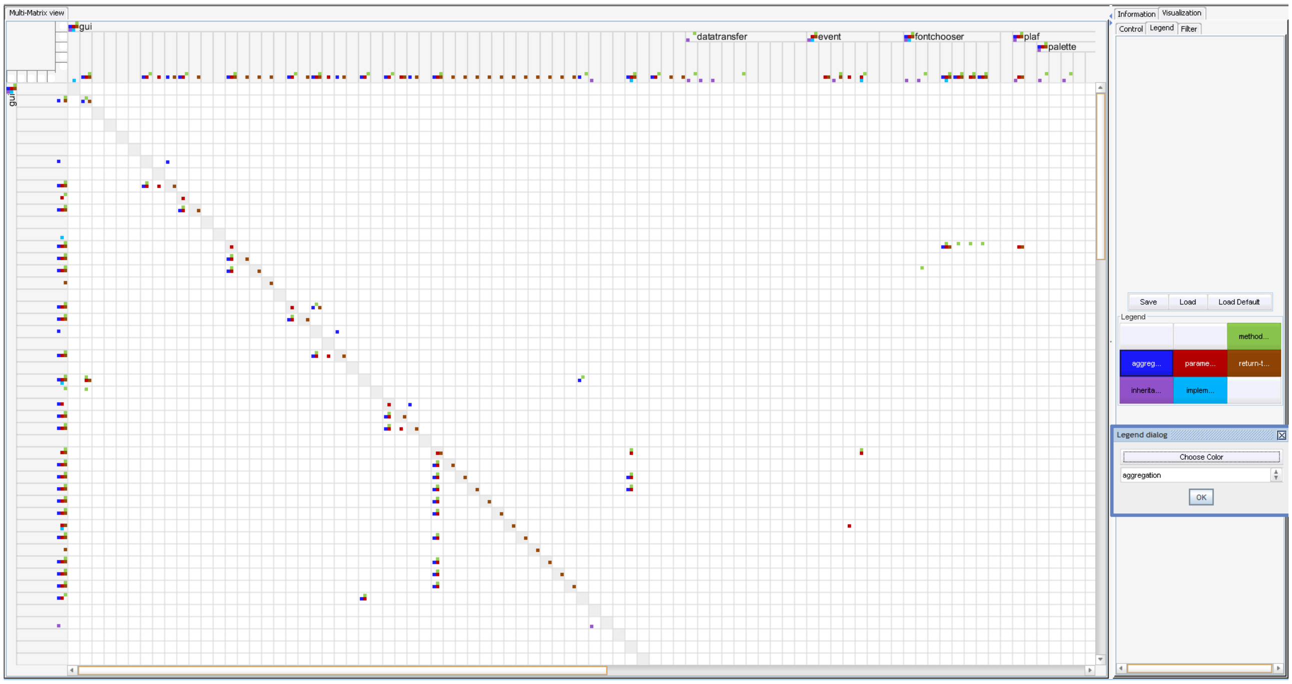


Figure 4.5: Interacting with the enhanced color legend

Chapter 5

Determination of Pattern in the interactive Multi-Matrix Visualization

5.1 Preamble

This chapter is the extended version of the explorative user study [8] that was conducted with interactive versions of the parallel node-link visualization (PNLV) [26] and the interactive multi-matrix approach (IMMV) [101, 7, 9], and Chapter 4. It focuses on a realistic application scenario. The study was performed as cooperation of the TU Kaiserslautern and of the University of Trier. Please note, that a summary of this study was included by the dissertation of Fabian Beck [27]— a co-author of this study. The core purpose of this qualitative study is to explore visual structures that are formed by visualizing relationships between classes as inheritance and aggregation in software systems. PNLV and IMMV have been used to visualize these relationships in different datasets. A counterbalanced within-subject design was used to evaluate these two visualizations.

This extended version added Section 5.4 as new part, which covers the two pretests of the explorative user study. Additionally, the introduction section (Section 5.2) was modified. The other Sections, Section 5.3, Section 5.5, Section 5.6, Section 5.7, Section 5.8, and Section 5.9 are taken from the paper.

5.2 Introduction

Directed and undirected graphs are widely used to model and to understand the relations between entities that together form a software system. In software systems, different types of couplings exist between classes and interfaces such as inheritance, aggregation, usage, code clones, evolutionary (co-change) coupling, or semantic similarity. These couplings are complementary to each other, revealing multiple aspects of the same software system. Hence, approaches for analyzing graphs benefit from discerning instead of aggregating different types of edges.

Much research has concentrated on using visual approaches for analyzing graphs in a scalable way [98], but, only recently, some approaches focused specifically on discerning and comparing multiple types of edges [26, 46, 52, 53, 101]. In this work, two complementary approaches were selected and evaluated in a software engineering scenario. The first approach, called *parallel node-link visualization* (PNLV) [26], depicts multiple types of edges in separate, parallel node-link diagrams that are placed side by side. The second approach, called *interactive multi-matrix visualization* (IMMV) [101, 7, 9] (Chapter 3 and 4), uses an adjacency matrix for visually representing the graph with multiple types of edges in each cell. These were the only approaches

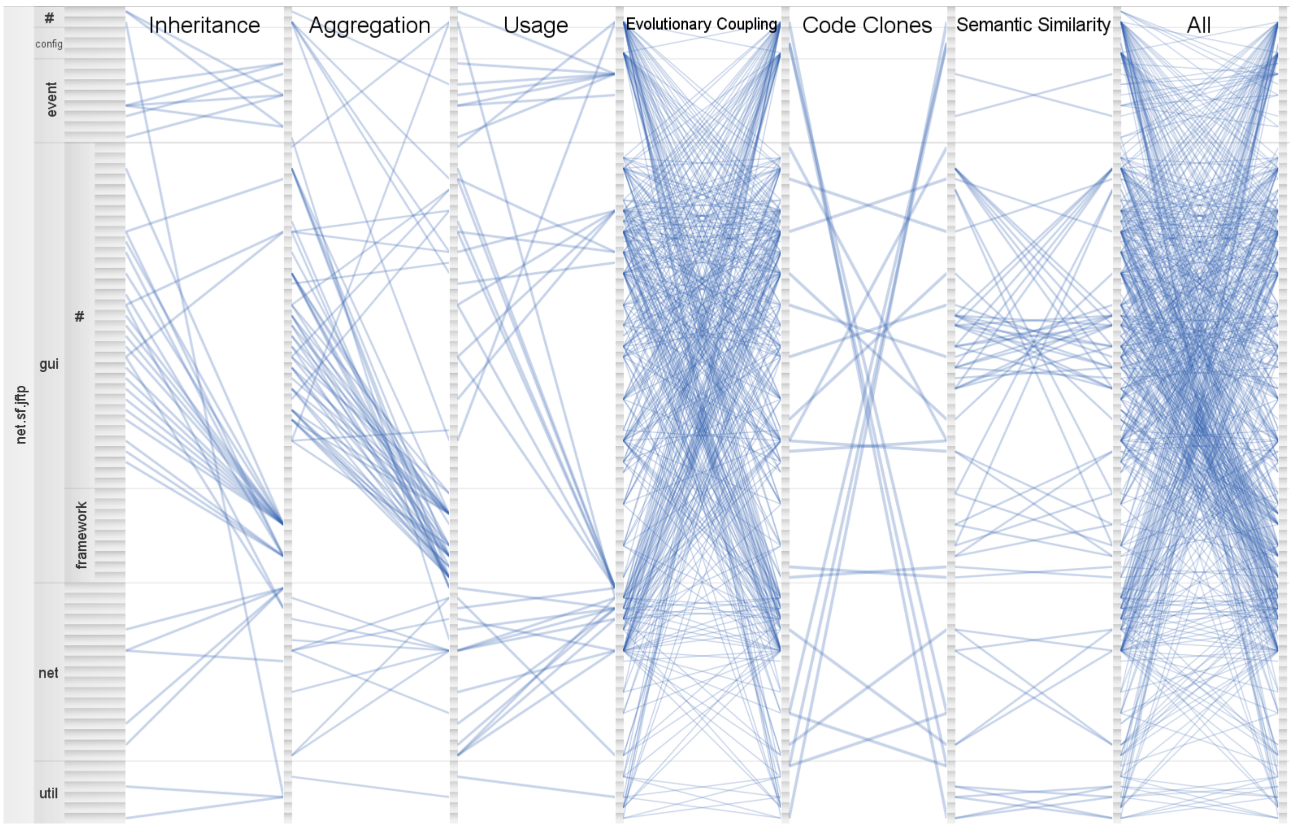


Figure 5.1: JFtp project with multiple types of couplings visualized by the parallel node-link visualization (PNLV) of [26]

that provide the required scalability (with respect to the number of nodes, edges, and types of edges) to represent coupling graphs of real-world software systems having several types of coupling. Figure 5.1 and Figure 5.2 contrasts the two approaches used, visualizing the *JFtp* project.

The specific research question is: *"What higher-level coupling structures users are able to retrieve and compare with the help of these visualizations?"* Further, it was investigated how the detected and compared structures can be used for tasks in the context of software engineering. The main contributions of this study are the following:

- The evaluation of the utility of two recent visual approaches [26, 101, 9] for comparing multiple types of edges in a realistic software engineering scenario.
- The identification of visual structures and the general graph structures that users find with these visualizations.
- The study of the strategies that the participants applied for comparing several types of edges.
- The exploration of software engineering problems that can be addressed with these visualizations.

For graphs with only edges of a single type, researchers have already conducted comparison studies between node-link and matrix graph visualizations based on predetermined, low-level tasks [57, 58, 69]. These studies suggest that matrix visualizations are often more suitable for analyzing larger and denser graphs. The study extends these evaluations to multi-type edges and to more complex, higher-level tasks. A realistic application scenario was designed and

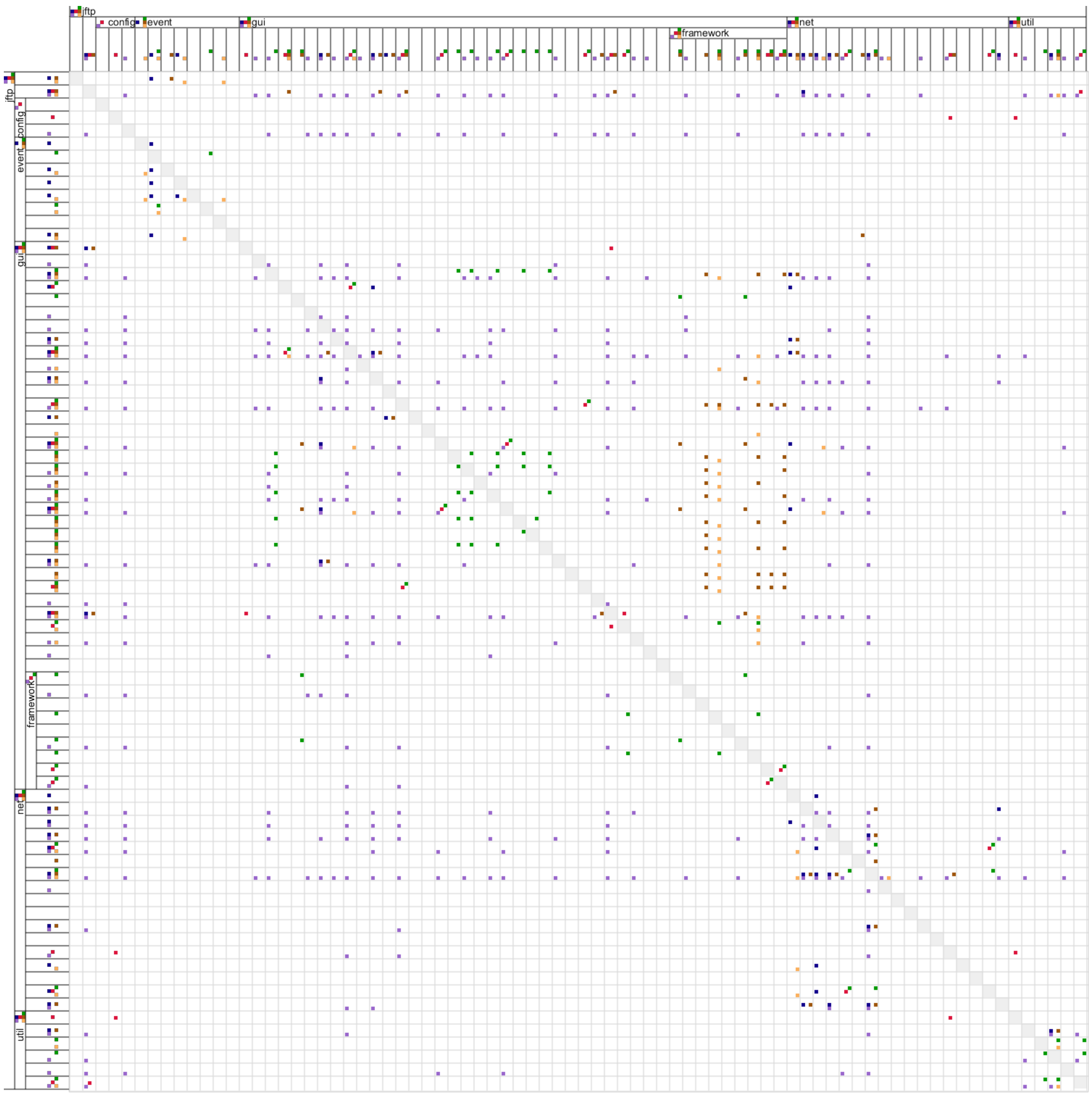
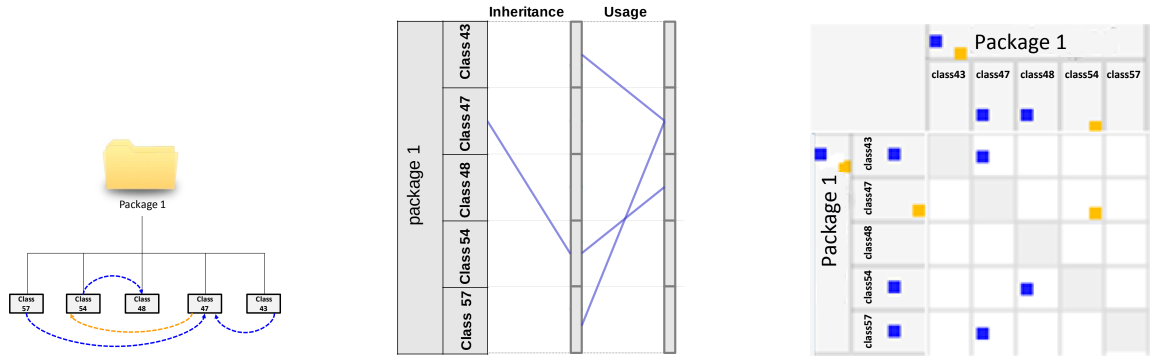


Figure 5.2: JFtp project with multiple types of couplings visualized by the interactive multi-matrix visualization (IMMV) of [101, 7] (Chapter 4)

eight software engineers were asked to gain an understanding of previously unknown software systems. The observations show which specific software engineering problems can be solved with the two visualization approaches. In contrast to the expectations and previous results on low-level tasks, the two approaches showed very similar characteristics though following contrary visualization paradigms.

The remainder of the work is structured as follows: In Section 5.3, the two evaluated interactive visualization approaches are presented in detail. In Section 5.4.1, the first pretest of this study is presented, while the follow-up pretest is described in Section 5.4.2. The experimental design of the study is described by Section 5.5. Further, Section 5.6 reports the results of the study by analyzing visual structures, their interactive exploration by the participants, and the participants' feedback. The results are discussed in a broader context in Section 5.7. Related work on visually discerning multiple types of edges as well as on comparing node-link and matrix visualizations is presented in Section 5.8. Finally, Section 5.9 concludes the paper.



(a) Standard node-link representation (b) PNLV (c) IMMV

Figure 5.3: Illustrating example for visualizing a software system with different types of code couplings modeled as a graph with multiple types of edges; orange: inheritance couplings; blue: usage couplings (except PNLV where colors are used for highlighting).

5.3 Visualization Approaches

In this study, we target two related visualization approaches that are briefly introduced in the following: the parallel node-link visualization (PNLV, Section 5.3.1) and the interactive multi-matrix visualization (IMMV, Section 5.3.2). As an illustrating example, Figure 5.3 presents a small sample software system consisting of one package that includes five classes. These classes are connected by two types of code coupling: inheritance and usage. The system is modeled as a graph—while the classes (and interfaces) of the system form vertices, the different types of code couplings are mapped to different types of edges. Moreover, the package structure is used to hierarchically organize the vertices. Figure 5.3 shows three variants of visualizing this sample data set: Figure 5.3(a) depicts a standard node-link representation, Figure 5.3(b) sketches the diagram in PNLV, and Figure 5.3(c) represents the data set in IMMV.

5.3.1 Parallel Node-Link Visualization (PNLV)

In the parallel node-link visualization (PNLV) [26] as shown in Figure 5.3(b), multiple node-link diagrams are juxtaposed as columns side by side. In each column, the nodes representing the classes and interfaces are placed above each other on a vertical axis. Each node is split and has one port for all its outgoing edges and one port for all its incoming edges; these ports are aligned horizontally. The edges are directed from the outgoing ports on the left to the incoming ports on the right. Each of the juxtaposed diagrams represents a different type of edges.

A layered icicle plot is attached to the left side of the diagram to show the hierarchy of the software system. The horizontal lines in the icicle plot between boxes that represent packages are extended through the whole visualization. The vertical separators between the juxtaposed node-link diagrams repeat the leaf level of cells in the icicle plot. Inside the icicle plot, nodes display labels if possible or on demand when hovered by the mouse. The visualization is adapted to the window size so that all data is always completely visible.

Highlighting by color is used for discerning a set of selected nodes from non-selected ones as well as for marking their outgoing and incoming edges. Edges starting at selected nodes are colored green, those ending at selected nodes are colored red, and those starting and ending at selected nodes are colored brownish-green; a light blue color is used for the other edges. A single node is highlighted by clicking on one of its visual representation in the diagram while a set of nodes contained in a package is selected by clicking on the package. The participant can open the source code of a class or interface by double-clicking.

5.3.2 Interactive Multi-Matrix Visualization (IMMV)

In the interactive multi-matrix visualization (IMMV) [101, 9] as shown in Figure 5.3(c), the classes of a software system and the code couplings between them are visualized using an adjacency matrix representation of the underlying graph. Every cell of the matrix is divided into sub-cells, each sub-cell representing a different edge type (code coupling). A colored sub-cell appears in the matrix if there exists an edge of the respective type from the vertex in the current row to the vertex in the current column. To more clearly distinguish the types, a different color is used for each type; a color legend is attached to the matrix.

Analogous to PNLV, IMMV is combined with layered icicle plots to display the package structure of the software system [9]. A copy of the same icicle plot is placed at the left as well as on the top of the matrix. Packages are labeled and the names of classes and interfaces can be retrieved on demand as tooltips. Further, summaries of the code couplings are integrated into the icicle plots aggregating all couplings of a row or column respectively; these summaries are aggregated on package level and displayed in the package representations. If the matrix is larger than the window, the participant can scroll to explore the complete matrix.

A set of classes and interfaces or packages can be selected by clicking for highlighting them in the icicle plots and in the matrix: the rows and columns of all selected entities become surrounded by a strong black border line. It is further possible to select a cell of the matrix—both respective row and column are highlighted. Again, the participant can open the source code of a class by double-clicking.

5.4 Pretests

Two pretest were performed to refine the experiment design and to avoid mistakes. Section 5.4.1 describes the first pretest, while Section 5.4.2 describes the second pretest.

5.4.1 First Pretest

In this pretest experiment, one participant was involved. He is a PhD computer science student from the University of Trier. The primary task of the participant is to find which visual structures can be displayed by each tool.

5.4.1.1 Experiment Setup

The participant used a PC with an LCD 23” screen with full HD resolution, mouse, and keyboard. The operating system was Windows 7. The two tools were implemented using the Java programming language. The format of the datasets was the GraphML File Format. A3 papers were used for the visualizations on paper. The data was loaded into the PNLV tool using batch files (jar files) to decrease the preparation time. The data was loaded into the IMMV tool manually. The source code files of the datasets were displayed in Notepad++ [45] on demand.

5.4.1.2 Sample Description

There was one participant in this pretest. He is a PhD computer science student from the University of Trier. His age is 29 years and his gender is male. He has 10 years of experience in

Coupling types	JFtp			JUnit			JHotDraw		
	P	N	E	P	N	E	P	N	E
Inheritance	8	78	40	26	119	63	9	82	10
Aggregation	8	78	66	26	119	35	9	82	17
Usage	8	78	38	26	119	251	9	82	56
Evolutionary	8	78	380	26	119	76	9	82	71
Code-Clone	8	78	22	26	119	40	9	82	50
Semantic Similarity	8	78	62	26	119	236	9	82	132

Table 5.1: Statistics on the software systems used as datasets in the first pretest with P: number of packages, N: number of nodes, and E: number of edges.

programming. He regularly uses the Java language for projects with 51-100 classes and less than three developers. He regularly uses UML for software development as visual representation.

5.4.1.3 Dataset Description

Six different types of couplings were considered: Inheritance, Aggregation, Usage, Evolution, Code Clones, and Semantic Similarity. Three datasets were used:

- JFtp, a Java file transfer client.
- JUnit, an open source Java unit-testing framework.
- JHotDraw, a Java GUI framework.

The statistics of these datasets are given in Table 5.1.

5.4.1.4 Experiment Protocol

Two persons are responsible for managing the experiment: a moderator and an observer. The observer takes notes. The sequence of the experiment followed these steps:

1. The data tutorial was presented and read for the participant by the moderator. Then, the experiment was performed according to the combination in Table 5.2.

Pretest number	Presentation order
1	JUnit + IMMV → JFtp + PNLV

Table 5.2: The combinations of tools and datasets

2. (a) The participant followed the tool tutorial for IMMV, which was read by the moderator. After the tutorial, the participant was free to ask his own questions to the moderator.
   (b) The participant performed a trial for IMMV to see if he understands the tool. The tasks of the trial were described by the moderator. The JHotDraw dataset was used in this trial. The participant was free to ask his own questions during the trial.
   (c) The participant performed the paper test for IMMV using the JUnit dataset. The experiment question was:
      - What interesting visual structures do you find in IMMV? Please mark and describe them?

   (d) The participant performed the interactive test for IMMV using the JUnit dataset. The experiment questions were:
      - Select the three visual structures, which are significantly different from each other and which hint at an interesting phenomenon. Please use the interactive visualization to explore them. Explain your findings.
      - Why did you select these visual structures?

3. (a) The participant followed the tool tutorial for the PNLV, which was read by the moderator. After the tutorial, the participant was free to ask his own questions to the moderator.
   (b) The participant performed a trial for the PNLV to see if he understands the tool. The tasks of this trial were described by the moderator. Also, the JHotDraw dataset was used in this trial. The participant was free to ask his own questions during the trial.
   (c) The participant performed the paper test for PNLV using the JFtp dataset. The experiment question was:
      - What interesting visual structures do you find in PNLV? Please mark and describe them?
   (d) The participant performed the interactive test for PNLV using the JFtp dataset. The experiment questions were
      - Select the three visual structures, which are significantly different from each other and which hint at an interesting phenomenon. Please use the interactive visualization to explore them. Explain your findings.
      - Why did you select these visual structures?

4. The participant performed a color blindness test.

5. The participant was provided with the general questionnaire, which consists of closed questions, open questions, and statements, and the final questionnaire, which consists of closed questions, and five-level Likert scale, open questions, and statements.

5.4.1.5 Results

In this pretest, the participant was asked to mark visual structures in the printed visualizations. The user identified fan visual structures and one cross beam visual structure in the PNLV paper test, and he identified lines visual structures, clusters off the main-diagonal, and clusters on the main-diagonal in IMMV paper test (Table 5.11 shows the classification of the identified visual structures). After that, the participant explored three of the identified visual structures using the respective interactive tool. In PNLV, he selected two fan and one cross beam visual structures. For IMMV, he selected line, main-diagonal cluster, and off-diagonal cluster. By exploring the two fans (PNLV) and the line (IMMV), he connected them to find a central class/interface. By exploring the cross beam (PNLV), and main-diagonal cluster and off-diagonal cluster (IMMV), he connected them to understand a package related software engineering task. Also, he used highlighting and opening the source code editor to interact with both tools. In the questionnaire of this pretest, he preferred PNLV for analyzing software projects because of the separated relation views (columns). Hence, the focus would be on single relation type. In comparison, he recommended IMMV for larger projects because PNLV could get messy when displaying too many edges. With respect of usefulness, both tools received the same answers (PNLV: 2; IMMV 2; scale from strongly agree (1) to strongly disagree (5) that it

Step	Table Number
IMMV trial using the JHotDraw dataset	Table A.1
PNLV trial using the JHotDraw dataset	Table A.2
IMMV paper test using the JUnit dataset	Table A.3
The interactive IMMV test using the JUnit dataset	Table A.5
PNLV paper test using JFtp dataset	Table A.4
The interactive PNLV test using JFtp dataset	Table A.6
The general questionnaire	Table A.7
The final questionnaire	Table A.8

Table 5.3: The results for each step in the first pretest

No	Step	Sub step	Start	End	Time
1	Data Tutorial		15:28	15:35	0:07
2.1	IMMV	Tutorial	15:36	15:43	0:07
2.2		Trial	15:43	15:55	0:12
2.3		Paper	15:55	16:07	0:12
2.4		Tool	16:10	16:24	0:14
3.1	PNLV	Tutorial	16:26	16:30	0:04
3.2		Trial	16:31	16:38	0:07
3.3		Paper	16:38	16:45	0:07
3.4		Tool	16:46	16:57	0:11
4	Color Blindness		16:58	16:59	0:01
5	Questionnaire		17:00	17:30	0:30
	Total				1:52

Table 5.4: Experiment protocol and time per step in the first pretest

is useful). The participant saw the main area of application of the two tools in improving the architecture or design of software systems.

Appendix A lists all the results for each step as shown in Table 5.3.

The recorded times for each steps in the pretest experiment are shown in the Table 5.4. Many steps exceeded the time limit for each step because of the following reasons:

- The data tutorial was too long and complicated.
- The matrix and the node-link tutorials were too long and they had redundancy.
- The trials and questionnaires were too long.
- The labels of the matrix on paper were not clear.
- The tooltips took too much time to appear in the matrix tool.

5.4.1.6 Results Analysis

In the PNLV paper test, the participant looked at separate couplings (columns) to find visual structures. The participant looked mainly at fan visual structures. With larger dataset, it could be hard to find such visual structure because of visual clutter in PNLV.

In IMMV paper test, the participant looked at the combination of multi-types couplings to find visual structures in the matrix. The existence of different combinations could explain why the participant spent more time to find them.

5.4.1.7 Problems and Modification

Many problems appeared in the pretest and a lot of feedback was provided by the participant. Decreasing the total time of the experiment from 112 minutes to 90 minutes was the first issue. To decrease the total time:

- The data tutorial was reduced by removing details such as the thresholds used in the datasets and by more focusing on the concepts.
- The two tool tutorials were shortened by removing the redundancy and their screenshots were changed to show the JHotDraw dataset instead of the JFtp dataset.
- The trial questions, the general, and the final questionnaires were shortened by removing and merging questions according to the feedback from the participant.
- The question "Why did you select these visual structures?" was removed from the experiment questions.

Additionally, it was hard to record the use of interactions on-the-fly by taking notes during the interactive tests, so recording screen and voices and then extracting the sequence of interactions manually later were considered for the final experiment.

Moreover, IMMV was enhanced on paper by:

- Enlarging the font size of the labels and modifying the alignment of vertical labels
- Improving the current color legend
- Using a larger paper size for the matrix visualization

Also, the interactive tool for IMMV was enhanced by including the following features according to the feedback:

- Displaying the label using the tooltips by moving the mouse over the cell in the matrix
- Adding the batch files to load data in the matrix tool
- Opening the source code using mouse double click
- Not collapsing packages with single classes (path compression)

It was hard to vary among dataset sizes for the participant because it was expected that he would consume too much time. The JFtp and the JUnit datasets, which were used in the experiment, have different sizes (JUnit is larger than JFtp). The JFtp dataset was used with the PNLV tool and the JUnit dataset was used with the IMMV. In the final questionnaire, the user indicated that he preferred using PNLV over IMMV for small projects because of the separated relation views. He also indicated that he preferred using IMMV for large projects because PNLV can get messy when displaying too many edges. To make sure that the dataset size dose not bias the decision of the participants in the final experiment, using two similar sized datasets is recommended.

5.4.2 Second Pretest

In this pretest experiment, one participant was involved. He is a PhD computer science student from the University of Trier. The primary task of the participant is to find which visual structures can be displayed by each tool.

Couplings types	Stripes			Checkstyle			JHotDraw		
	P	N	E	P	N	E	P	N	E
Inheritance	20	238	143	22	261	207	9	82	10
Aggregation	20	238	131	22	261	69	9	82	17
Usage	20	238	614	22	261	479	9	82	56
Evolutionary	20	238	262	22	261	586	9	82	71
Code-clone	20	238	296	22	261	440	9	82	50
Semantic similarity	20	238	462	22	261	1014	9	82	132

Table 5.5: Statistics on the software systems used as datasets in the second pretest with P: number of packages, N: number of nodes, and E: number of edges.

5.4.2.1 Experiment Setup

The participant used a PC with an LCD 23" screen with full HD resolution, mouse, and keyboard. The operating system was Windows 7. The two tools were implemented using the Java programming language. The format of the datasets was the GraphML File Format. A1 paper was used for IMMV and A2 paper was used for PNLV on the paper. The data was loaded into the PNLV tool using batch files (jar files) to decrease the preparation time. The data was loaded into the matrix tool manually instead of using batch files because the previous improvement did not work properly. The source code files of the datasets were displayed in Notepad++ [45] on demand. Screen and voice were recorded with Camtasia Studio [91].

5.4.2.2 Sample Description

There was one participant in this pretest. He is a PhD computer science student from the University of Trier. His age is 29 years and his gender is male. He has 15 years of experience in programming. He regularly uses the Java language for projects with 51-100 classes alone.

5.4.2.3 Dataset Description

Six different types of couplings were considered: Inheritance, Aggregation, Usage, Evolutionary, Code Clones, and Semantic Similarity. Three datasets were used:

- JHotDraw, a Java GUI framework.
- Stripes, an open source framework for developing web applications.
- Checkstyle, a tool for enforcing coding standard.

The statistics of these datasets are given in Table 5.5.

5.4.2.4 Experiment Protocol

Two persons are responsible for managing the experiment: a moderator and observer. The observer takes notes. The sequence of the experiment followed these steps:

1. The data tutorial was presented and read for the participant by the moderator. Then, the experiment was performed according to the combination in Table 5.6.
2. (a) The participant followed the tool tutorial for PNLV, which was read by the moderator. After the tutorial, the participant was free to ask his own questions to the moderator.

Pretest number	Presentation order
2	Checkstyle + PNLV → Stripes + IMMV

Table 5.6: The combinations of tools and datasets

   (b) The participant performed a trial for PNLV to see if he understands the tool. The tasks of the trial were described by the moderator. The JHotDraw dataset was used in this trial. The participant was free to ask his own questions during the trial.

   (c) The participant performed the paper test for PNLV using the Checkstyle dataset, his voice was recorded. The experiment question was:

   - What interesting visual structures do you find in PNLV? Please mark and describe them?

   (d) The participant performed the interactive test for PNLV using the Checkstyle dataset, the screen was recorded. The experiment question was:

   - Select the three visual structures, which are significantly different from each other and which hint at an interesting phenomenon. Please use the interactive visualization to explore them. Explain your findings.

3. (a) The participant followed the tool tutorial for IMMV, which was read by the moderator. After the tutorial, the participant was free to ask his own questions to the moderator.

   (b) The participant performed a trial for IMMV to see if he understands the tool. The tasks of this trial were described by the moderator. The JHotDraw dataset was used in this trial. The participant was free to ask his own questions during the trial.

   (c) The participant performed the paper test for IMMV using the Stripes dataset, his voice was recorded. The experiment question was:

   - What interesting visual structures do you find in IMMV? Please mark and describe them?

   (d) The participant performed the interactive test for IMMV using the Stripes dataset, the screen was recorded. The experiment question was:

   - Select the three visual structures, which are significantly different from each other and which hint at an interesting phenomenon. Please use the interactive visualization to explore them. Explain your findings.

4. The participant performed a color blindness test.

5. The participant was provided with the general questionnaire, which consists of closed questions, open questions, and statements and the final questionnaire, which consists of closed questions, and five-level Likert scale open questions and statements.

5.4.2.5 Results

In this pretest, the participant was asked to mark visual structures in the printed visualizations. The user identified fans, gaps, beam, and cross beam visual structures in the PNLV paper test, and he identified line, off-diagonal clusters, and main-diagonal clusters in the IMMV paper test (Table 5.11 shows the classification of the identified visual structures). After that, the participant explored three of the identified visual structures using the respective interactive tool. In PNLV, he selected one fan and two beams visual structures. For IMMV, he selected one line, one off-diagonal cluster, and main-diagonal cluster visual structures. By exploring the beams

Step	Table Number
IMMV trial using the JHotDraw dataset	Table B.2
PNLV trial using the JHotDraw dataset	Table B.1
IMMV paper test using the Stripes dataset	Table B.3
The interactive IMMV test using the Stripes dataset	Table B.5
PNLV paper test using the Checkstyle dataset	Table B.4
The interactive PNLV test using the Checkstyle dataset	Table B.6
The The general questionnaire	Table B.7
The final questionnaire	Table B.8

Table 5.7: The results for each step in the second pretest

No	Step	Sub step	Start	End	Time
1	Data Tutorial		14:00	14:05	0:05
2.1	MMV	Tutorial	14:54	14:59	0:05
2.2		Trial	14:59	15:07	0:12
2.3		Paper	15:09	15:21	0:12
2.4		Tool	15:22	15:41	0:19
3.1	PNLV	Tutorial	14:06	14:12	0:06
3.2		Trial	14:13	14:20	0:07
3.3		Paper	14:21	14:32	0:11
3.4		Tool	14:35	14:51	0:16
4	Color Blindness		15:41	15:42	0:01
5	Questionnaire		15:42	16:28	0:46
	Total				2:20

Table 5.8: Experiment protocol and time per step in the second pretest

(PNLV), one off-diagonal cluster (IMMV), and one main-diagonal cluster (IMMV) visual structure, he connected them to understand a package related software engineering task. By exploring the fan (PNLV) and the line (IMMV), he connected them to find a central class/interface. Also, he used highlighting and opened the source code editor to interact with both tools. In the questionnaire of this pretest, he preferred PNLV for analyzing software projects because he had the feeling that it is more intuitive and that it provides good overview but he was not sure if it works for larger projects. For the matrix, he had a much steeper learning curve. With respect of usefulness, the tools received the answers (PNLV: 2; IMMV 3; scale from strongly agree (1) to strongly disagree (5) that it is useful). The participant saw the main area of application of the two tools in program comprehension and improving the architecture or design of software systems. In addition, he saw debugging/bug fixing as possible application area of the PNLV tool.

Appendix B lists all the results for each step as shown in Table 5.7.

The recorded times for each step in the pretest experiment are shown in the Table 5.8. Many steps exceeded the time limit for each step because of the following reasons:

- The trial questions were long.
- The final questionnaire was too long.
- The tooltips took too much time to appear in the matrix tool (the previous improvement did not work properly).

5.4.2.6 Results Analyses

The participant was interested in general phenomena. He used the source code editor only a few times. In the PNLV paper test, the participant looked at separate couplings to find visual structures. The participant looked mainly at the fan visual structure. Also, he looked at the cross visual structure and the empty spaces in PNLV.

In the IMMV paper test, the participant looked at the combination of couplings to find visual structures in the matrix. The existence of different combinations could explain why the participant spent more time to find them.

5.4.2.7 Problems and Modification

Many problems appeared in the second pretest and a lot of feedback was provided by the participant. Decreasing the total time of the experiment from 140 minutes was the first issue. The time increased from the first pretest because the participant spent too much time to fill the final questionnaire and to use the matrix interactive tool. To decrease the total time:

- The data tutorial was reduced again by shortening details about some code couplings.
- The trial questions were rephrased.
- The general and the final questionnaires were shortened again by removing and merging their questions according to the feedback from the participant.

The participant missed the overview of the matrix on the paper because the paper size was too large (A1). Accordingly, the matrix visualization was enhanced on paper by:

- The papers of both tools are printed using A2 paper size.
- The color of the hierarchies' borders of the matrix is now darker on paper.

Also, the interactive matrix visualization was enhanced by including the following features according to the feedback:

- The tooltips show the source class and the destination class when moving the mouse over the matrix cells.
- The color legend is not interactive anymore.
- The thickness of the highlighting lines was decreased.
- The participant can highlight many classes and many packages at the same time by pressing $< control + C >$ on the keyboard.
- The participant can remove the highlighting by pressing $< control + A >$ on the keyboard.

The participant said that "he felt boring after the first tool tests because the test needed too much time". Also, he tried to find different phenomenas using the second tool because he assumed using the same dataset. To counterbalance for biases such as learning and tiring effects in the final experiment, using two similar sized datasets is recommended. Additionally, the participant should know that he is using different datasets during the experiment.

5.5 Final Experiment Design

While carefully designing the experiment, we discussed quantitative as well as qualitative approaches but finally decided to conduct a mostly qualitative study in order to explore how users—provided with a very general task—work naturally with the complex visualization approaches. Ellis and Dix [50] argue that explorative studies such as ours are most suitable for understanding the utility of information visualizations. Lam et al. [71] surveyed and classified empirical research in information visualization; with respect to their taxonomy of seven scenarios, our approach falls into the category of *evaluating visual data analysis and reasoning (VDAR)*—field studies and partially controlled experiments are typical and appropriate for this kind of research investigating how visualization supports data exploration and knowledge discovery.

5.5.1 Research Goal

The specific research question that we want to answer is: *"What higher-level coupling structures users are able to retrieve and compare with the help of these visualizations?"* Hence, the core purpose of our study is to determine which visual structures the participants are able to identify and to interpret in the two presented visualizations in a realistic software engineering context. We define a *visual structure* as any set of visual elements that are perceptually grouped. On a basic level, perceptual grouping is described by the Gestalt Laws [70]. A visual structure can be, for example, a fan of links (PNLV), a group of equally colored cells (IMMV), or the co-occurrence of types of edges as links in different columns (PNLV) or as colored sub-cells (IMMV).

Working with the visualizations, identifying structures like these is a first step towards making sense of the presented information. As a next step, the visual structures need to be connected to specific graph structures such as vertices having a high degree or strongly coupled clusters of vertices. These graph structures can be embedded into the domain-specific context to finally draw conclusions from the visualization. Our study intends to cover this process of interpretation as summarized in Figure 5.4 as a whole.

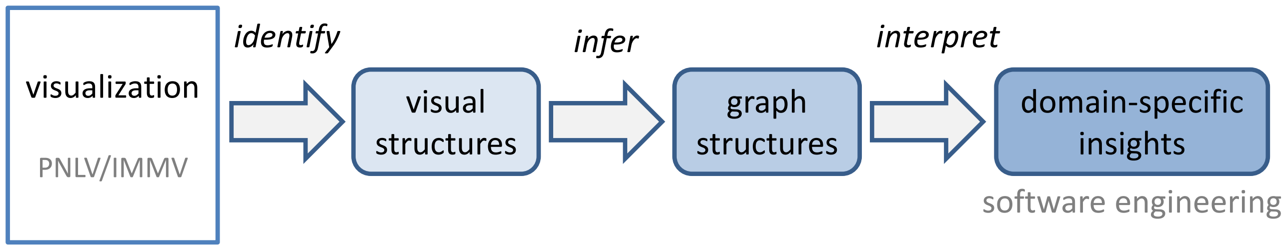


Figure 5.4: Interpretation process for deriving domain-specific insights from a graph visualization.

Finally, we analyzed which strategies the participants applied for comparing different edge types.

5.5.2 Experiment Setup

The study was performed as a lab experiment following a counterbalanced, within-subject design: each participant worked with both visualizations. Two people were responsible for managing the experiment: a moderator explaining and leading the experiment and an observer taking notes. The visualizations were first shown to the participants printed in color on A2

		JFtp	JUnit	Stripes	Checkst.
	category	small	small	medium	medium
	# package	8	26	20	22
	# nodes	78	119	238	261
# edges	inheritance	40	63	143	207
	aggregation	66	35	131	69
	usage	38	251	614	479
	evolutionary coupling	380	76	262	586
	code clones	22	40	296	440
	semantic similarity	62	236	462	1014

Table 5.9: Statistics on the software systems used as datasets in the experiment.

paper. In the course of the experiment, the participants also used the interactive versions of the visualizations on a Windows 7 PC with a 23" LCD screen with full HD resolution, a mouse, and a keyboard. The source code files of the data sets were displayed in Notepad++ [45] on demand. Screen and voice were recorded with Camtasia Studio [91].

5.5.3 Participants

Eight participants, seven males and one female, volunteered for the experiment. They were between 26 and 45 years old; a color deficiency test showed that they did not have any color vision restrictions. All of them had at least a Master/Diploma degree in Computer Science or Mathematics, as well as programming experience between 3 and 30 years with a median of 11 years. Seven of them used the Java programming language regularly. The largest sizes of teams they developed software together were between two developers and more than ten developers. Two of them regularly worked with visual representations for software development such as VisDB or UML diagrams. Four participants were professional software developers from industry, the other four from academia.

5.5.4 Datasets

Adopted from a study on code coupling [29], six different types of couplings were considered:

- **inheritance**: extend and implement dependencies
- **aggregation**: usage of another class as the type of a class attribute
- **usage**: structural dependencies on method level
- **evolutionary coupling**: files that were changed together frequently in the past
- **code clones**: files that are reasonably covered by the same code clones
- **semantic similarity**: similar vocabulary used in identifiers and comments

These couplings were retrieved for a set of open source Java projects. Table 5.9 provides statistics on the sizes of the projects that the participants analyzed: JFtp and JUnit represent small projects, while Stripes and Checkstyle represent medium-size ones. The two small and the two medium-size projects were employed together pairwise in the experiment. An additional data set, JHotDraw, was used as a sample for tutorials and trials.

Order	Condition	Step	Avg Time
1st		tutorial	**4 min**
2nd + 3rd	PNLV	tutorial	4 min
		trial	5 min
		paper test	9 min
		tool test	12 min
			30 min
	IMMV	tutorial	4 min
		trial	4 min
		paper test	8 min
		tool test	10 min
			26 min
4th		questionnaire	**18 min**
		total time	**82 min**

Table 5.10: Experiment protocol and average time per step in the experiment.

5.5.5 Experiment Protocol

The experiment followed the protocol outlined in Table 5.10, which is a within-subject design with two experimental conditions: using PNLV and using IMMV. To counterbalance for biases such as learning and tiring effects, the order of the conditions was varied systematically across all participants (i.e., four participants began with PNLV and four with IMMV). Moreover, different data sets were used in the two conditions: half of the participants used the two small projects (JFtp and JUnit; four participants) and half of the participants used the two medium-size ones (Checkstyle and Stripes; four participants). The order of the data sets was also systematically varied.

At the beginning of the experiment, the moderator introduced the data sets and the different types of code couplings in a short tutorial using a slide show (average: 4 min). Then, the participants analyzed two different software projects of the same size in two different experimental conditions (PNLV: 30 min; IMMV: 26 min). The software projects and the visualizations were combined in a counter-balanced design, both for the combination of software systems and tools and their order of presentation. Allowing the participants to take as much time as they need for the tasks (like in a realistic scenario) resulted in different average times. Finally, the participants were asked to fill in a questionnaire consisting of general demographic questions and specific questions on the two visualizations (18 min). In total, the experiment took 82 minutes on average, including short breaks for switching, for instance, between paper test and tool test.

Each of the experimental conditions started with a training of the participants. First, a brief oral tutorial (power point slides) was given about the respective visualization technique (4 min). Then, the participants were asked to solve two simple tasks with the interactive visualization to familiarize themselves with the tool (PNLV: 5 min; IMMV: 4 min). The JHotDraw data set was used for the tutorials. After the training, the participant performed the first experimental task as a paper-based test (PNLV: 9 min; IMMV: 8 min): *"What interesting visual structures do you find in the visualization? Please mark them in the visualization, give them a number, and orally describe them."* Afterward, we switched to the interactive visualization of the same data set and the participants proceeded with the following task (PNLV: 12 min; IMMV: 10 min): *"Select three of the numbered visual structures, which appear to be interesting. For each selected structure, please use the interactive visualization to explore the structure. Explain your findings orally."*

The execution of the tasks was recorded using Camtasia Studio [91]. Both screen and voice were recorded to allow for a more detailed analysis.

At any time, the participants were allowed to ask questions. If a participant obviously had problems with a task or visualization, the moderator or observer provided help. No hard time limits were enforced in any step. To implement a thinking-aloud approach, the participants were asked and occasionally reminded to orally explain their thoughts and findings. The described experimental protocol was evaluated and improved with two additional test participants before the actual experiment was conducted.

5.6 Results

Following the outline of our study also for describing the results, we first present visual structures that were identified in the paper tests (Section 5.6.1); then, we report how the participants explored these structures in the interactive tool tests (Section 5.6.2), and finally analyze the answers provided in the questionnaires (Section 5.6.3).

5.6.1 Visual Structures (Paper Test)

The participants were asked to mark visual structures in the printed visualizations. We categorized these structures and found four different repeatedly used categories of structures for each visualization approach that almost all of the obtained visual structures can be classified into. In case of PNLV, these categories of visual structures match categories derived theoretically by Burch et al. [38] for a related visualization technique: *fan*, *beam*, *cross beam*, and *gap*. For IMMV, we named the four categories—according to their visual appearance—*line*, *diagonal cluster*, *off-diagonal cluster*, and *empty area*. Connecting these categories of visual structures to graph structures, we observed that they are encoding pairwise the same graph information. While Table 5.11 provides an overview of the outlined classification scheme, details on the graph structures and related visual structures are discussed in the following; an example detected by the participants illustrates each structure.

1. *High degree:* Vertices in the graph having a high degree of edges.

   PNLV: In PNLV, they appear as fan-like structures of links, either on the left side for outgoing edges or, more typically for a software project, on the right side for incoming edges.

   IMMV: In IMMV, the same graph structures are reflected in horizontal or vertical lines of equally colored sub-cells.

   For example, four of four participants who analyzed JFtp (two participants used PNLV, two others IMMV) detected a high in-degree for a small set of entities in the `framework` package for inheritance as well as aggregation (PNLV: 2/2, IMMV: 2/2). Though located in the same package, a closer investigation revealed that these were different entities with respect to inheritance than with respect to aggregation.

2. *Within-package edges:* Groups of edges connecting vertices of the same package were also observed in both visualizations.

   PNLV: In PNLV, the participants marked edges forming a beam or an x-shape where the source and destination classes of these edges are included in the same package.

   IMMV: In IMMV, equivalent structures appear as blocks of cells on the main diagonal of the matrix, which have the same or a similar combination of sub-cells.

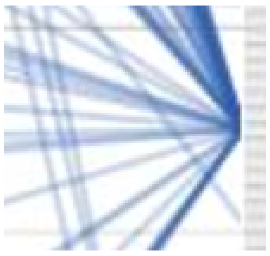
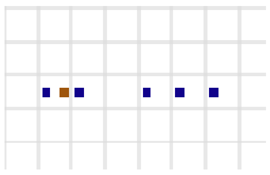
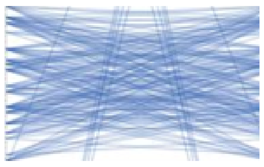
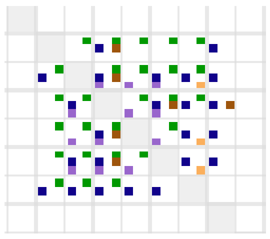
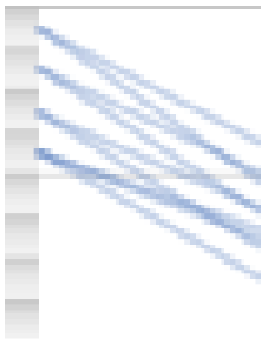
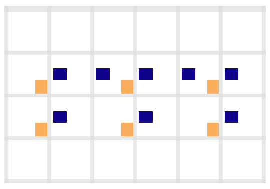
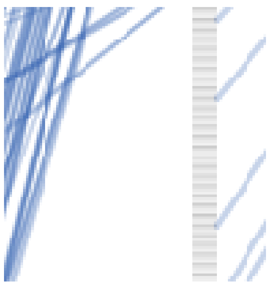
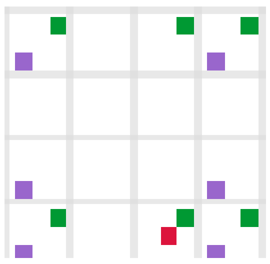
Graph Structure	Visual Structure	
	PNLV	IMMV
high degree	fan (7) 	line (8) 
within-package edges	beam (8) 	diagonal cluster (8) 
cross-package edges	cross beam (6) 	off-diagonal cluster (7) 
no edges	gap (2) 	empty area (2) 

Table 5.11: Mapping of visual structures to graph structures; numbers of participants in parentheses who identified the visual structure using the respective visualization.

A typical example of a set of within-package edges which was detected by the participants is the `tag` package in Stripes: many entities are connected by code clone couplings to other entities of the same package (PNLV: 2/2, IMMV: 2/2).

3. *Cross-package edges:* If a group of edges does not connect the vertices of the same package but of two different packages in the same direction.

   PNLV: PNLV shows a beam or an x-shaped structure that crosses the borders of packages.

   IMMV: In IMMV, these structures can be detected as blocks of cells not located on the main diagonal.

   For instance, based on evolutionary coupling, the `theories` package of JUnit is connected by a number of cross-package edges to one of the `runners` packages (PNLV: 2/2, IMMV: 0/2).

4. *No edges:* Vertices that do not have any outgoing or incoming edges (to all or a subset of other vertices) form another graph structure.

   PNLV: In PNLV, these vertices are represented as nodes on the left or right side of a diagram that do not have any links attached (in a specific range) and form kinds of gaps in the linear list of nodes.

   IMMV: In IMMV, empty areas that do not contain any colored cells (in general or with respect to a specific color) hint at the same graph structure.

   Entities without edges can be found frequently, but were only rarely marked by the participants as distinct structures. For instance, one participant detected a larger set of entities within the `coding` package with PNLV; another participant found with IMMV that the `framework` package of JFtp lacks within-package edges in general.

Table 5.11 also summarizes how many participants identified the respective visual structures in PNLV and in IMMV. Contrasting the two visual structures of each pair, it shows that both were identified by approximately the same number of participants in the two visualizations. A large number of participants identified a vertices with a high degree (PNLV: 7, IMMV: 8), groups of within-package edges (PNLV: 8, IMMV: 8), and groups of cross-package edges (PNLV: 6, IMMV: 7), while only few marked the visual structures classified as *no edges* (PNLV: 2, IMMV: 2).

Since the visualizations discern between multiple types of edges, the identification of visual structures is also related to visually comparing the different types. In PNLV, these types are viewed side by side; hence, participants needed to mark similar structures across different columns to indicate a comparison: three participants linked fan structures, two participants beam structures, and one participant contrasted a beam structure to a gap structure. In contrast, the comparison of types of edges is inherent in IMMV because the different types are shown within the same cells and can neither be perceived nor marked independently of each other. Hence not surprising, more participants identified those multi-type structures in IMMV than in PNLV; in particular, eight participants marked diagonal clusters consisting of multiple types, six participants line structures, and three off-diagonal clusters.

The participants identified the visual structures in a specific sequence. In order to analyze trends, we split the sequence into half (similar to a median split) based on the sequential number that was assigned in the experiments (in cases of an odd number of identified structures, we excluded the one in the middle). Figure 5.5 reports the results of this analysis split by

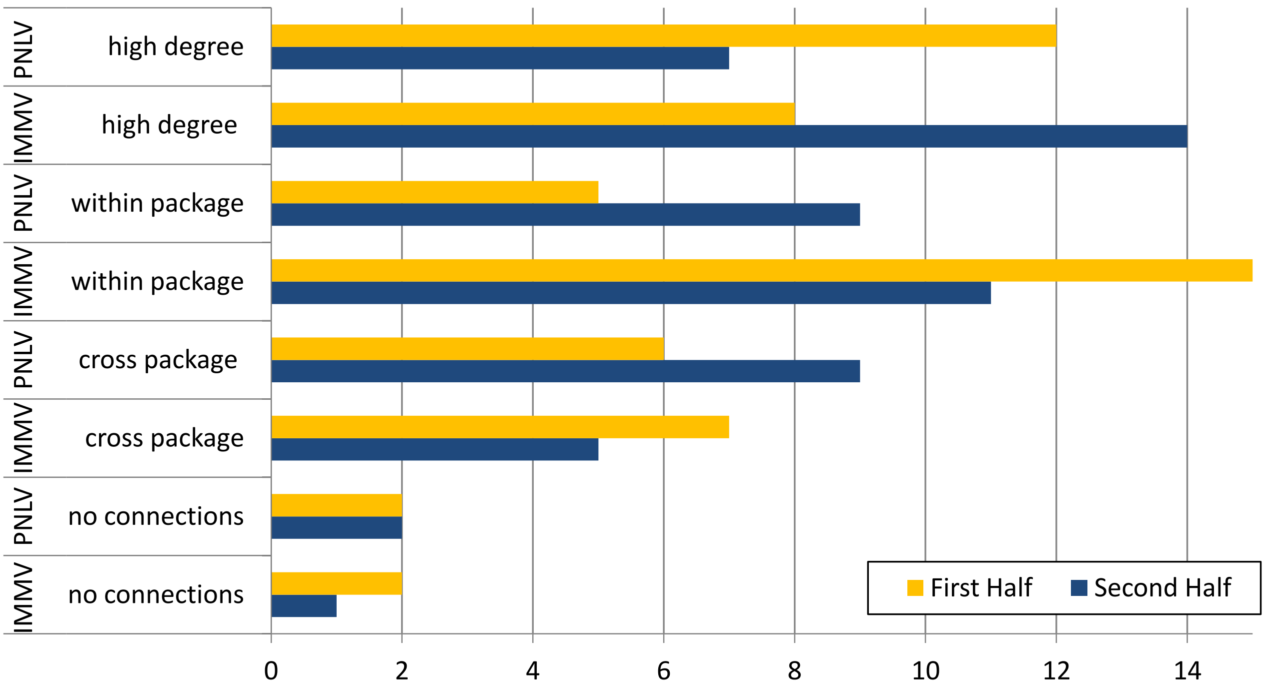


Figure 5.5: Numbers of visual structures identified by the participants in the first and second half of their sequence of structures split by visualization and graph structure.

visualization and graph structure. While we do not observe any fundamental differences, a few trends are notable: vertices with a high degree tend to be more frequently found in the first half than in the second in PNLV while it is the other way around for IMMV; within-package and cross-package structures, in contrast, were found earlier by trend with IMMV than PNLV. These trends might be connected to small differences in difficulties for finding certain structures, to training and experience required, or effects of reading order.

5.6.2 Interactive Exploration (Tool Test)

After analyzing the visualizations on paper, the participants explored three of the identified visual structures using the respective interactive tool. The insights they were able to gain identify specific tasks related to software engineering that can be targeted with the help of the visualizations. We were able to categorize each interactive exploration of a visual structure as one of three software engineering tasks and connected these tasks to analyzed visual structures and applied high-level interaction techniques. Please note that multiple structures or interactions were assigned to a single instance of an interactive exploration (in two cases we were not able to unambiguously assign a visual structure). Moreover, we recorded whether a participant compared different types of couplings during the interactive exploration. Split by the identified software engineering tasks, the results of this classification process are summarized in Table 5.12 and reported in the following.

Find a central class/interface: This task, most frequently investigated by the participants (PNLV: 11; IMMV: 9), is closely related to code entities having a high in-degree for structural dependencies such as inheritance, aggregation, and usage. As discussed above and confirmed by Table 5.12, these can be detected by looking for fan-in structures in PNLV (11 of 11 cases) or vertical lines in IMMV (8 of 9 cases). The interaction strategy typically applied was to first select the respective class (PNLV: 11 of 11 cases; IMMV: 8 of 9 cases) and then to open and read the source code of the class (PNLV: 7 of 11 cases; IMMV: 7 of 9 cases); in some instances, interpreting the names of the related classes and

Task	PNLV	Freq.	IMMV	Freq.	Total
find a central class/interface	*structure:* fan (11) *interaction:* select class (11), read source code (7), interpret names (4)	11 [8]	*structure:* line (8) *interaction:* select class (8), read source code (7), interpret names (4)	9 [3]	20 [11]
understand a package	*structure:* beam (8), cross beam (2) *interaction:* read source code (7), select class (6), select package (5), interpret names (5), compare source code (4)	10 [8]	*structure:* diagonal cluster (7), empty area (2) *interaction:* interpret the names (5), read source code (4), select package (3), select class (2), compare source code (2)	7 [6]	17 [14]
identify a high-level coupling	*structure:* cross beam (2) *interaction:* read source code (2)	2 [1]	*structure:* off-diagonal cluster (4), empty area (2) *interaction:* read source code (5), interpret names (4), select cell (3), select class (2), select package (2)	7 [7]	9 [8]

Table 5.12: Software engineering tasks addressed by the participants through the use of the interactive visualization tools; frequency values refer to the number of investigated structures summed for all participants, in square brackets the frequency involving the comparison of multiple types of couplings; structures and interactions are listed if they occurred at least two times, the exact frequency is provided in parentheses.

interfaces was even already sufficient or made opening the editor superfluous (PNLV: 4 of 11 cases; IMMV: 4 of 9 cases). Depending on the type of coupling, the role of the central class is different: a high inheritance in-degree hints at a central code entity in the inheritance hierarchy while a high usage could identify an important data class. In 11 of the 20 cases, the participants also compared different types of couplings for the selected code entity.

Understand a package: Another frequent task that the participants addressed was trying to understand the purpose and characteristics of a particular package (PNLV: 10; IMMV: 7). Often, a set of within-package edges hinted at a particularly interesting package (PNLV: 8 of 10 cases; IMMV: 7 of 7 cases), but also other visual structures occasionally provided a starting point for exploring a package (PNLV: cross beam, IMMV: empty area). The interactions for exploring a package were diverse: in at least half of the cases, the participants read the source code (PNLV and IMMV), interpreted the names of the code entities (PNLV and IMMV), or selected a class or package (PNLV). Sometimes, also source code files were directly compared by quickly switching between the tabs of the editor (PNLV: 4 of 10 cases; IMMV: 2 of 7 cases). Understanding a package typically involved analyzing multiple types of coupling (14 of 17 cases).

Identify a high-level coupling: As the least frequent task among the three, the participants analyzed groups of couplings that are similar and together form a kind of high-level coupling, usually between two different packages (PNLV: 2; IMMV 7). Cross-beams (PNLV: 2 of 2 cases) or off-diagonal clusters (IMMV: 4 of 7 cases) usually served as a visual indicator for noteworthy high-level couplings. To further explore these structures, the participants often studied the source code of the connected code entities (PNLV: 2

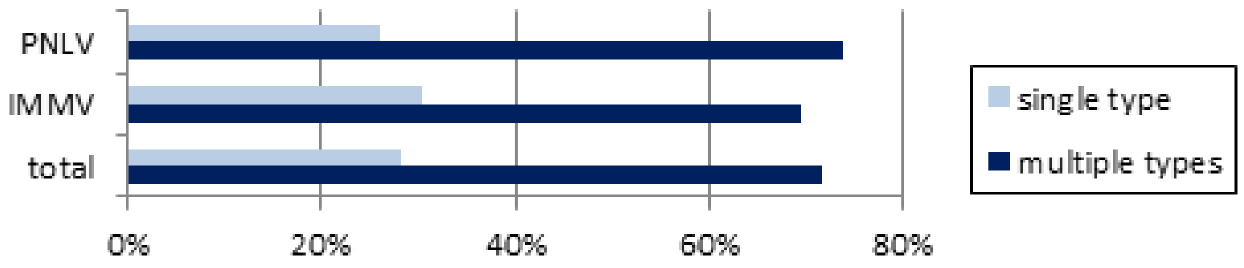


Figure 5.6: Percentage of explored visual structures that involve the comparison of multiple types of edges.

of 2 cases; IMMV: 5 of 7 cases) and applied other high-level interactions in IMMV. The detected high-level couplings were usually impacted by two or more different types of couplings (8 of 9 cases).

In general, the experiment shows that the participants frequently used the possibility to compare different types of edges: As Figure 5.6 reports, for 72% of the analyzed structures, the participants compared two or more different types of edges; this also includes structures that were only marked in one type of coupling in the paper test, but extended to multiple types in the interactive test. Contrasting the two visualization approaches again does not reveal any larger difference: the percentage of cases that involved comparing multiple types is similar (PNLV: 74%; IMMV: 70%). This clearly relativizes that we found less visual structures including a comparison of types based on the paper test in PNVL than in IMMV and is remarkable because, as discussed above, it requires an extra effort to do a visual comparison in PNLV.

5.6.3 Questionnaire

Asked for their preference and the usefulness of the visualization, PNLV and IMMV were rated about equal: While four participants preferred IMMV, three liked PNLV more (one participant could not decide). Moreover, with respect to usefulness, both tools received nearly the same average answers (PNLV: 2.75; IMMV 2.88; scale from strongly agree (1) to strongly disagree (5) that it is useful). Motivating their decision on the preference, the participants provided diverse reasons such as good overview in PNLV and IMMV, familiarity with the node-link paradigm, but also problems with visual clutter in PNLV. For example, one participant stated that "PNLV has good overview with respect to different dependency types and IMMV has good overview with respect to the package structure". Another participant used PNLV and IMMV on small data sets. He preferred PNLV: "Based on presented examples it seems to be better organized, easier to navigate, or is visually more appealing". However, using different data set sizes in our experiment decreases such effects that are based on the size.

Regarding software engineering, the participants see the main area of application of the two tools in program comprehension (PNLV: 4 times; IMMV: 4 times) and improving the architecture or design of software systems (PNLV: 4 times, IMMV: 3 times). For PNLV, the participants listed finding central classes as the most interesting insight they gained from the visualization (4 times). For IMMV, the picture is not as clear: participants mentioned different insights, none more than once. When asked if they would like to use the visualizations in their daily software development work, only three participants gave positive feedback for PNLV and one for IMMV. Explaining their reluctance, the participants provided different reasons, among them that the visualizations do not show the right information, that the implementations are not yet ready for practical application, and that the participants themselves are unfamiliar with visualization. Possibly related, only two of the participants declared that they regularly use visual representations for software development.

The participants also had the possibility to propose enhancement for the two tools: For the PNLV tool, the most frequently mentioned missing feature was zooming (4 times)—the original tool actually had a zooming functionality, but it was deactivated for the experiment to make the comparison fair because zooming was not implemented for the IMMV tool. With respect to the IMMV tool, zooming was only mentioned once; improving the selection mechanism and filtering types of edges was proposed by two participants each (the latter was also already implemented but deactivated in the experiment).

5.7 Discussion

The limitations of the explorative study as well as implications of the results on visualization and on software engineering are discussed in the following.

5.7.1 Threats to Validity

By systematically varying the order of the visualizations and data sets, we counterbalanced for possible biases such as learning and tiring effects. We also tried to adjust the two approaches as far as possible by optimizing the readability of both, by deactivating features that were not implemented in one of the approaches, and by using paper versions in parts of the experiment to counterbalance for different interaction techniques. But still, the design or the tool implementations might introduce a bias towards one of the two visualization approaches.

The within-subject design allowed the participants to review and contrast the two visualization approaches in the questionnaire, which provided valuable feedback. But, at the same time, this design biases participants—no matter which visualizations they saw first—to look for similar things in the second visualization. We tried to circumvent parts of the problem by providing different, nevertheless, nearly equally sized, data sets for the two visualizations; but still, the participants might be influenced by this previous experience. The effect could be that the two visualizations showed some more similarities in this study than they would have shown in an equivalent between-subject study.

The quantitative parts of the results have to be interpreted with care because they rely on a small number of participants and are not backed by inference statistics—random effects might have reasonably influenced the numbers. Despite we did not interpret small differences, also our results based on larger difference or similarities in numbers should only be treated as preliminary results that need further quantitative evaluation. Moreover, our study design did not allow for measuring time and accuracy of the analyses the participants performed with the visualizations; the visualization approaches, however, might have shown relevant differences on this level.

Although the transferability of the results is limited by the narrow area of application and the specific data sets, some features of the study also foster transferability: Through considering multiple types of code coupling, the data set includes edge types with different characteristics such as directed and undirected types, or dense and sparse types. For instance, structural dependencies are usually scale-free networks [77]. In contrast, evolutionary couplings usually form denser graphs having many cliques. Moreover, the visual structures relate to general graph structures, hence, are independent of the application domain.

5.7.2 Implications on Visualization

Matrix and node-link are two very different metaphors for representing graphs. Also the visual comparison as realized in the two studied visualization approaches is reasonably different: applying the taxonomy of Gleicher et al. [59], PNLV is based on *juxtaposition*, while IMMV employs a form of *superposition* (i.e., overlay). But despite those fundamental differences in the visualization approach, both techniques seem to be similarly suitable for investigating multiple types of edges in graphs:

- A comparable number of participants identified the same graph structures (Table 5.11).
- The participants were able to address the same task in the interactive exploration and applied similar interaction strategies (Table 5.12).
- A comparable number of interactively explored structures involved the comparison of multiple types of edges (Figure 5.6).
- In the questionnaire, the participants rated both approaches as equally useful and their personal preference was balanced.

Only small differences are observed between the approaches: In the paper-based test, participants more frequently marked multi-type visual structures in IMMV—maybe due to the superposition approach to visual comparison. In the interactive versions, the selection mechanism seemed to be more important in PNLV. And in the questionnaire, the participants criticized edge clutter in PNLV.

5.7.3 Implications on Software Engineering

The identified task that the participants mainly addressed (Table 5.12) can all be considered as belonging to the application of program comprehension, that is, understanding a software system as required for being able to extend, test, or maintain the system. This matches with the opinion of the participants captured in the questionnaire. Some participants propose also to use the visualizations for improving the architecture or design of systems—this application is not directly reflected in the recorded interactive usage scenarios and will only be possible to capture if the participants already know the analyzed system in detail before the experiment.

The visual distinction of multiple types of couplings seems to add value to solving these tasks: First, similar visual structures for different types of edges refer to different software engineering concepts. Second, the participants frequently applied a visual comparison between two or more types and gained interesting insights from these comparisons—this appears to be particularly important for understanding packages and identifying high-level dependencies.

The connection to the code, that is, the raw data in this experiment, turned out to be very important and was frequently used: while the visualization served as an instrument to navigate, to raise hypotheses, and to answer simple questions, the source code needed to be studied to gain a deeper knowledge about the system, to understand certain couplings, to check the hypotheses, and to answer more complex questions. One participant opened several source documents in Notepad++ [45] to compare these documents. A mechanism for directly opening additional documents in adjacent views would be benefical for supporting these comparisons. This holds, e.g., for source code related to the different classes connected by an undirected relation (e.g., code clone).

5.8 Related Work

The scalable visualization of graphs is an established area of research [98]; within this area, some approaches—like the studied PNLV and IMMV—particularly focus on visually discerning multiple types of edges: Based on node-link diagrams, for instance, Erdemir et al. [52] provided a visualization that uses visual attributes of the links (e.g., color, style, strength) to reflect different attributes of edges; when there exist multiple types of edges between two nodes, only the link having the highest priority is shown. Pretorius and van Wijk [80] introduce special nodes for representing different types of edges—a link goes from the source node through the edge-type node to the target node. Also related are node-link-based graph comparison visualizations where two or more graphs are juxtaposed [13], stacked [34], or contrasted as a visual *diff* [16]. In a matrix-based representation, an alternative to splitting the cells is only using colors for discerning different types of edges [53, 28, 30], which, however, only scales to a very limited number of edge types.

Related to discerning multiple types of edges, *dynamic graphs* discern multiple points in time for vertices as well as for edges. Hence, visualizing dynamic graphs is similar to visually comparing different types of edges. Animated node-link diagrams [43, 54], the standard approach to dynamic graph visualization, however, only allow for comparing edges in directly consecutive time steps. In contrast, recent timeline-based approaches support the comparison of edges across larger time spans: For instance, the *Parallel Edge Splatting* technique [38] uses juxtaposed node-link diagrams and can be considered as a variant of PNLV. But also matrices are used in this context such as the *Pixel Oriented Matrix* [86], which splits matrix cells into sub-cells similar to IMMV.

Node-link and matrix graph visualization have been already contrasted to each other, but only for single types of edges: Ghoniem et al. [57, 58] evaluated these using seven simple, generic tasks as well as different sizes and different densities of random undirected graphs. Their results indicate that node-link diagrams are more suitable for most tasks for smaller graphs while matrix visualizations are more readable for large ones. An exception forms the task of finding paths where node-link diagrams perform better or at least comparable. Keller et al. [69] extended this work by performing a similar study, still with simple, domain-independent tasks but on non-random data: they largely confirmed the previous results but pointed out that *"depending on the model, and even on personal preference, either representation can be advantageous"*.

In contrast to these studies, we addressed a more specific visualization scenario in a more complex, realistic application also including the distinction of multiple types of edges. The results of our study extend the previous studies, in particular, with respect to performing higher-level tasks such as finding graph structures: although low-level tasks showed quite different characteristics for node-link and matrix, the high-level tasks we studied did not provide any considerable differences between the contrasted approaches.

5.9 Conclusions

The conducted explorative user study evaluated two approaches for comparing different types of edges in graph diagrams. In a realistic scenario, eight participants analyzed code couplings of software systems both based on static images as well as on interactive visualizations in a 82 minutes (on average) within-subject lab experiment. The choice between the two visualization approaches in this application mainly seems to be a matter of personal preference as the two approaches—though based on opposing paradigms—did not show any basic differences in our study: the participants were able to identify equivalent structures in the presented graphs, addressed the same software engineering tasks with the interactive tools, and rated the usefulness of the approaches alike. In particular, vertices with a high degree, groups of similar edges within packages and between different packages were detected by most participants with both approaches. With respect to the targeted area of application in software engineering, the visualizations seem to be most suitable for program comprehension, namely, for finding central classes and interfaces, for understanding packages, and for identifying high-level dependencies—comparing multiple types of couplings seems to have helped the participants in these tasks.

Chapter 6

Conclusion

This part described the development of IMMV and the determination of pattern in IMMV compared to PNLV using an explorative user study. As IMMV and PNLV produce visual patterns that can be meaningful by contrasting multiple types of edges (relations) in the view of software engineers, four categories of visual patterns in IMMV compared to PNLV were determined using real data sets in the extended version of the explorative user study [8] using a counterbalanced within-subject design. The categories of determined structures included: *High degree, Within-package edges, Cross-package edges, No edges.* Additionally, the two tools tend to be beneficial for software comprehension. Based on class views, finding differences and similarities between multiple relations types seems to be the shared mechanism followed in general for visual comparison in the two tools.

Part II

Topological Visualization of Directed Graphs

Chapter 7

Preamble

Visual structures formed by IMMV and PNLV visualizations were determined and classified in the previous part as four categories: *High degree, Within-package edges, Cross-package edges, No edges*. In addition to these structures, other structures that are interesting for software engineers as cycles and hierarchical structures need additional visualizations to determine them and to investigate them. This part covers the extended version of the work introduced by Abuthawabeh and Zeckzer [10]. Their work focuses on enhancing the decomposition and drawing process of directed graphs using topological visualization. This approach was used to visualize the JFtp software data set [29] and was compared with two existing approaches.

The extended version enriched the introduction (Chapter 8) and the background and problem statement(Chapter 9) with additional illustrating examples. It shows the images of the previous work summarized in Chapter 10. Introducing new algorithms chapter (Chapter 14), it adds the complete algorithms of the decomposition and hierarchy construction process to Section 14.1 and presents the complete algorithms of the drawing process to Section 14.2 explaining them using additional examples and showing Class diagrams and Call graphs of all algorithms used. Chapter 14 includes data structures and their time complexity that were used in implementation (Section 14.3), time complexity and space complexity of the decomposition algorithms (Section 14.4) and drawing algorithms (Section 14.5). Chapter 13 is also modified by adding Section 13.1, Section 13.2, and Section 13.3.

Chapter 8

Introduction

Important information in application areas like Software Engineering or Bio-Informatics and Biology is encoded in directed graphs. Typically, these are represented using node-link diagrams (Figure 8.1). Finding important information necessitates an appropriate visualization of these graphs. Many approaches exist for special sub-problems of directed as well as of undirected graphs as summarized in the "Handbook of Graph Drawing and Visualization" [89] and outlined in the previous work chapter.

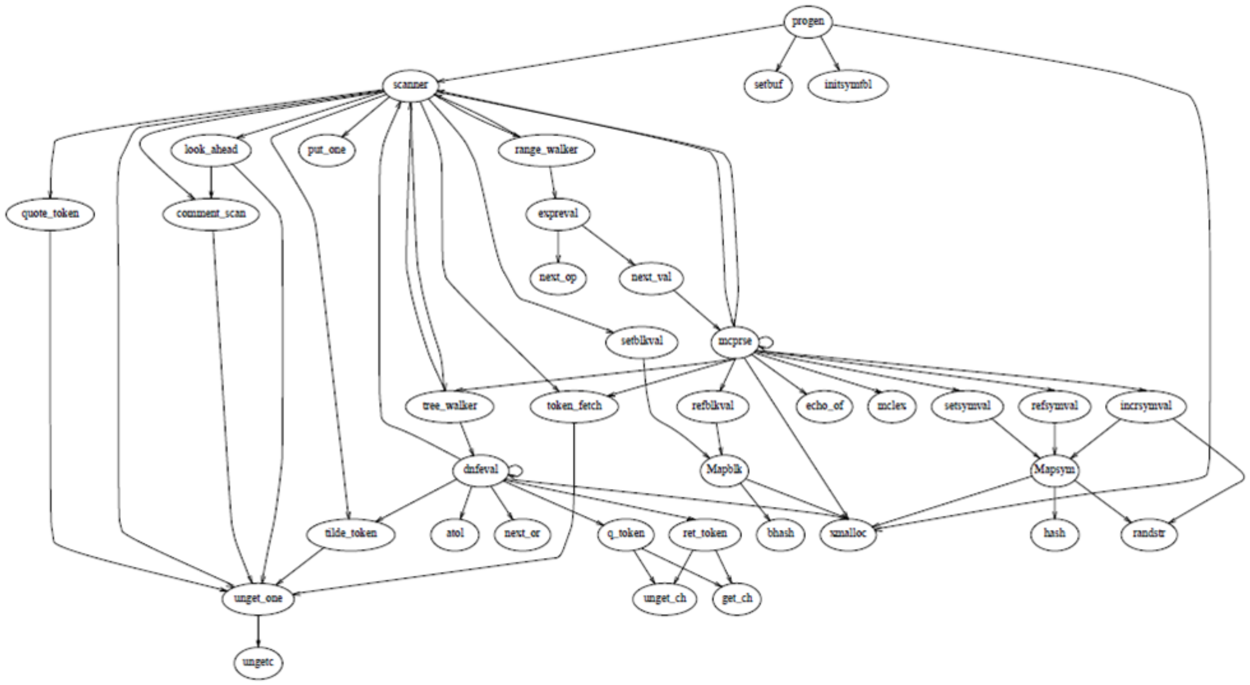
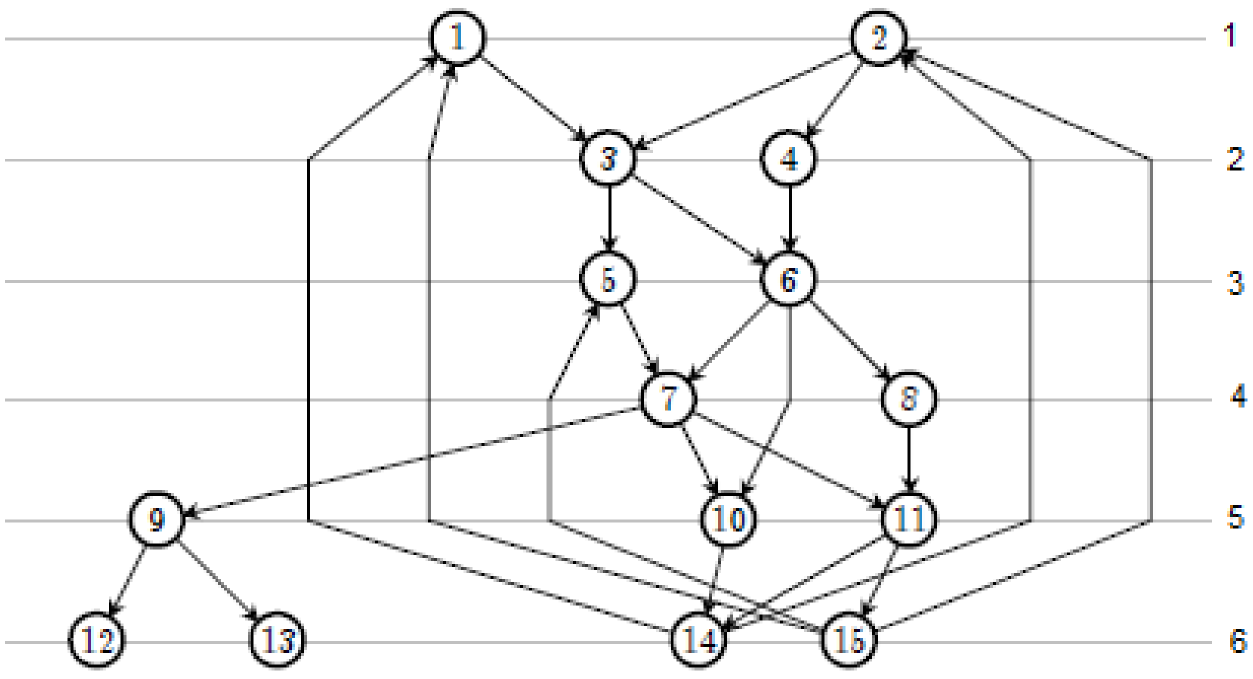
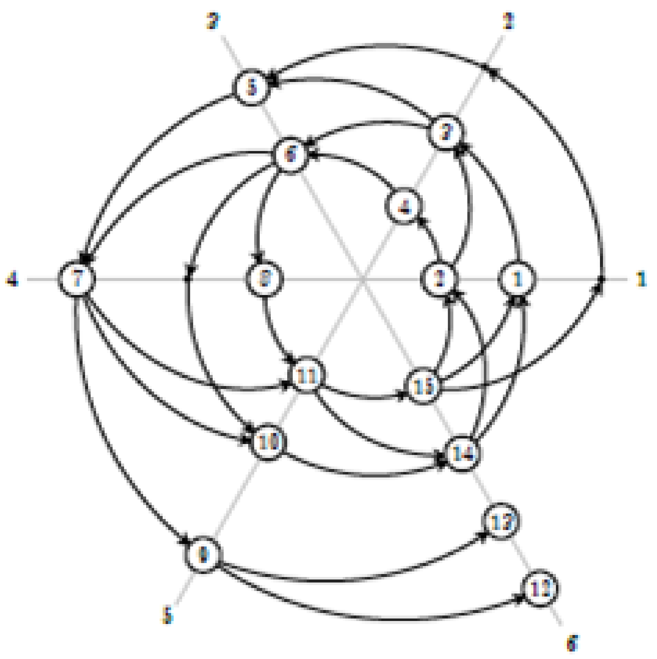


Figure 8.1: Call Graph (#nodes=38, #edges=58), (figure courtesy of [56]).

For arbitrary directed graphs, two approaches are described by the Handbook of Graph Drawing and Visualization [89]. The first uses the Sugiyama algorithm for directed acyclic graphs [87] (Figure 8.2(a)). If the graph is cyclic a minimal number of edges is reversed to make it acyclic. Then, a layered layout of the graph is determined using the Sugiyama algorithm. In the final drawing, reversed edges are again drawn in their original direction. The second approach proposed by Bachmaier et al. [22] uses a cyclic layout with cyclically arranged layers instead (Figure 8.2(b)). However, even acyclic parts are drawn cyclically in this approach.



(a) Sugiyama layout example [87] , (figure courtesy of [89]



(b) Cyclic level layout example [22], (figure courtesy of [89]

Figure 8.2: Two examples of graph drawing approaches for arbitrary directed graphs

In the aforementioned application areas, it is important to distinguish between cyclic and acyclic (parts of) directed graphs. In software engineering, cyclic parts might be critical due to unwanted cyclic dependencies while acyclic parts describe 'normal' dependencies. In Bio-Informatics and Biology on the other hand, cycles are important structural features that need to be easily recognizable. In these domains, several approaches try to handle this problem by drawing cycles using a circular layout, adding edges as straight lines inside circles, which may cause many crossing, and by applying different layouts for other features [31, 32, 44, 72].

Dividing the graph into cyclic and acyclic parts improves its understandability by domain experts like Software Engineers and Bio-Informaticians. Therefore, a topological approach is proposed whose goal is to divide the directed graph into three parts: cycles, directed acyclic graphs (DAGs), and trees. These are then drawn using an area-aware version of the improved Walker's algorithm for layouting trees [37], an area-aware version of the Sugiyama algorithm for layouting DAGs [87], and Bachmaier's cyclic layout [22]. The approach is similar to the ones proposed by Archambault et al. [17] for undirected graphs and by AlTarawneh et al. [12] based on strongly connected components for directed graphs. However, the notion of cycle differs from the definition of strongly connected components as used by AlTarawneh et al. [12]. Moreover, some restrictions of the latter approach are overcome. Applying the algorithm to large graphs, constructs a topology-based, two-level hierarchy that allows to interactively handle large graphs similar to the approach by Archambault et al. [19].

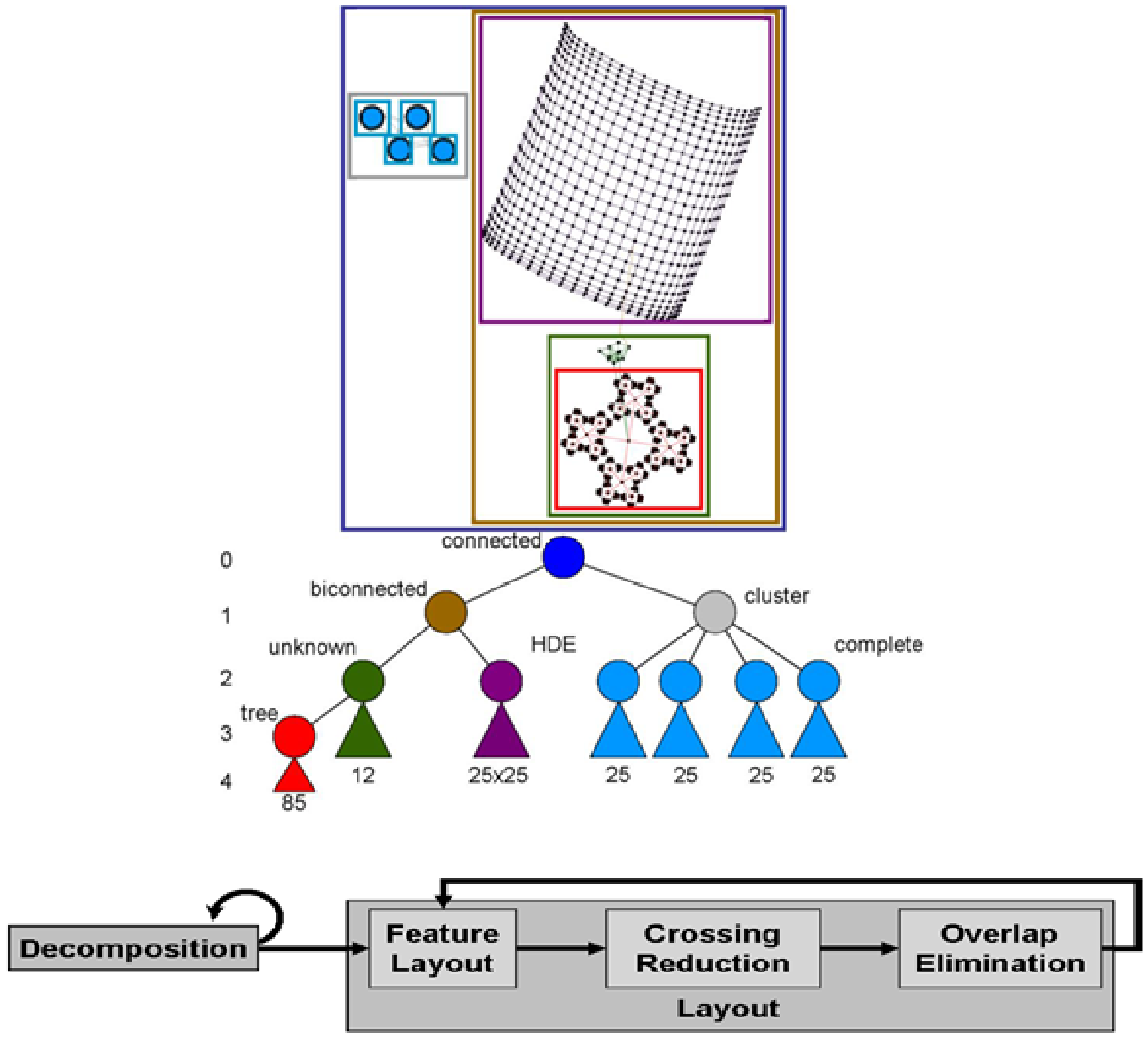


Figure 8.3: Topological visualization for undirected graphs, (figures courtesy of [17]).

The contributions presented in this part are:

- A new decomposition process for directed graphs including
  - an algorithm for finding maximal non-trivial cyclic subgraphs, and
  - an algorithm for detecting trees and DAGs that are connected to these maximal non-trivial cyclic subgraphs.

Chapter 9

Background and Problem Statement

9.1 Background

In this study, directed graphs are considered.

Let $V = \{v_1, \ldots, v_n\}$ be a finite set of n *vertices*. Let $E \subseteq V \times V$ be a finite set of e *directed edges*, i.e., $\forall v_i, v_j \in V : (v_i, v_j) \in E \wedge (v_j, v_i) \in E \rightarrow (v_i, v_j) \neq (v_j, v_i)$. Then, a *directed graph* is defined as the tuple $G = (V, E)$. The *in-degree* of a vertex $v \in V$ is the number of distinct edges $(v_i, v) \in E$. Analogously, the *out-degree* of a vertex $v \in V$ is the number of distinct edges $(v, v_j) \in E$. The directed graph $G' = (V', E')$, $V' \subseteq V$, $E' \subseteq E \cap (V' \times V')$ is called *subgraph* of G. If $E' = E \cap (V' \times V')$, then G' is called the subgraph of G *induced* by V'. A *path* in G is a sequence on m different vertices $(v_1, \ldots, v_m), v_i \in V$, such that $\forall v_1, \ldots, v_{m-1} \in P : (v_i, v_{i+1}) \in E$. We identify the sequence of edges $(e_1 = (v_1, v_2), \ldots, e_{m-1} = (v_{m-1}, v_m))$ with the path P and say that these edges *belong to* P. A path P is called a *cycle* iff $v_m = v_1$. A graph without cycles is called *acyclic* or more precisely *directed acyclic graph (DAG)*. Let H be the *hierarchy* associated with a graph G such that the leaves of H are the vertices V of G. A *level in the hierarchy* contains all elements of the hierarchy having the same distance to the root in the hierarchy.

A *connected component* of an undirected graph is defined as a maximal subgraph (V', E') of an undirected graph such that $\forall v_i, v_j \in V' \exists P = (v_i, \ldots, v_j)$. That means, there is a path between each pair of nodes in the subgraph. Please note, that all edges are undirected and thus, there is a path from v_i to v_j iff these is a path from v_j to v_i. A *weakly-connected component (wCC)* of a directed graph is the subgraph induced by the vertices of the connected component of the undirected graph obtained by 'forgetting' the direction of the edges of the original directed graph. That is, first all edges of the directed graph are considered to be undirected. Then, duplicates are removed (each double edge in the directed graph is represented by a single edge in the undirected graph). The resulting undirected graph is split into connected components. The vertices of these connected components then induce a subgraph of the original directed graph, each: the weakly-connected components of the directed graph.

The approach presented here is based on defining the notion of *non-trivial cyclic subgraphs (ntCS)*. This is motivated by the following observation. Strongly connected components (SCC) are defined as a subgraph $S = (V_S, E_S)$ of G induced by $V_S \subseteq V$, such that each vertex is reachable by each other vertex by a path P: $\forall v_i, v_j \in V_S : v_i \neq v_j \rightarrow \exists P = (v_i, \ldots v_j)$. However, in application areas like software engineering, a distinction between trivial cycles and non-trivial cyclic subgraphs is appropriate. Trivial cycles are cycles between two vertices. Let $v_1, v_2 \in V$ be two vertices in G and $e_1 = (v_1, v_2), e_2 = (v_2, v_1) \in E$ the edges between them. We call e_2 a *back edge* of e_1 and the pair (e_1, e_2) a *double edge*. Then, the subgraph $G' = (\{v_1, v_2\}, \{e_1, e_2\})$ is a strongly connected component, and the paths (v_1, v_2, v_1) and (v_2, v_1, v_2) are *trivial cycles*.

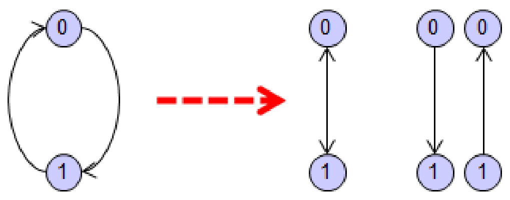
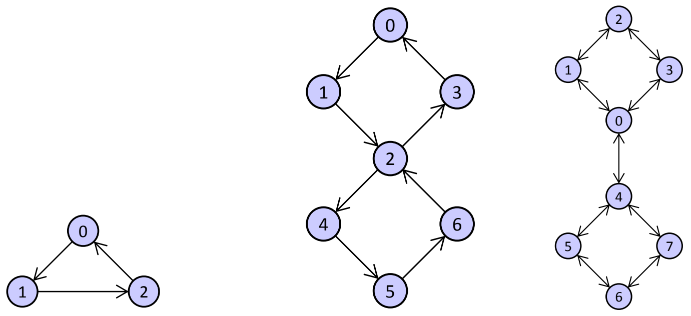


Figure 9.1: Trivial Cycle.



(a) C1=$\{0, 1, 2\}$ Single cycle

(b) C1=$\{0, 1, 2, 3, 4, 5, 6\}$ Two cycles sharing one node will be combined

(c) C1=$\{0, 1, 2, 3\}$ C2=$\{4, 5, 6, 7\}$ Two cycles

Figure 9.2: Three examples of non-trivial cyclic subgraphs (ntCSs)

We call these cycles trivial, as in a node-link diagram they can by depicted by a line with two arrows (Figure 9.1). No cyclic depiction is necessary.

Non-trivial cyclic subgraphs (ntCS) are defined as follows. For each strongly connected component, all double edges (e_1, e_2) are removed, if e_1 and e_2 are only belonging to trivial cycles, respectively. The remaining isolated subgraphs are still strongly connected, as only the 'bridges' between them were removed. These subgraphs are the non-trivial cyclic subgraphs. All examples in Figure 9.2 are SCCs. At the same time, they are all ntCSs, except the one shown in Figure 9.2(c), which can be split into two ntCSs (induced by the vertex sets $V_1 = \{0, 1, 2, 3\}$ and $V_2 = \{4, 5, 6, 7\}$ and a non-trivial tree (see below) consisting of two nodes $V_3 = \{0, 4\}$ and the double edge $((0, 4), (4, 0))$. The non-trivial tree connects the two ntCSs. Please note if we take this double edge (non-trivial tree) and we remove one of its edges then cycle breaks. While if we take any of the other double edges and we remove one of its edges then cycle remains. Considering call graphs in software engineering, trivial cycles are common: two classes can both call methods of the other class, respectively. However, ntCSs are unwanted in general, as they make understanding of the behavior of a program more difficult [75].

The introduction of ntCSs makes it necessary to adapt the definition of trees and DAGs. We define (trivial, non-trivial) DAGs and (trivial, non-trivial) (up, down) trees as follows. A connected subgraph V' is a *trivial down tree*, if it does not contain any double edges or cycles, and if it contains only one node with in-degree 0 called *root node* v_r or simply root, and $\forall v_i \in V' : v_i \neq v_r \rightarrow \exists P = (v_r, \ldots, v_i)$, and P is unique (Figure 9.3(a)). A subgraph is a *non-trivial down tree*, if it does not contain ntCSs, and it can be transformed into a trivial down tree by removing all back edges of double edges.Figure 9.3(b) shows an example. The graph consists of one strongly connected component without ntCSs and three double edges. Removing the back edges $\{(1, 0), (2, 1), (3, 1)\}$ from the graph yields a subgraph that is a trivial down tree. A *down tree* is either a trivial or a non-trivial down tree. A subgraph is a *(trivial, non-trivial) up*

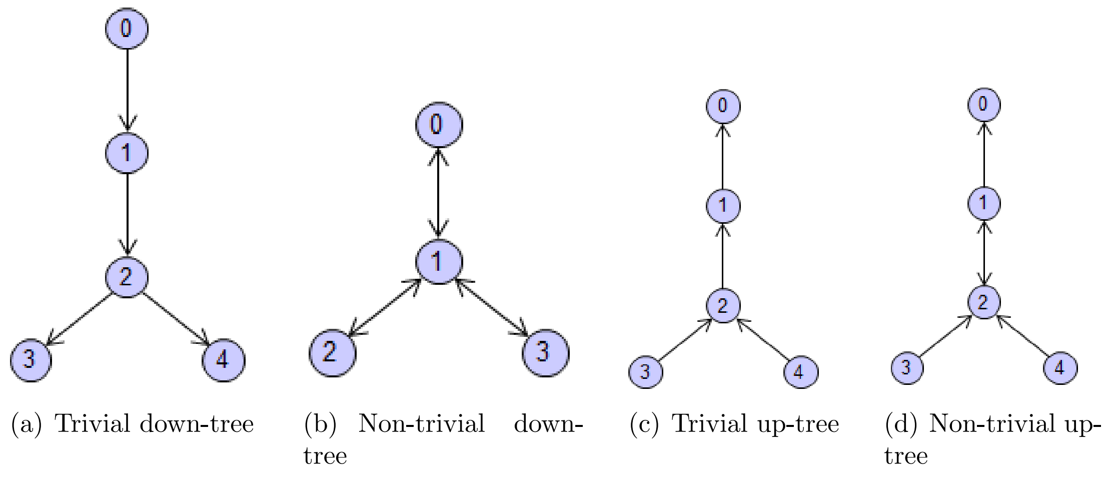

(a) Trivial down-tree (b) Non-trivial down-tree (c) Trivial up-tree (d) Non-trivial up-tree

Figure 9.3: The different types of trees considered.

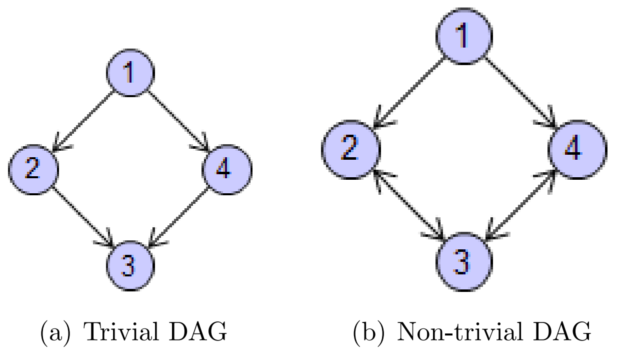

(a) Trivial DAG (b) Non-trivial DAG

Figure 9.4: The different types of DAGs considered.

tree, if after reversing all edges, it is a (trivial, non-trivial) down tree. A *(trivial, non-trivial) tree* is either a (trivial, non-trivial) up tree or a (trivial, non-trivial) down tree (Figure 9.3). Accordingly, a subgraph is a *trivial DAG*, if it does not contain any double edges or cycles, and is not a trivial tree (Figure 9.4(a)). Accordingly, a subgraph is a *non-trivial DAG*, if it does not contain ntCSs and it is neither a non-trivial down tree nor a non-trivial up tree (Figure 9.4(b)). A *DAG* is either a trivial DAG or a non-trivial DAG. The difference between trivial and non-trivial tree and DAGs, the non-trivial contains double edges. These definitions do not always yield unique structures. Removing other back edges in the example given in Figure 9.3(b) would yield other non-trivial down trees, up trees, or DAGs.

9.2 Problem Statement

The problem addressed in this paper is stated as follows:

> Given a directed graph, find all non-trivial cyclic subgraphs and draw them in an area-aware cyclic node-link diagram. Additionally, find all directed acyclic graphs and all trees attached to one, or connecting two or more non-trivial cyclic subgraphs, and draw them in a hierarchical node-link diagram using optimized area-aware layout algorithms.

Chapter 10

Previous Work

10.1 Topology-Based Graph Drawing

Karp et al. [66] introduced Grasper-CL, a commercial Graph Management System for layouting graphs using a divide-and-conquer method. Graphs are drawn based on topological embedded features over multiple levels. Archambault et al. [17, 18, 19] introduced a method for drawing undirected graphs by deriving a hierarchy of topological features from the original graph and using graph drawing algorithms adapted to each feature. Further, they introduced additional ideas for drawing undirected graphs and for interacting with the hierarchy obtained from the topological decomposition of these undirected graphs (Figure 10.1). Lately, AlTarawneh et al. [12] introduced a general framework adapting this idea to directed graphs in a work-in-progress report. However, their approach is based on arbitrary splits of strongly connected components that are then classified into different features. Neither of these features is formally defined and no examples of the final layout or visualization was provided. Our method proposes a rigorous definition of the underlying graph theoretical concepts. Further, we find maximal non-trivial cyclic subgraphs of the weakly connected components of a given directed graph. Based on these, the remaining subgraphs are then classified as (up, down) trees and DAGs according to the definitions given in Section 9.1. All these subgraphs are then drawn using optimal algorithms and combined in a topological drawing of the directed graph.

10.2 Graph Drawing of Structures in Bio-Informatics and Biology

In bio-informatics and biology, several approaches were proposed for drawing the cyclic parts of directed graphs. Karp et al. [67] follow steps similar to their previous paper [66] to find circular, branched, linear, and complex topologies using layout algorithms mainly from the Grasper-CL toolbox [66] (Figure 10.2). Becker et al. [31] draw directed graphs using circular, hierarchical, and force-directed algorithms and try to handle edge crossing using rotation (Figure 10.3). Bourqui et al. [32] present MetaViz to draw the complete metabolic network, which is a mixed graph modeled as bipartite graph, without duplicating nodes to show network topology taking into account the application domain conventions (Figure 10.4). Gabouje and Zimányi [44] presented the C^2GL algorithm to draw biochemical graphs (directed graphs) by defining three types of containers (metanodes) (Figure 10.6). Recently, Lambert et al. [72] extended the work of Bourqui at al. [32] and Rohrschneider at al. [83] to draw the complete metabolic network using a pseudo-orthogonal visualization (Figure 10.5). All four approaches [31, 32, 44, 72] draw cycles using a circular layout adding edges as straight lines inside circle, which may cause many crossing, while applying different layouts for the other features. Many other authors [84, 100]

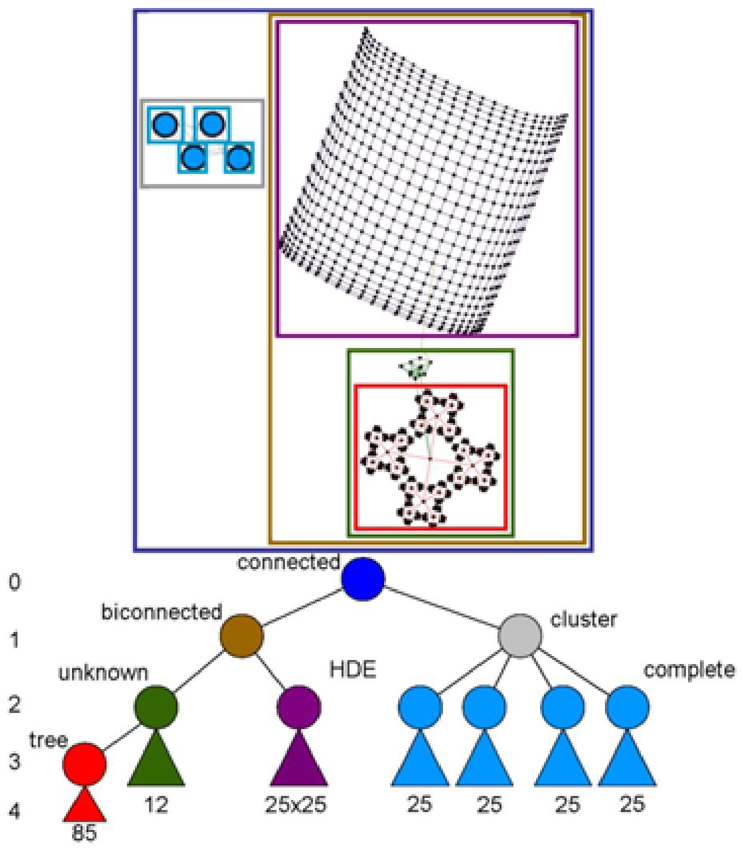


Figure 10.1: Multilevel drawing of undirected graphs using a hierarchy of topological decomposed components, (figure courtesy of [17]).

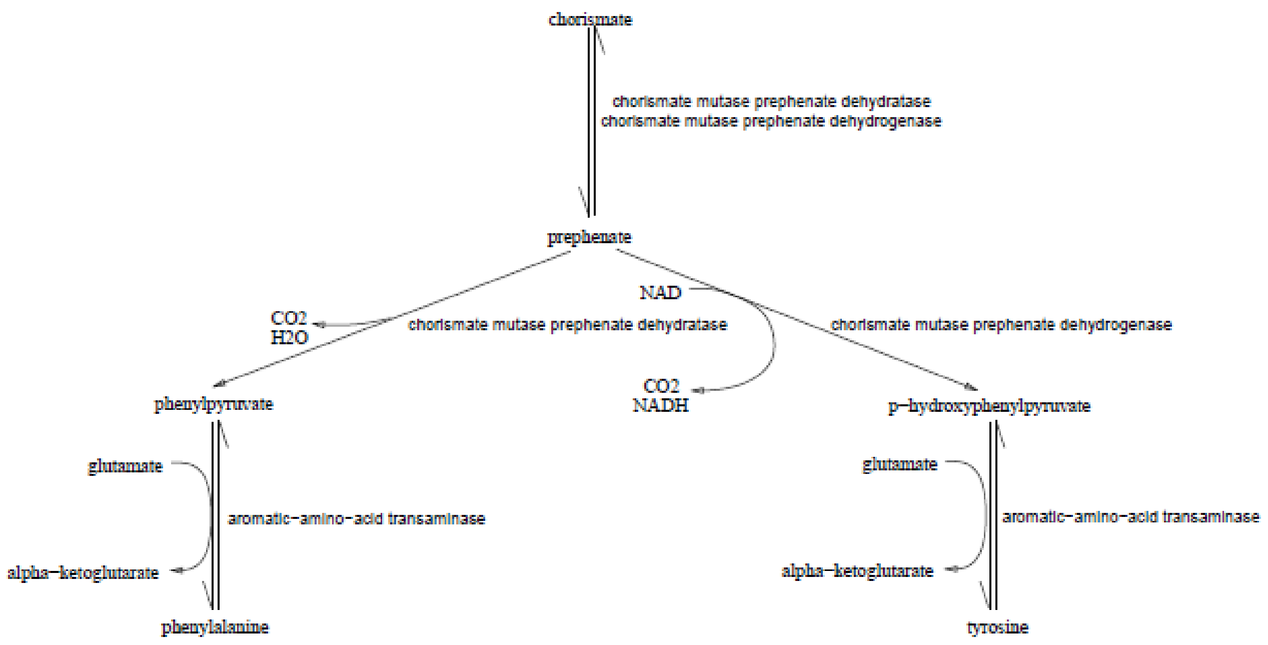

chorismate
chorismate mutase prephenate dehydratase
chorismate mutase prephenate dehydrogenase
prephenate
NAD
CO2
H2O
chorismate mutase prephenate dehydratase
chorismate mutase prephenate dehydrogenase
phenylpyruvate
CO2
NADH
p−hydroxyphenylpyruvate
glutamate
aromatic−amino−acid transaminase
alpha−ketoglutarate
phenylalanine
glutamate
aromatic−amino−acid transaminase
alpha−ketoglutarate
tyrosine

Figure 10.2: Drawing a tree pathway using the Grasper-CL toolbox [66], (figure courtesy of [67]).

worked on the same topic, too. Our approach differs, as it is the first to extract the non-trivial cyclic subgraphs and to draw them in an optimal way using the algorithm proposed by Bachmaier et al. [22] together with DAGs and trees, while allowing to build hierarchies that can be used for the interactive analysis of the directed graphs.

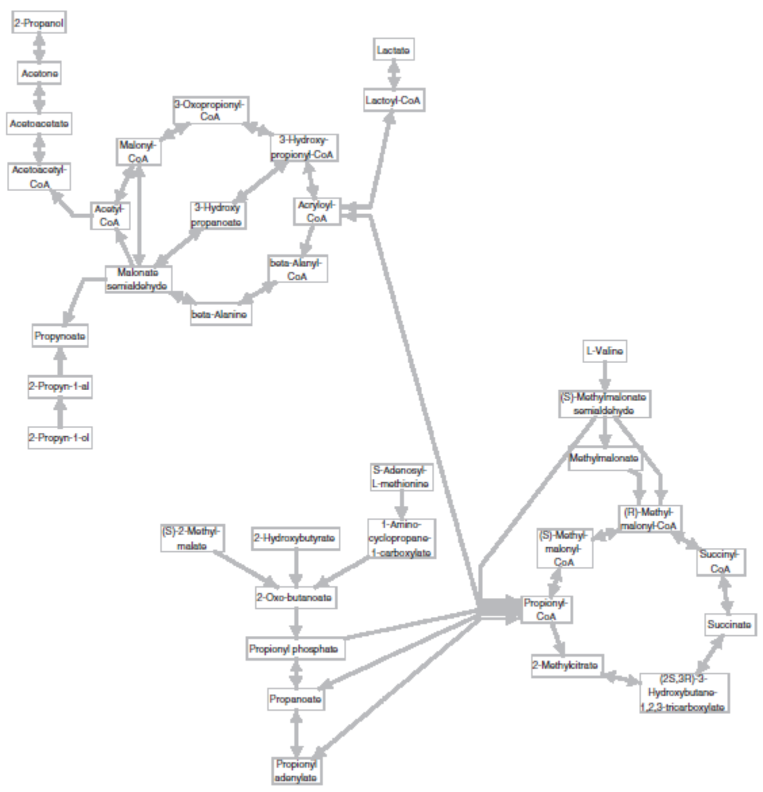


Figure 10.3: The output of using the drawing approach of Becker and Rojas [31] for a metabolic pathway, (figure courtesy of [31]).

10.3 Creation of Hierarchies

Some attempts try to use graph metrics to cluster subgraphs and to build hierarchies based on these clusters. Most of these approaches address undirected graphs, however. Batagelj et al. [25] proposed an interactive hierarchical visualization of undirected graphs using Visual Hybrid (X, Y)-clustering (Figure 10.7). Bourqui et al.[33] proposed a clustering algorithm for dynamic undirected graphs (Figure 10.8). We address directed graphs instead.

Elmqvist et al. [51] proposed an interactive large-scale graph visualization based on a matrix representation (Figure 10.9). In this paper, we focus on the topological visualization of directed graphs using (near-)optimal node-link representations.

10.4 Cyclic Drawings of Graphs

Recently, Bachmaier et al. [22, 20, 21] extended the Brandes and Köpf algorithm [35] for cyclic level drawing by showing cycles in 2D as closed poly-spiral curves surrounding a center (Figure 10.10).

We use it for drawing the non-trivial cyclic subgraphs, only. Further, we separate non-cyclic subgraphs, drawing them using a hierarchical layout.

10.5 Finding Cycles in Software Systems

Laval et al. [75] presented a linear algorithm to find short package cycles, which are the main interest cycles for developers instead of all cycles. In addition, they proposed new undesirability metrics for ranking cycles.

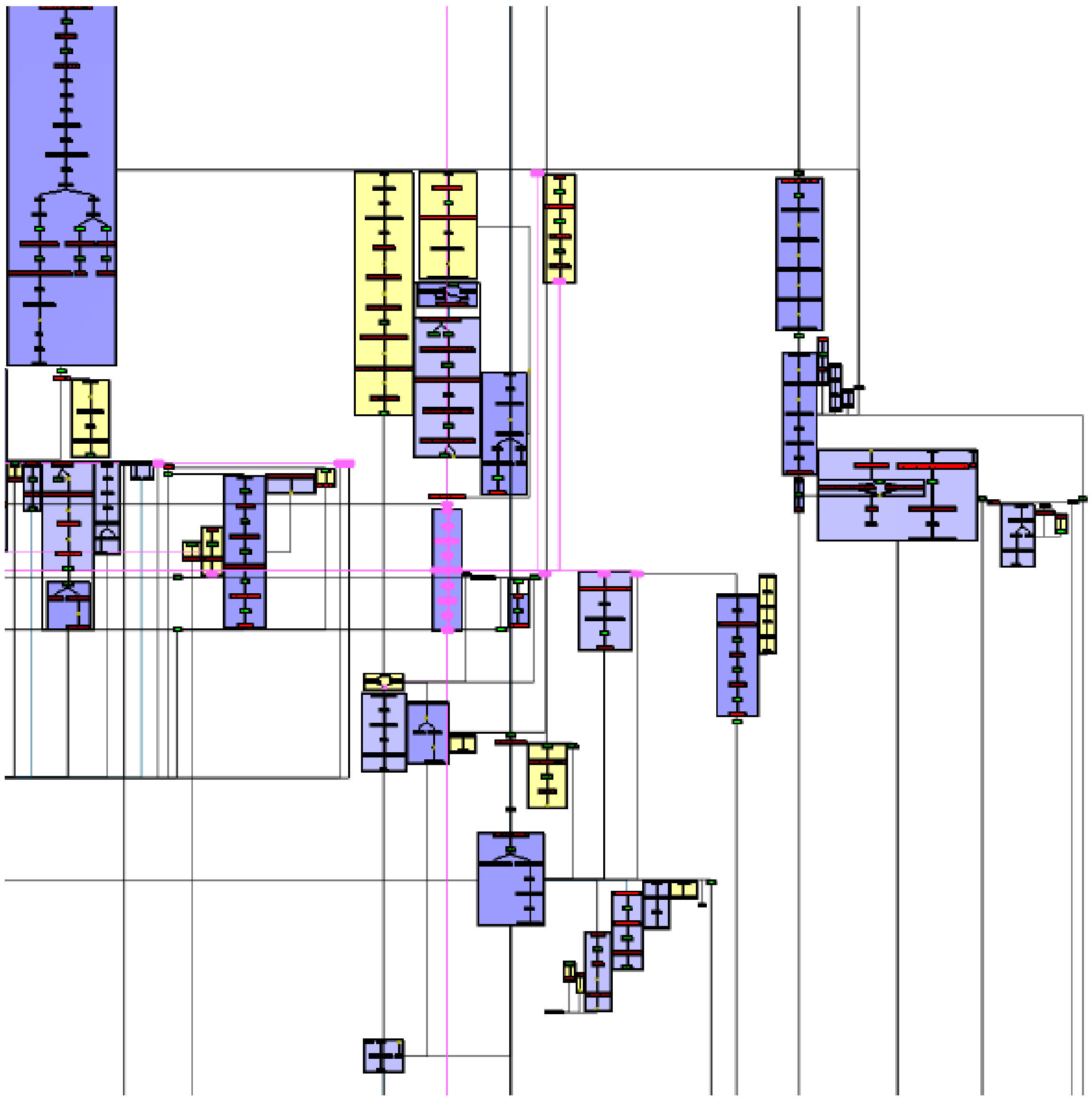


Figure 10.4: Visualizing pathways using a method introduced by Bourqui et al. [32], (figure courtesy of [32]).

Our approach detects non-trivial cyclic subgraphs, instead. Moreover, we propose a complete methodology for the topological decomposition of graphs combined with an optimal drawing of the found components.

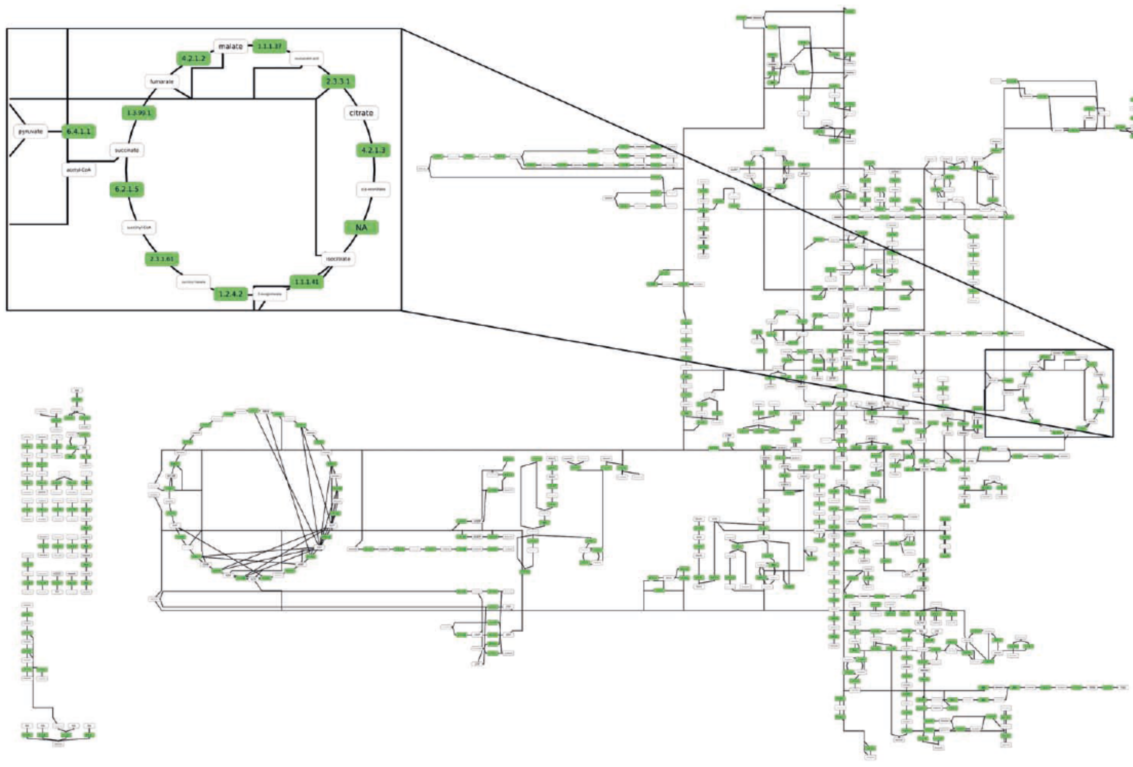


Figure 10.5: Applying the approach of Lambert et al. [72] for drawing metabolic network, (figure courtesy of [72]).

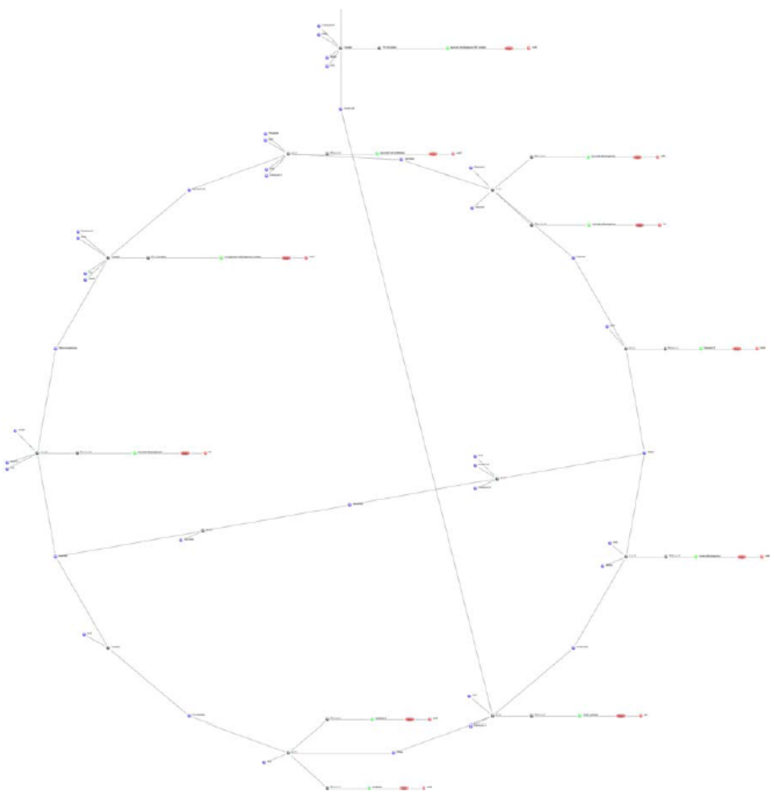


Figure 10.6: Layouting a cycle subgraph using the approach of Skhiri dit Gabouje and Zimianyi [44], (figure courtesy of [44]).

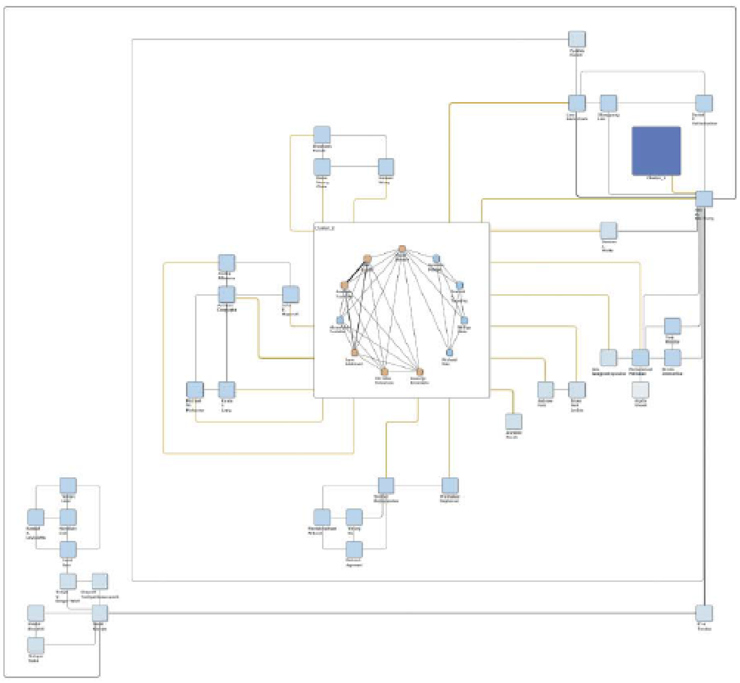


Figure 10.7: Drawing undirected graphs using Visual Hybrid (X, Y)-clustering where one cluster unfolded, (figure courtesy of [25]).

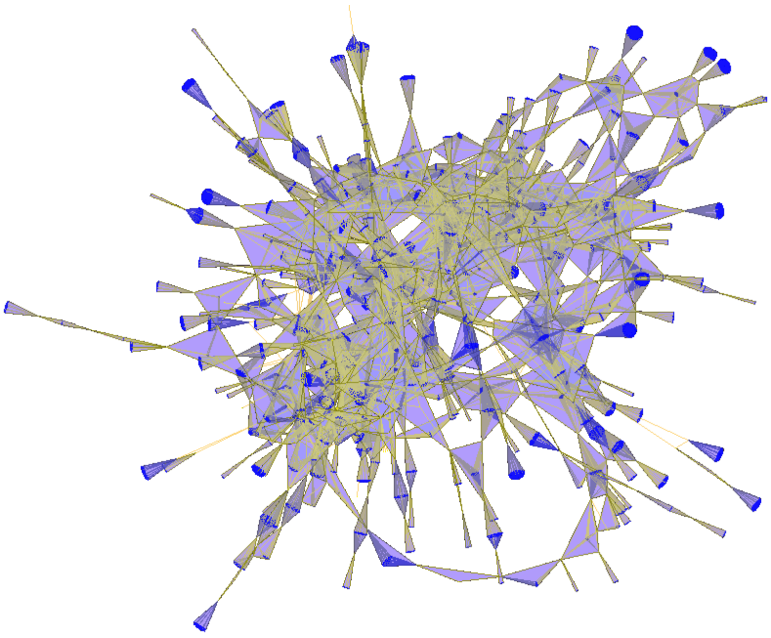


Figure 10.8: An example of decomposed graph using a clustering algorithm for dynamic undirected graphs, (figure courtesy of [33]).

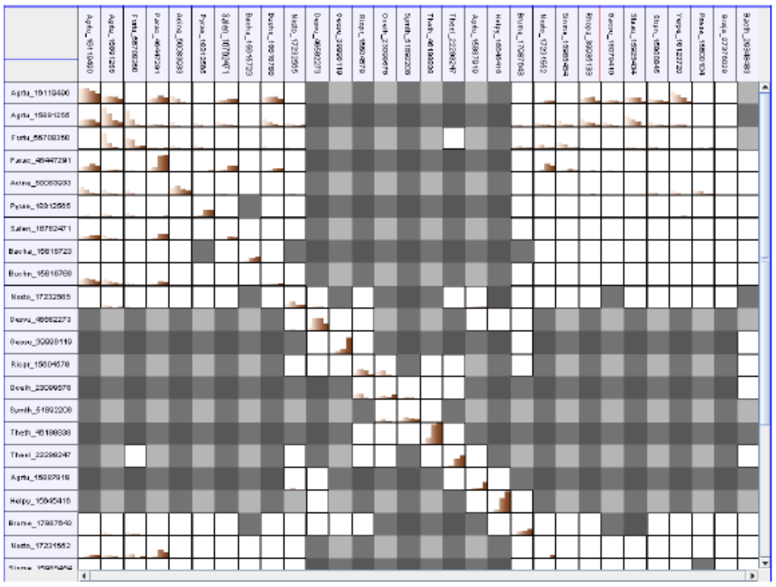


Figure 10.9: An example showing large protein data using an interactive matrix visualization, (figure courtesy of [51]).

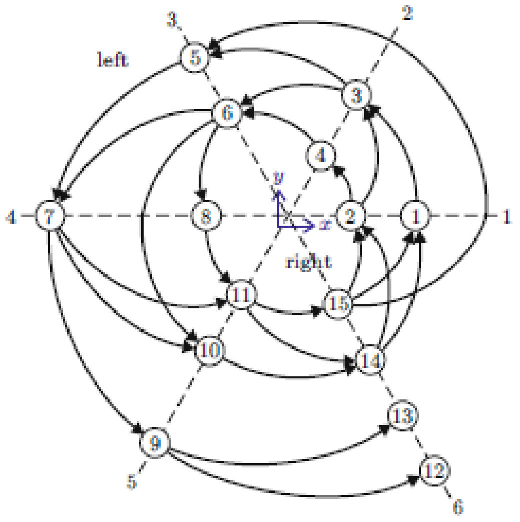


Figure 10.10: Drawing directed graph using cyclic layout, (figure courtesy of [20, 21]).

Chapter 11

Overview of the Process

Creating a topological visualization of directed graphs follows a process that was adapted from the one presented by Archambault et al. [17] for undirected graphs. The general idea in both approaches is to detect topological subgraphs for which optimized layout algorithms exist, followed by drawing the subgraphs using the optimized algorithms and combining the subgraphs to obtain the final layout of the graph.

In our approach, we aim at decomposing the directed graph into sub-graphs representing non-trivial cyclic subgraphs (ntCS), down-trees, up-trees, and DAGs according to the definitions presented in Chapter 9. These will then be drawn using optimized layout algorithms. The overall process and the sub-processes for decomposing and for drawing the directed graph are shown in Figure 12.1. The main process consists of three steps:

1. Taking a directed graph as input, its weakly-connected components (wCCs) are computed.
2. The wCCs are decomposed and a two-level hierarchy is built (Chapter 12)
   (a) Each wCC is decomposed into its topological subgraphs.
   (b) Each subgraph is represented by a meta-node. The meta-nodes are connected by meta-edges that are computed as the union of the edges between two connected topological components. This yields a coarser meta-graph.
   (c) The meta-graph is classified as DAG or tree, as no ntCSs can occur. This is due to the construction of the meta-graph.
3. The graph is drawn (Chapter 13).

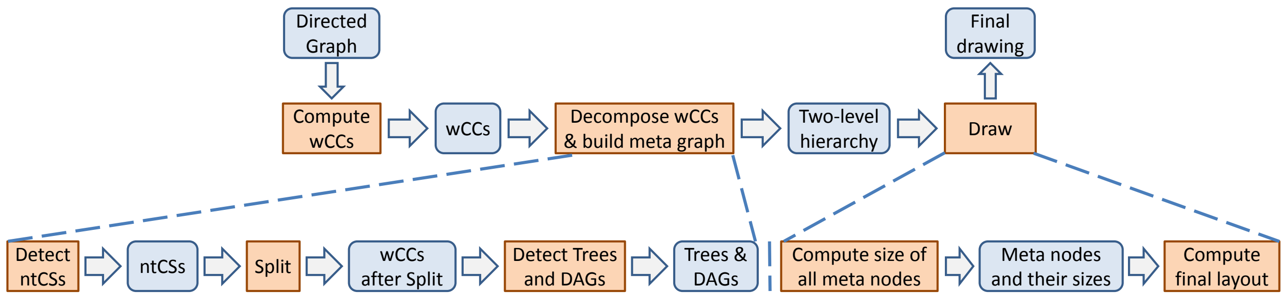


Figure 11.1: Overall process for drawing directed graphs, the sub-process for decomposing wCCs and the sub-process for drawing.

Chapter 12

Decomposition and Hierarchy Construction

12.1 Overview of Decomposition Process

The decomposition of the directed graph constructs topological subgraphs (ntCSs, DAGs, and trees) from each wCC based on our algorithms using the following sub-process (Figure 12.1):

1. A new algorithm to detect ntCSs is performed (Section 12.2).
2. The split (Section 12.3) performs two steps
   (a) All edges belonging to ntCSs are removed.
   (b) Starting at the ntCS nodes, for each edge, wCCs are computed using depth first search. Many wCCs might belong to one ntCS node.
3. Each of the computed wCCs is classified as up-tree, down-tree, or DAG (Section 12.4).
4. The meta-graph is built (Section 12.5).

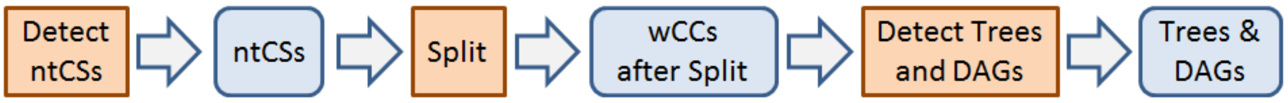


Figure 12.1: Decomposition sub process. Each wCC is decomposed into its topological subgraphs.

12.2 Cycle-Detection

Following the requirements of the application areas, we propose a detection algorithm for non-trivial cyclic subgraphs (ntCSs). In particular, we are not interested in detecting or enumerating all cycles [90, 65, 92, 88] (all algorithms are implemented in JGraphT [24]). Also, our definition is different from strongly connected components.

Algorithm 14.1 uses backtracking and depth first search to find all ntCSs traversing all edges in the graph exactly once. To visualize and to draw ntCSs later in an optimized way, the algorithm considers cycles of length two (double edge) as a special, bi-directional edge and considers shared cycles (cycles with shared nodes) as one ntCS. The sub-figures of Figure 12.2 show different cases that are considered by the algorithm. Except for Figure 12.2(c), they represent one ntCS, each:

a) A single cycle that does not share any nodes or edges with other cycles (Figure 12.2(a)). This case can be handled by checking if a current node was visited using the same current traversed path.

b) Two cycles sharing one node will be combined into one cycle (Figure 12.2(b)). After finding each cycle individually as in the previous case, the two cycles will be merged into one cycle.

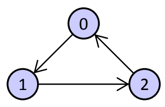
c) Two cycles, which are connected by one double edge (Figure 12.2(c)). This case is handled by removing length two cyclic parts from the path while checking if the path is (part of) a cycle.

d) Two cycles sharing one edge will be combined (Figure 12.2(d)). After finding each cycle individually, the two cycles will be merged into one cycle.

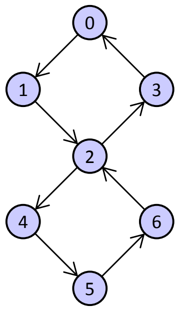
e) Two cycles sharing more than one edge will be combined (partial cycle) (Figure 12.2(e)). First, one cycle is detected. Then, the current path is checked for two nodes of the current path being contained in the same ntCS. If this is the case, then the intermediate nodes of the path between those nodes are added to the ntCS.

f) Single cycle with two double edges (Figure 12.2(f)). Let the nodes visiting sequence be $= 1, 2, 3, 2, 1$. Following this sequence will not result in cycle $C1$ at the end. To handle this situation, back edges (in this case edge $3 \rightarrow 2$) are considered after all other edges of a node.

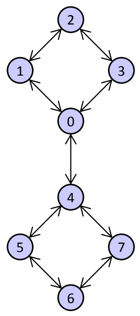
g) All other cases can be considered being a combination of the previous cases. For example, two cycles with double edges sharing one edge will be combined (Figure 12.2(g)) by applying cases e) and f). Additional examples are shown in Figure 12.3.



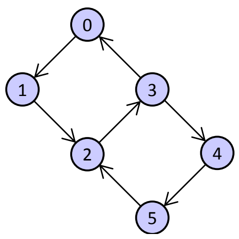
(a) C1=$\{0, 1, 2\}$
Single cycle



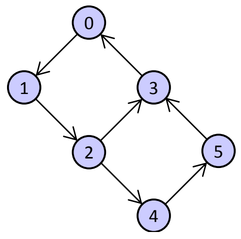
(b) C1=$\{0, 1, 2, 3, 4, 5, 6\}$
Two cycles sharing one node will be combined



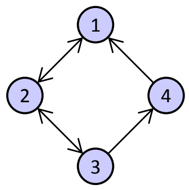
(c) C1=$\{0, 1, 2, 3\}$
C2=$\{4, 5, 6, 7\}$
Two cycles



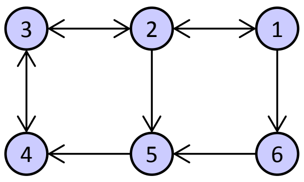
(d) C1=$\{0, 1, 2, 3, 4, 5\}$
Two cycles sharing one edge will be combined



(e) C1=$\{0, 1, 2, 3, 4, 5\}$
Two cycles sharing one edge will be combined (partial cycle)

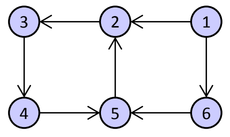


(f) C1=$\{1, 2, 3, 4\}$
Single cycle with two double edges

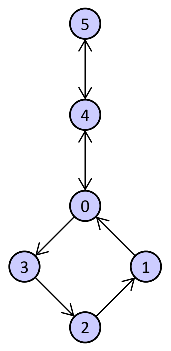


(g) C1=$\{1, 2, 3, 4, 5, 6\}$
Two cycles sharing one edge will be combined

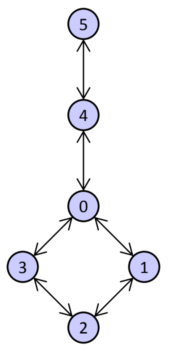
Figure 12.2: Potential Cycle Cases



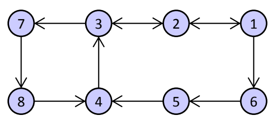
(a) C1=$\{2, 3, 4, 5\}$
Single cycle



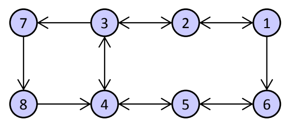
(b) C1=$\{0, 1, 2, 3\}$
Single cycle



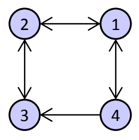
(c) C1=$\{0, 1, 2, 3\}$
Single cycle with four double edges



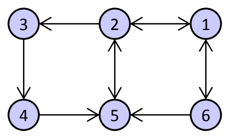
(d) C1=$\{1, 2, 3, 4, 5, 6, 7, 8\}$
Two cycles sharing one single edge will be combined



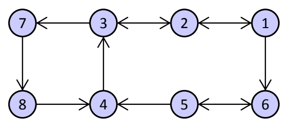
(e) C1=$\{1, 2, 3, 4, 5, 6, 7, 8\}$
Two cycles sharing one double edge will be combined



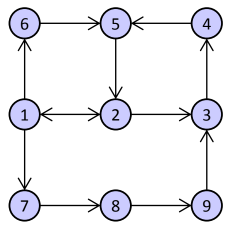
(f) C1=$\{1, 2, 3, 4\}$
Single cycle with three double edges



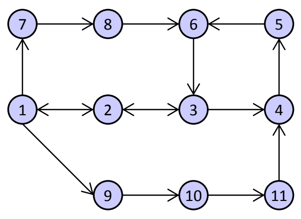
(g) C1=$\{1, 2, 3, 4, 5, 6\}$
Two cycles sharing one double edge will be combined



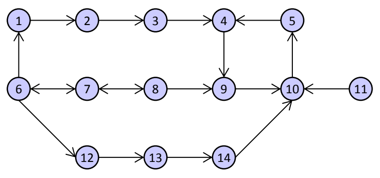
(h) C1=$\{1, 2, 3, 4, 5, 6, 7, 8\}$
Two cycles sharing one single edge will be combined



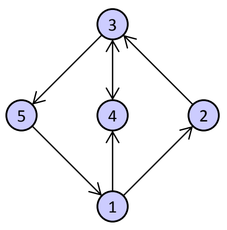
(i) C1=$\{1, 2, 3, 4, 5, 6, 7, 8, 9\}$
Many cycles sharing edges will be combined



(j) C1=$\{1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11\}$
Many cycles sharing edges will be combined

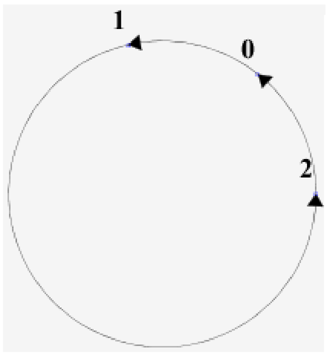


(k) C1=$\{4, 5, 9, 10\}$Single cycle

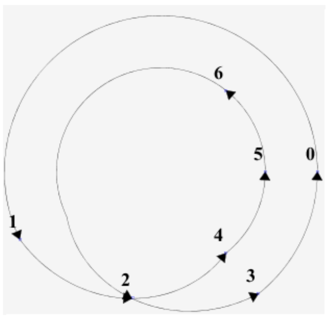


(l) C1=$\{1, 2, 3, 4, 5\}$
Many cycles sharing edges will be combined

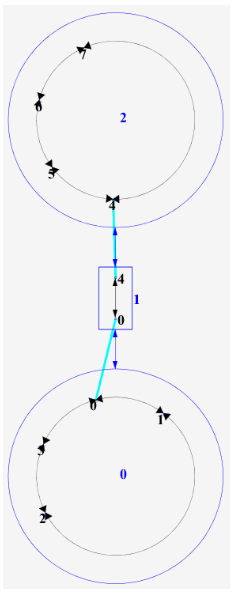
Figure 12.3: Additional examples for combining the previous cycle cases (Figure 12.2)



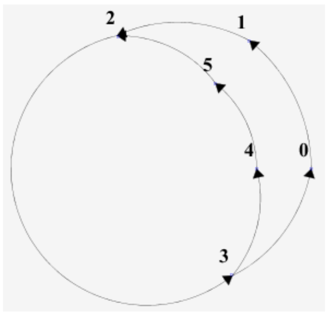
(a) Layout of Figure 12.2(a)



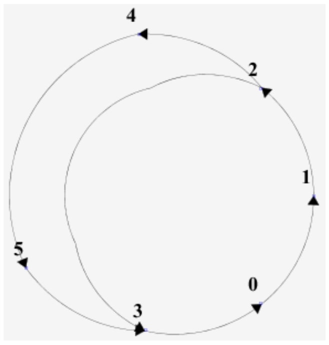
(b) Layout of Figure 12.2(b)



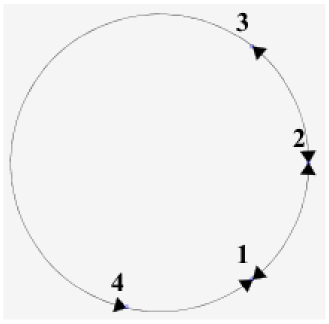
(c) Layout of Figure 12.2(c)



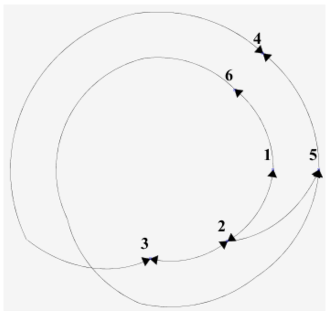
(d) Layout of Figure 12.2(d)



(e) Layout of Figure 12.2(e)

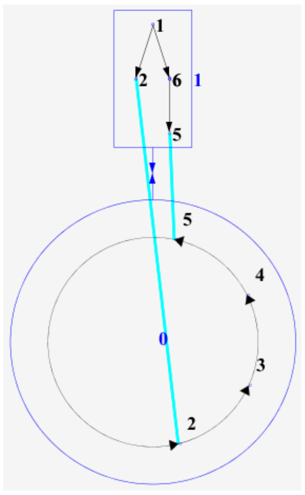


(f) Layout of Figure 12.2(f)

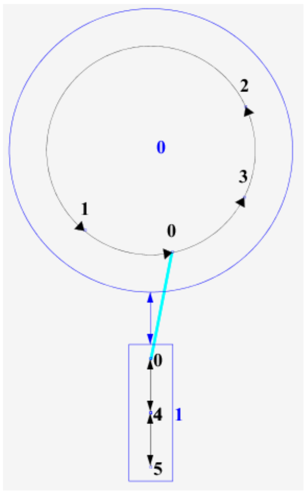


(g) Layout of Figure 12.2(g)

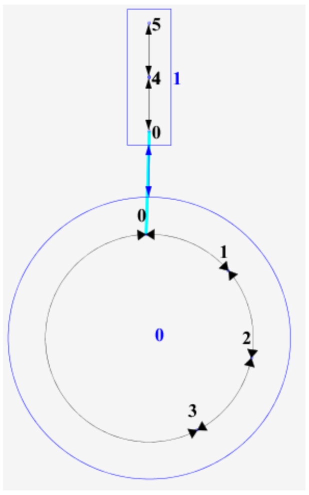
Figure 12.4: Layout of cycle cases in Figure 12.2



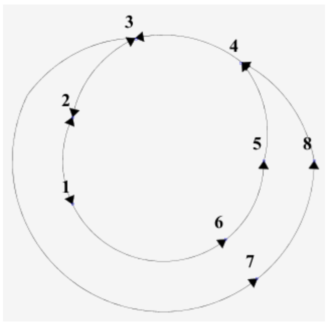
(a) Layout of Figure 12.3(a)



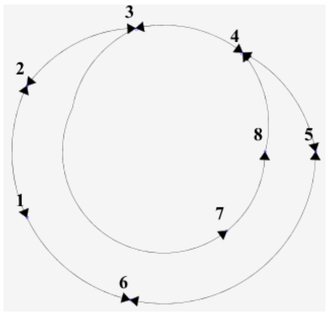
(b) Layout of Figure 12.3(b)



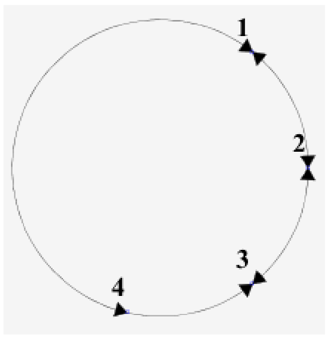
(c) Layout of Figure 12.3(c)



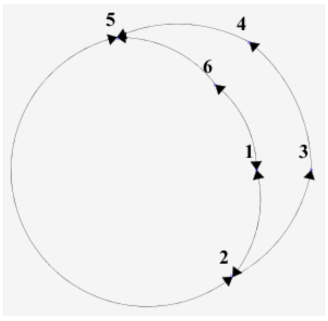
(d) Layout of Figure 12.3(d)



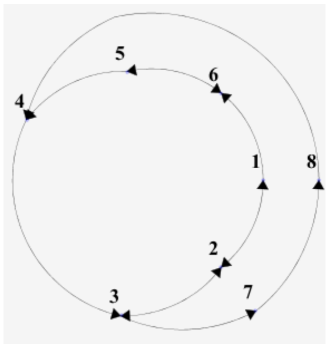
(e) Layout of Figure 12.3(e)



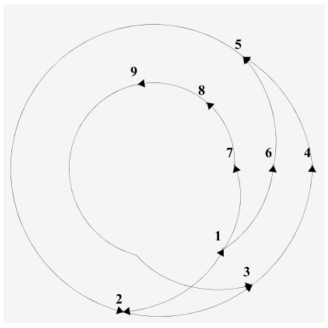
(f) Layout of Figure 12.3(f)



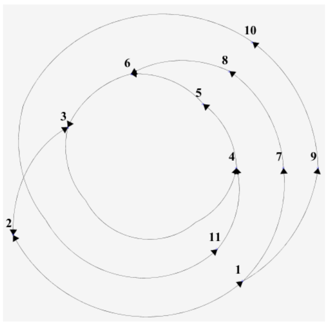
(g) Layout of Figure 12.3(g)



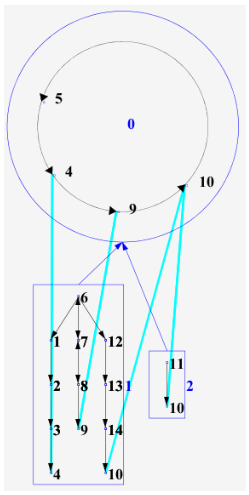
(h) Layout of Figure 12.3(h)



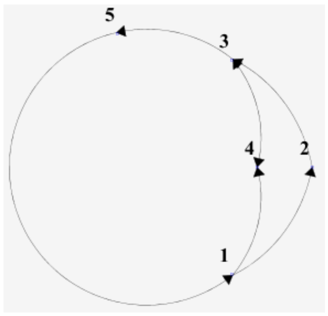
(i) Layout of Figure 12.3(i)



(j) Layout of Figure 12.3(j)



(k) Layout of Figure 12.3(k)



(l) Layout of Figure 12.3(l)

Figure 12.5: Layout of cycle cases in Figure 12.3

12.3 Split

After detecting all ntCSs and before finding trees and DAGs, all edges belonging to the ntCSs found are deleted and the obtained graph is split into wCCs at the nodes of the ntCSs. Figure 12.6 shows an example. The graph was decomposed into four ntCSs after applying the cycle detection. If we remove all edges belonging to the ntCSs found, we are left with one wCC. The left part connects cycle 1 to cycle 2 and 3 via a DAG, and the right part connects cycle 1 to cycle 4 via a tree. Therefore, we split the wCC into two wCCs at node 2. Using only the latter would make the structure less clear, especially in more complex cases. Therefore, we chose to implement a method that splits the wCC in these cases at the nodes of the ntCSs.

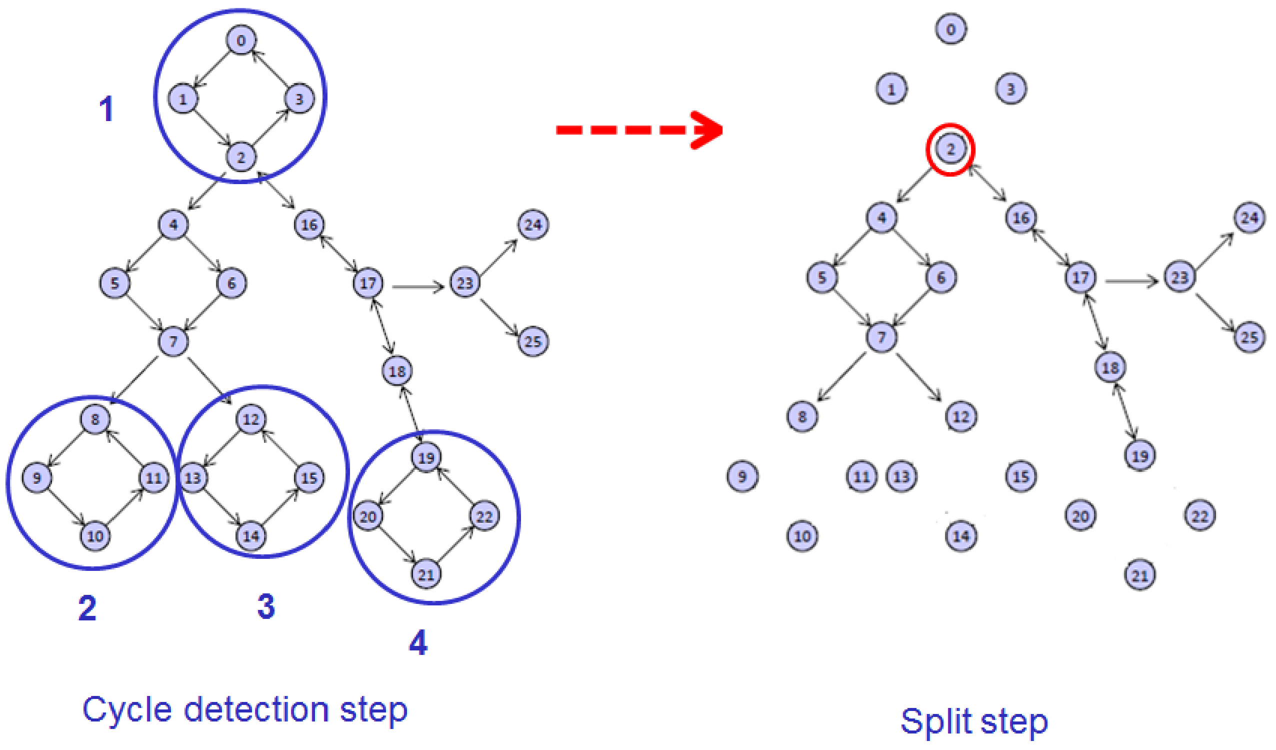


Figure 12.6: Split step example

Starting at a ntCS node n_{c1}, for each edge $e \in \{(n_{c1}, v)|v \in V\} \cup \{(v, n_{c1})|v \in V\}$, the node v of this edge is used as the start of a depth first search for computing the wCCs. The search stops, if the current node has no unvisited edges or if it is a node of a ntCS. All edges of the found wCCs are deleted before handling the next edge. After all edges of the current ntCS node are addressed, the next ntCS node is examined, until all ntCS nodes of this wCC were addressed. This algorithm has complexity $O(n + e)$ as each node and edge is only used once.

12.4 Detect DAGs and Trees

The wCCs found in the previous step are classified into up-trees (UT), down-trees (DT), and DAGs using depth first search. The Algorithm 14.16 detects graph types based on the four main conditions shown in Table 12.1. To visualize and to draw trees and DAGs later in an optimal way, the algorithm considers cycles of length two (double edges) as one edge. Figure 12.7 shows examples for the four potential tree and DAG cases according to the conditions given in Table 12.1. Case i implies immediately, that a DAG that is detected as a down-tree can

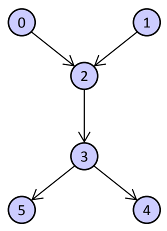
Case	Condition	Result
i	$I \geqslant 2$ and $O \geqslant 2$	DAG
ii	$I < 2$ and $O \geqslant 2$	down-tree or DAG
iii	$I \geqslant 2$ and $O < 2$	up-tree or DAG
iv	$I < 2$ and $O < 2$	down-tree, up-tree, DAG

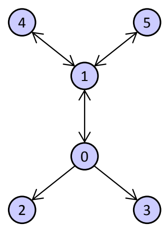
Table 12.1: Conditions used for detecting down-trees, up-trees, and DAGs. Examples are given in Figure 12.7

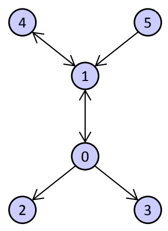
.

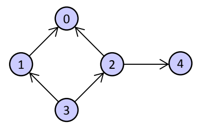
have at most one node with in-degree 0, and an up-tree can have at most one node with out-degree 0 (Figure 12.7(a)). Case ii implies, that the subgraph is a down-tree if a down-tree can be constructed, otherwise, the subgraph is a DAG (Figures 12.7(b), 12.7(c), and 12.7(d)). Case iii implies, that the subgraph is an up-tree if an up-tree can be constructed, otherwise, the subgraph is a DAG (Figures 12.7(e), 12.7(f), and 12.7(g)). In Case iv, first it attempts to construct a down-tree as in Case ii. If no down-tree can be constructed, then it attempts to construct an up-tree as in Case iii. If no up-tree can be constructed, the subgraph is a DAG (Figures 12.7(k), 12.7(l), 12.7(h) 12.7(n), 12.7(j), 12.7(m), and 12.7(i)). The topological drawing of the examples given by Figure 12.7 are shown in Figure 12.8.

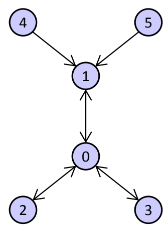
Figure 12.9 shows an example of the final decomposition for one wCC. The wCC is decomposed into four ntCSs (0, 3, 4, 5), one tree (1), and one DAG (2). Each subgraph is represented by a meta-node.

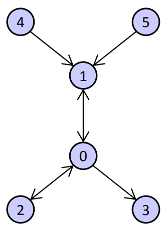

(a) Case i: DAG
$I = 2$, $O = 2$

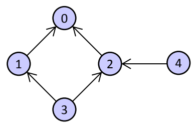

(b) Case ii: DT
$I = 0$, $O = 2$

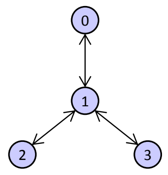

(c) Case ii: DT
$I = 1$
$O = 2$

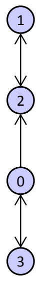

(d) Case ii: DAG
$I = 1$
$O = 2$

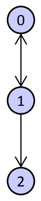

(e) Case iii: UT
$I = 2$, $O = 0$

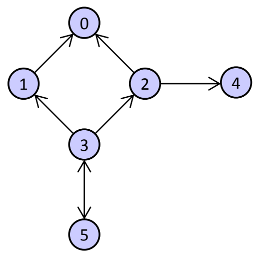

(f) Case iii: UT
$I = 2$
$O = 1$

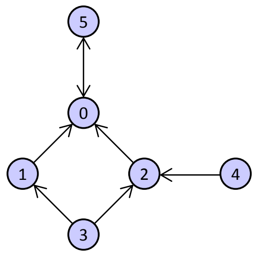

(g) Case iii: DAG
$I = 2$
$O = 1$

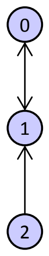

(h) Case iv: Non-trivial DT $I = 0$, $O = 0$


(i) Case iv: DT
$I = 0$
$O = 0$


(j) Case iv: DT
$I = 0$
$O = 1$


(k) Case iv: DAG
$I = 0$
$O = 1$


(l) Case iv: DAG
$I = 1$
$O = 0$


(m) Case iv: DT
$I = 1$
$O = 1$

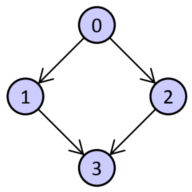
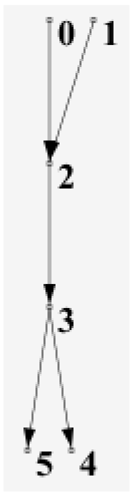
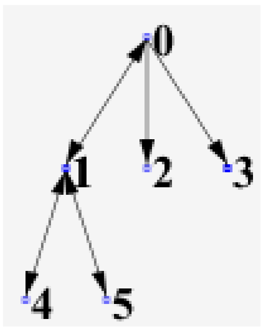

(n) Case iv: DAG
$I = 1$
$O = 1$

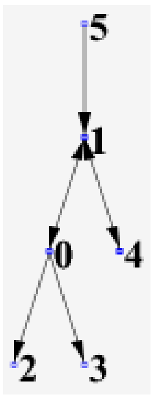
Figure 12.7: Examples for down-trees, up-trees, and DAGs. Classification according to the cases listed in Table 12.1



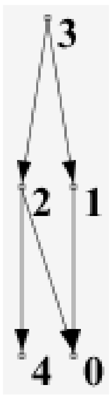
(a) Layout of Figure 12.7(a)



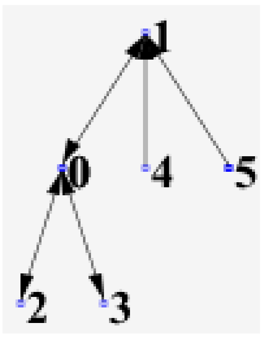
(b) Layout of Figure 12.7(b)



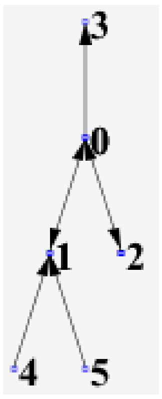
(c) Layout of Figure 12.7(c)



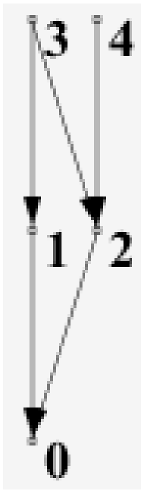
(d) Layout of Figure 12.7(d)



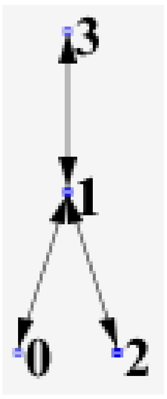
(e) Layout of Figure 12.7(e)



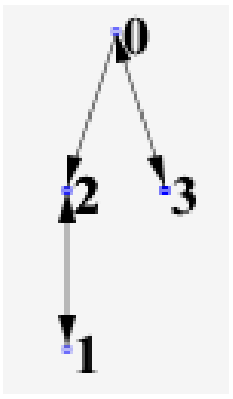
(f) Layout of Figure 12.7(f)



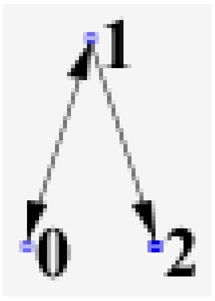
(g) Layout of Figure 12.7(g)



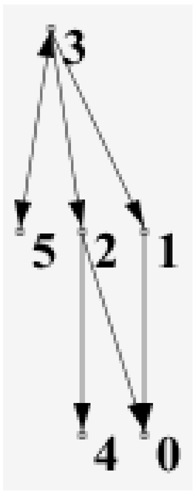
(h) Layout of Figure 12.7(h)



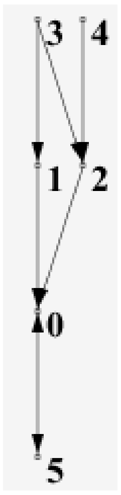
(i) Layout of Figure 12.7(i)



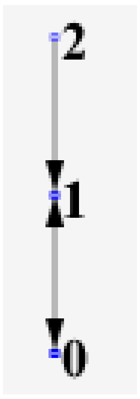
(j) Layout of Figure 12.7(j)



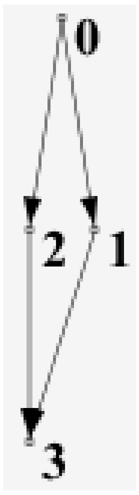
(k) Layout of Figure 12.7(k)



(l) Layout of Figure 12.7(l)



(m) Layout of Figure 12.7(m)



(n) Layout of Figure 12.7(n)

Figure 12.8: Layout of trees and DAGs cases in Figure 12.7

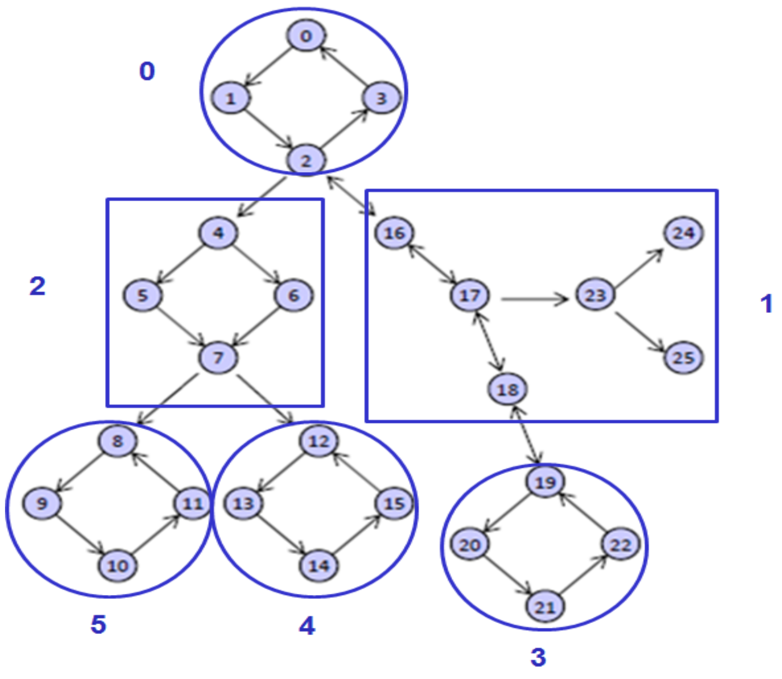


Figure 12.9: Final Decomposition example.

12.5 Hierarchy Construction

After the directed graph was decomposed into its topological subgraphs, a new graph is constructed. Therefore, each subgraph is represented by a meta-node. These meta-nodes are connected by meta-edges. This yields a coarser meta-graph.

Each meta-edge connects a ntCS and a DAG or a tree. There are five types of meta-edges:

- single outgoing edge ($\longrightarrow$)
- single incoming edge ($\longleftarrow$)
- double edge ($\longleftrightarrow$),
- blocking edge ($\rightarrow\!\leftarrow$),
- double path edge ($\leftharpoonup\!\!\!\!\!\rightharpoondown$).

The type of the meta-edge depends on how the ntCS is connected to DAGs and trees. If the ntCS is connected to DAGs and trees with only one edge, then the edge type is determined as follows:

- A *single outgoing edge* is used, if a path can be constructed such that:
  - the first node belongs to the ntCS,
  - the first edge is an outgoing edge,
  - each node of the path except the first and the last has only one incoming and one outgoing edge,
  - and the last node of the path is either a node of another ntCS or a node without outgoing edges or a node with more than two incoming or outgoing edges.

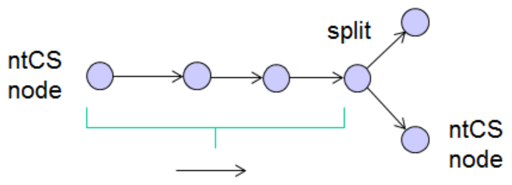
  Please notice, that the path can not end at the same ntCS as this would be a partial cycle.

- A *single incoming edge* is used, if the same conditions hold as for the single outgoing edge, except that the first edge is an incoming edge and the last node of the path is either a node of another ntCS or a node without incoming edges or a node with more than two incoming or outgoing edges.

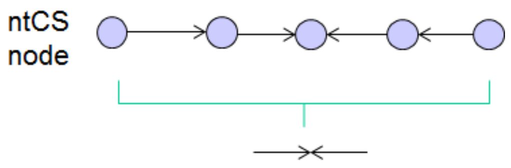
- A *double edge* is used, if the same conditions hold as for the single outgoing edge, except that the path consists only of double edges. In this case, each inner node of the path has exactly two incoming and two outgoing edges.

- A *blocking edge* is used, if a path similiar to the one for single outgoing edges can be constructed where the last node of the path has two incoming edges. In this case, no path from the ntCS to any other ntCS can be constructed; the path is *blocked*. It is also used if the path consists of incoming edges with the last node having two outgoing edges. Again, no path to this ntCS from another ntCS can be constructed.

- A *double path edge* requires two edges of the DAG or tree being connected to a ntCS (see below).



(a) Outgoing meta-edge



(b) Blocking meta-edge

Figure 12.10: Examples for meta-edges

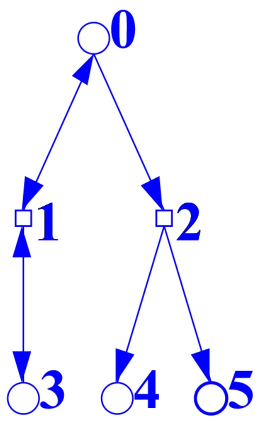
Type 1	Type 2	Type 3	Type 4	Result
$\longleftarrow$				$\longleftarrow$
$\longleftarrow$	$\rightarrow\!\leftarrow$			$\longleftarrow$
$\longrightarrow$				$\longrightarrow$
$\longrightarrow$	$\rightarrow\!\leftarrow$			$\longrightarrow$
$\rightarrow\!\leftarrow$				$\rightarrow\!\leftarrow$
$\longrightarrow$	$\longleftarrow$			$\leftharpoonup\!\!\!\!\rightharpoondown$
$\longrightarrow$	$\longleftarrow$	$\rightarrow\!\leftarrow$		$\leftharpoonup\!\!\!\!\rightharpoondown$
$\longleftrightarrow$				$\longleftrightarrow$
$\longrightarrow$	$\longleftrightarrow$			$\longleftrightarrow$
$\longleftarrow$	$\longleftrightarrow$			$\longleftrightarrow$
$\rightarrow\!\leftarrow$	$\longleftrightarrow$			$\longleftrightarrow$
$\longrightarrow$	$\longleftarrow$	$\longleftrightarrow$		$\longleftrightarrow$
$\longrightarrow$	$\rightarrow\!\leftarrow$	$\longleftrightarrow$		$\longleftrightarrow$
$\longleftarrow$	$\rightarrow\!\leftarrow$	$\longleftrightarrow$		$\longleftrightarrow$
$\longrightarrow$	$\longleftarrow$	$\rightarrow\!\leftarrow$	$\longleftrightarrow$	$\longleftrightarrow$

Table 12.2: Combination of edge types.

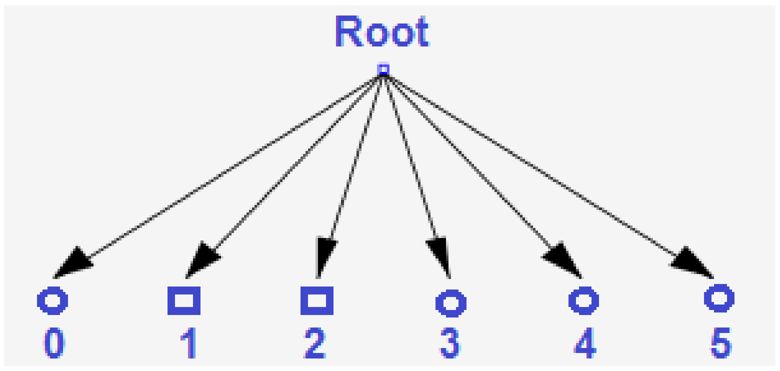
Examples of determining single outgoing edge and blocking edge are shown in Figure 12.10. The path ends at a node with more than one outgoing edge resulting in a single outgoing meta-edge (Figure 12.10(a)). However, if this path ends at a node with more than one incoming edge a blocking meta-edge is obtained (Figure 12.10(b)).

If the ntCS is connected to the DAG or the tree with two or more edges, then the meta-edge type is determined by determining the individual edge types and combining them as shown in Table 12.2. Each of the types combined to form the result appears at least once.

As all ntCSs are already found in the decomposition of the original graph, the meta-graph can only be a DAG or a tree. This DAG or tree is detected and constructed using breadth first search instead of depth first search. The edge types "double path edge" and "blocking edge" are treated similar to double edges. Breadth first search is used to avoid cyclic dependencies that otherwise could occur in meta-graph. Figure 12.11 shows a meta-graph and a two-level hierarchy that was derived from the final decomposition example in Figure 12.9. In Figure 12.11(b), the root node of the hierarchy represents the down-tree detected in the meta-graph and the leaf nodes represent nodes of the meta-graph in Figure 12.11(a), respectively.



(a) Meta-graph derived from example in Figure 12.9.



(b) Two-level hierarchy formed by meta-graph in 12.11(a)

Figure 12.11: Building two-level hierarchy example

Chapter 13

Drawing

13.1 Overview of Drawing

We can draw the complete directed graph, the meta-graph only, or the meta-graph with some meta-nodes replaced by the subgraphs that they are representing.

Drawing the Complete Graph

The sub-process for drawing the complete graph is shown in Figure 13.1. For each meta-node, its accumulated area is computed (Section 13.2). The algorithms used for drawing the meta-graph depending on its type are:

- *Trees:* An area-aware version of the improved Walker's algorithm for layouting trees [37].
- *DAGs:* An area-aware version of the Sugiyama algorithm for layouting DAGs [87].

The subgraph represented by a meta-node is drawn inside the area associated to its meta-node (Section 13.3).

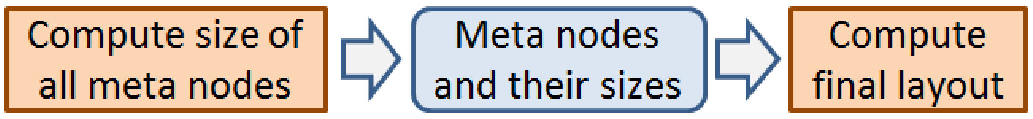


Figure 13.1: The steps used for layouting the original graph.

Drawing the Meta-Graph

Ignoring layouting all lower level subgraphs and using a constant size for each meta-node will lay out only the elements of the meta-graph. The example in Figure 13.2 shows a graph having only one wCC that is broken down into four ntCSs, one tree, and one DAG. The down-tree meta-graph is built from these components considering them as meta-nodes and connecting them by meta-edges. The layout shows the nodes and edges of the meta-graph using blue color, while using black color for all elements of the original graph. Figure 13.3 shows the two-level hierarchy that was derived from the example in Figure 13.2.

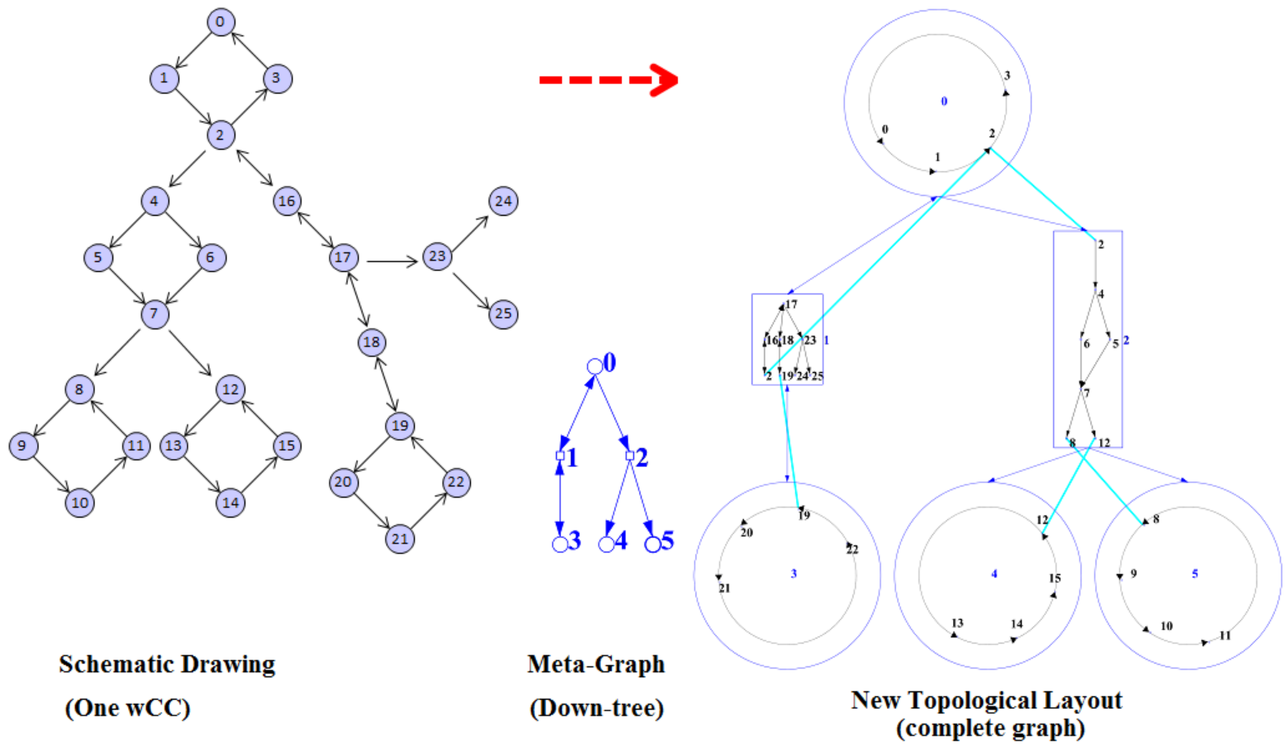


Figure 13.2: Example showing the layout of directed graphs.

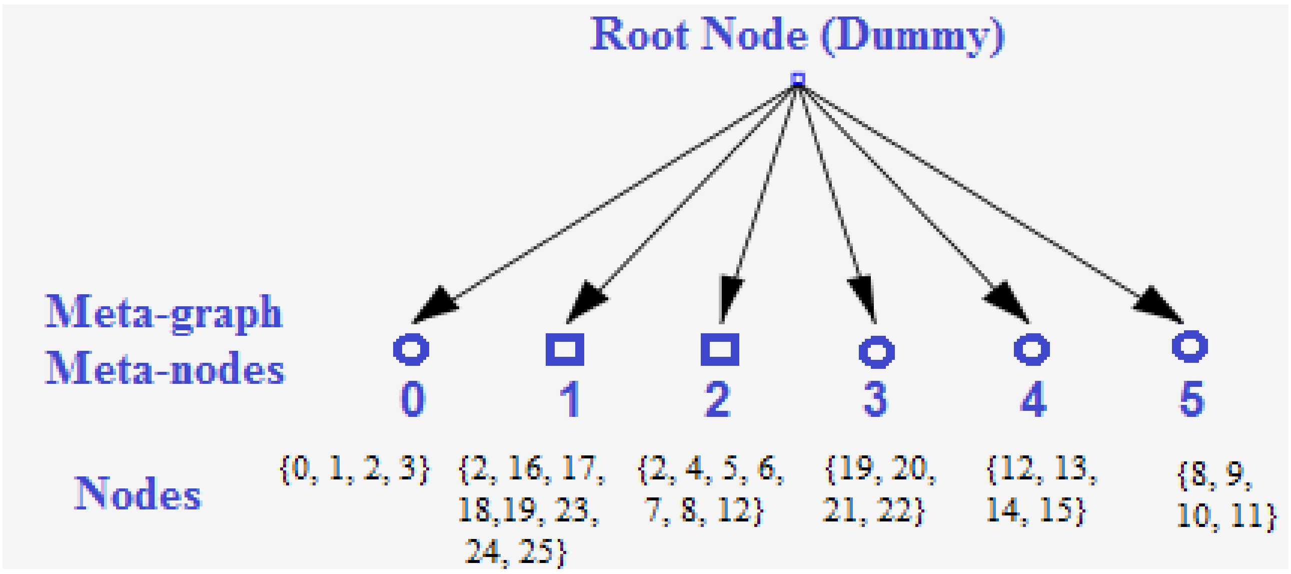


Figure 13.3: The two-level hierarchy of the example in Figure 13.2

13.2 Compute Size of all Meta Nodes

Algorithm 14.39 computes the size of the final area by first calculating the area size for each meta-node in level one, and then the accumulated area of the meta-node in level two is determined. First, the subgraph of the original graph represented by each meta-node in level one is drawn individually depending on its type using the following algorithms:

- *Non-trivial cyclic subgraphs:* Bachmaier's algorithm for cyclic graphs [22].
- *Trees:* The improved Walker's algorithm for drawing trees [37].

- *DAGs:* The Sugiyama algorithm for drawing DAGs [87].

13.3 Final Drawing

The final drawing step involves drawing each subgraph contained in the original graph and represented by a meta-node in level one within the area of its meta-node. This is performed in Algorithm 14.49, where the size for each meta-node and its associated sub-graph layout computed previously using Algorithm 14.39 are used.

The nodes of the original graph are drawn as black points labeled with their names. Each edge of a tree or a DAG is drawn as a black line; each edge of a ntCS is drawn as a black arc. Arrows on one end for single edges and on both ends for double edges show the direction of the respective edge. In addition, nodes shared between two topological components are duplicated and connected using a cyan line (Algorithm 14.58, Section 14.2.2.2). The meta-nodes are represented by drawing bounding blue circles around ntCSs and bounding blue rectangles around trees and DAGs. The meta-edges are also drawn in blue using the shapes (Section 12.5). The final drawing is optimized by rotating the cyclic subgraphs similar to the approach of Archambault et al. [17] (Algorithm 14.57, Section 14.2.2.1). This step reduces the overall distance between duplicated nodes.

Chapter 14

Algorithms

14.1 Decomposition and Hierarchy Construction Algorithms

14.1.1 Algorithms of Cycle Detection

The class diagram in Figure 14.1 shows an overview of the cycles detection implementation part. All global data attributes (variables) and data structures which are related to cycles detection are listed in all these diagrams. The Cycles class on the left of Figure 14.1 is the main class containing all methods for implementing the detection of ntCS s. All other classes in Figure 14.1 are container classes holding the data for the respective entities.

The global data attributes and data structures of the Cycles class are:

- *graph*: contains the input directed graph.
- $finallistOfallCyclesNodes$: a hashtable (Table 14.2) containing all cycles.
- *combinedCycles*: a Cycle object containing a copy of all cycle nodes (as a Hashset, Table 14.4) used during deleting the edges of all cycle nodes after detecting all cycles (pre-processing the graph for the next step).

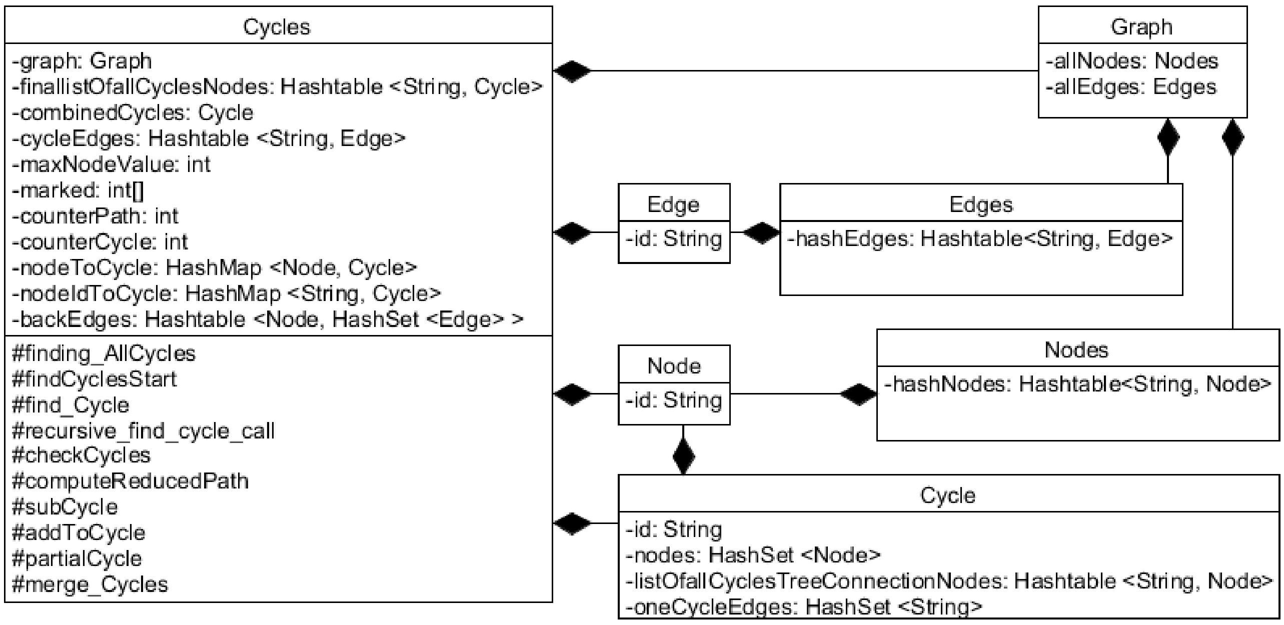

Cycles
-graph: Graph
-finallistOfallCyclesNodes: Hashtable <String, Cycle>
-combinedCycles: Cycle
-cycleEdges: Hashtable <String, Edge>
-maxNodeValue: int
-marked: int[]
-counterPath: int
-counterCycle: int
-nodeToCycle: HashMap <Node, Cycle>
-nodeIdToCycle: HashMap <String, Cycle>
-backEdges: Hashtable <Node, HashSet <Edge> >
#finding_AllCycles
#findCyclesStart
#find_Cycle
#recursive_find_cycle_call
#checkCycles
#computeReducedPath
#subCycle
#addToCycle
#partialCycle
#merge_Cycles
Graph
-allNodes: Nodes
-allEdges: Edges
Edge
-id: String
Edges
-hashEdges: Hashtable<String, Edge>
Node
-id: String
Nodes
-hashNodes: Hashtable<String, Node>
Cycle
-id: String
-nodes: HashSet <Node>
-listOfallCyclesTreeConnectionNodes: Hashtable <String, Node>
-oneCycleEdges: HashSet <String>

Figure 14.1: Class Diagram for Cycles-Detection part.

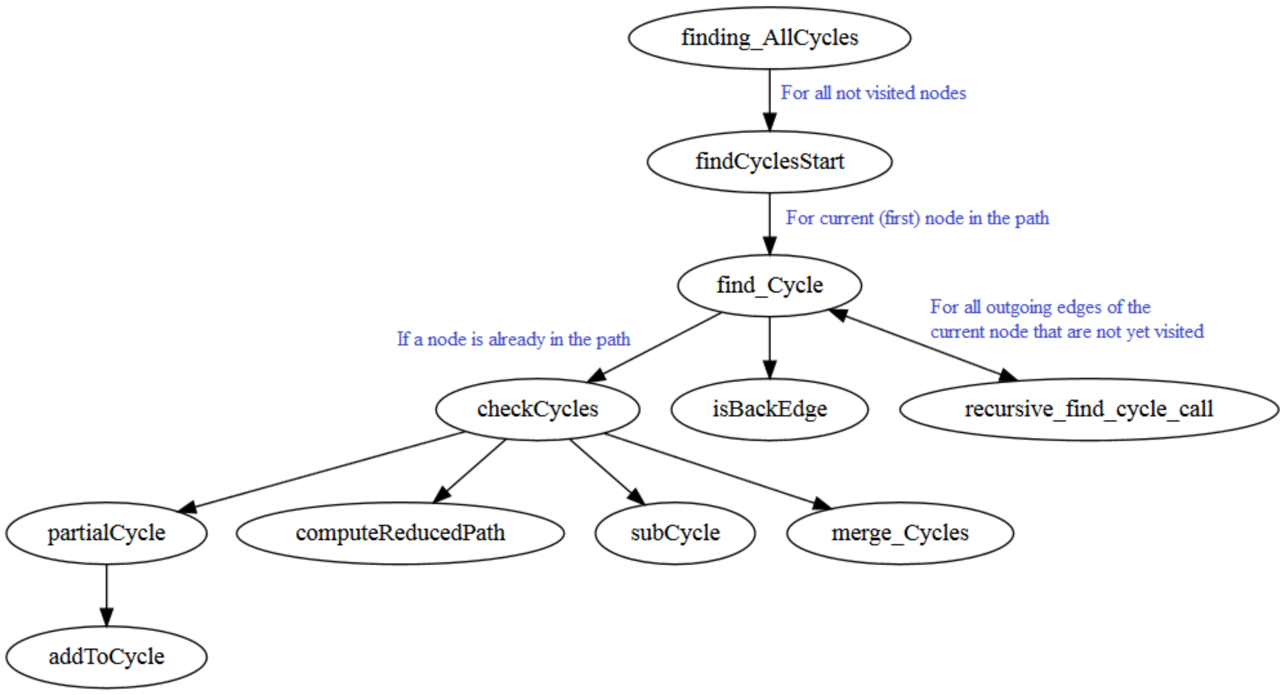

finding_AllCycles
For all not visited nodes
findCyclesStart
For current (first) node in the path
find_Cycle
If a node is already in the path
For all outgoing edges of the current node that are not yet visited
checkCycles
isBackEdge
recursive_find_cycle_call
partialCycle
computeReducedPath
subCycle
merge_Cycles
addToCycle

Figure 14.2: Call graph for all algorithms provided by the Cycles class (Figure 14.1).

- *cycleEdges*: a hashtable (Table 14.2), which contains all edges of all cycle nodes used during deleting the edges of all cycle nodes after detecting all cycles.
- *maxNodeValue*: an integer number, which is the maximum node id value (id is represented by an integer number) in the graph.
- *marked*: an array of boolean values indicating if a node was visited, where the indexes of the array are node ids.
- *counterPath*: a counter, which represents the current path number.
- *counterCycle*: a counter, which represents the number of currently found cycles.
- *nodeToCycle*: a HashMap (Table 14.2), which maps node objects to their cycles.
- *backEdges*: a hashtable (Table 14.2) that stores all back edges of nodes represented by the node id as key and the node back edges hashset (Table 14.4) as value. During visiting not visited outgoing edges of a node, an outgoing edge represents a 'back edge' if it is a reverse edge of an already visited edge.

A call graph of all algorithms provided by the Cycles class (Figure 14.1) is shown in Figure 14.2.

The ntCS detection algorithm is split into ten algorithms handling different cases of potential cycles:

- *FindingAllCycle* (Algorithm 14.1)
  - it marks all edges as not visited (Lines 1-3).
  - it marks all nodes as not visited (Lines 4-6), and
  - to detect all cycles, it calls Algorithm 14.2 for all not visited nodes (Lines 7-11).

- *FindCyclesStart* (Algorithm 14.2)

  The local data attributes and data structures are:

  - *node*: the starting node.
  - *listOfAllCycles*: a list (Table 14.1) of all cycles found.
  - *pathNodes*: a list (Table 14.1) containing path nodes.
  - *pathEdges*: a hashtable (Table 14.2) containing path edge ids that are linked to their position during visiting them.

  Algorithm 14.2 initializes the list containing all cycles, the path nodes list, and the path edges hash table (Lines 1-3). Then, three steps are performed:

  - The current (first) node is added to the path (path nodes list, line 4)
  - *counterPath* is incremented (path length, line 5)
  - The algorithm for finding cycles is called with all previously initialized data structures (line 6).
  - Finally, the list of the path nodes and the hash table of path edges are cleared.

- *Find_Cycle* (Algorithm 14.3)

  The local data attributes and data structures are:

  - *lastNodeInPath*: numeric value of last node in path (current node)
  - *isCycle*: boolean value indicating if a cycle is found.
  - *outgoingEdge*: current outgoing edge of current node.
  - *allNodeBackEdges*: a hashset (Table 14.4) containing all back edges of all nodes.
  - *newNodeBackEdges*: a hashset (Table 14.4) containing all back edges of the current node.
  - *targetNodeOfOutgoingEdge*: the target node of the current outgoing edge.
  - *incomingEdge*: the current incoming edge of the current node.
  - *backEdge*: the current back edge of the current node.

  It performs the following steps:

  - if a node is already in the path (marked with *counterPath*), it checks if a cycle was found by calling Algorithm 14.5 (Lines 1-4).
  - it marks current node with *counterPath* (Line 5).
  - For all outgoing edges of the current node that are not yet visited (Lines 7-18):
    - if the current edge is the reverse edge of an already visited edge (called 'back edge') then store it in the hash set of all back edges (line 10-12)
    - otherwise, it handles the current edge by calling Algorithm 14.4 (Line 14-15).
  - Back edges should not be handled until all incoming edges of the node are visited (Line 19).
  - If the current node has back edges, all of them are handled (Lines 20-28).
    - If the back edge was not visited, it is handled by calling Algorithm 14.4 (Line 25).

- *Recursive_find_cycle_call* (Algorithm 14.4)

  The following steps are performed:

  - Mark outgoing edge as visited (Line 1).
  - Add the target node and the outgoing edge to the corresponding paths (Lines 2-3).
  - Then, call Algorithm 14.3 (Line 4). Algorithm 14.3 and Algorithm 14.4 together implement the depth first search on the graph handling all nodes and edges in a way that allows to find all cycles.
  - Finally, backtracking is performed by removing target node and outgoing edge from their paths (Lines 5-6).

- *Check_Cycles* (Algorithm 14.5)

  The local data attributes and data structures are:

  - *pathSize*: the length (size) of the path.
  - *lastNodeInPath*: the last node in path nodes.
  - *lastNodeOccurneceinPathNodes*: the position (index) of the last node in path nodes.
  - *firstNodeOccurrenceinPathNodes*: the position of the first node occurrence (of the last node) in path nodes (in backward direction).
  - *nodeLastCycle*: the cycle containing the last node in path nodes.
  - *nodeIndex*: the position of the current node in path nodes.
  - *node_1Cycle*: the cycle containing a node at the position before the last node in path nodes.
  - *oneCycleEdges*: a hashset (Table 14.4) containing the edges of the cycle.
  - *nodesCycleList*: a list (Table 14.1) containing nodes to be added to a new cycle.
  - *cycle*: a new cycle instance.

  For a recently visited node, Algorithm 14.5 checks if a part of the path is a cycle:

  - It finds the first node occurrence (of the last node) in *PathNodes* going over each node in *PathNodes* from end node to start node (in backward direction) and checking if the node equals the last node (Lines 1-11).
  - If the difference between the last node position and its first occurrence position in path equals two, an edge of length two (double edge) was found and it returns false (Lines 12-14).
  - If last two nodes in path belong to same cycle, it returns false (Lines 15-17).
  - If first occurrence of node is between 0 and $pathNodeSize - 1$ in Path Nodes (Line 18):
    - It calls *computeReducedPath* Algorithm 14.7 to remove length two edges part, which does not belong to cycle from the path starting from first node occurrence (Line 19).
    - It creates cycle instance from reduced path and adds the new cycle to the list of all cycles Algorithm 14.8 (Lines 21-23). Figure 12.2(a) shows cycle $C1$, which has three nodes $\{0, 1, 2\}$.

    * It merges the found cycle with other cycles if it shares nodes with them (Algorithm 14.11, Figure 12.2(b)) and it returns true (Lines 24-25).
  - If recent node (last node in path) is not in the current path (but it was visited before):
    * It calls *partialCycle* Algorithm 14.10 (Figure 12.2(e)) to find, if a part of the path is a part of a cycle and return true if a partial cycle is found (Line 28).

- *isBackEdge* (Algorithm 14.6)

  The local data attributes and data structures are:

  - *pathEdges*: a hashtable (Table 14.2) containing path edge ids that are linked to their position during visiting them.
  - *edge*: an edge.

  The following steps are performed:

  - Get the reverse id of the edge.
  - Check if *pathEdges* contains the reversed id.

- *ComputeReducedPath* (Algorithm 14.7)

  The local data attributes and data structures are:

  - *pathSize*: the length (size) of the path nodes.
  - *potentialCycleNodesList*: the list (Table 14.1) containing all intermediate nodes.
  - *intermediateNode*: an intermediate node.
  - *finalCycleNodeSet*: the linkedhashset (Table 14.4) containing all final cycle nodes.
  - *finalCycleNodelist*: the list (Table 14.1) containing all final cycle nodes.
  - *indexIntermediateNode*: the position of the intermediate node in *potentialCycle NodesList*.
  - *testEdge*: the reverse edge id of the current intermediate node and the next node in *potentialCycleNodesList*.
  - *endEdgePosition*: the position of *testEdge* in *pathEdges*.
  - *nodeSource*: the intermediate node id in *finalCycleNodelist*.
  - *nodeTarget*: the next node id of the node after the intermediate node in *finalCycle Nodelist*.
  - *cycleEdge*: a cycle edge id, which is formed from *nodeSource* and *nodeTarget*.
  - *finalExtractedNodeslist*: the list (Table 14.1) containing the final cycle nodes set.

  Algorithm 14.7 reduces a part of the path from the potential cycle through searching over edges in edges path:

  - It creates a list containing all intermediate nodes between last node in *pathNodes* and its last occurrence in *pathNodes* (Lines 2-5).
  - It goes over all nodes in the list (Lines 10-20):
    * Linking between the current node and next node in the list as an edge (Line 11)
    * Adding current node to final cycle node Set (Line 12)

    * For the edge of linked two nodes, find a position of its reverse edge in the edges path (Line 13)
    * Changing the position where it should continue going over the remaining nodes in list to this found position (Shifting to this position in the for loop) (Lines 14-19)
  - It returns a list containing the final cycle nodes Set

  In Figure 12.2(c), edges $(0, 4)$ and $(4, 0)$ are an example of this case showing two separated cycles $\{C1, C2\}$.

- *SubCycle* (Algorithm 14.8)

  Algorithm 14.8 searches over all cycles in the list and checks if the cycle is a sub cycle (Lines 1-5). If no, it adds the cycle to the list (Line 6).

- *addToCycle* (Algorithm 14.9)

  The local data attributes and data structures are:

  - *indexIntermediateNode*: the position of the intermediate node in path nodes.
  - *intermediateNode*: the intermediate node.

  Algorithm 14.9 adds to a cycle an intermediate node in a path if the node is not contained in the cycle.

- *PartialCycle* (Algorithm 14.10)

  The local data attributes and data structures are:

  - *cycle*: a cycle containing the last node of *pathNodes*.
  - *nodeIndex*: the a position in path nodes.
  - *currentNode*: the node at position (*nodeIndex*) in path nodes.
  - *cyclePathNode*: the cycle containing *currentNode*.

  Algorithm 14.10 searches over all nodes in path nodes in backward direction finding the node, which is contained in the same cycle as the last node in the path nodes (Lines 3-13) and adds all intermediate nodes to the cycle (Line 8). Then merge cycles (Algorithm 14.11) is called Line (9). In Figure 12.2(e), the algorithm finds cycle $C1$ having four nodes $\{0, 1, 2, 3\}$. Then, it finds the partial part $\{4, 5\}$ merging them with cycle $C1$ at the end.

- *merge_Cycles* (Algorithm 14.11)

  The local data attributes and data structures are:

  - *listOfUncombinedCycles*: the list (Deque, Table 14.3) containing all unmerged cycles.
  - *tempFinallistOfallCycles*: a copy of final cycles hashtable (Table 14.2) (to remove repetition).
  - *cycle*: a cycle instance.
  - *cycleRepeat*: a cycle instance.
  - *tmpListOfUncombinedCycles*: the temporary list (Deque, Table 14.3) of uncombined cycles.

- *combinedCycle*: a cycle instance.
- *combined*: boolean variable.
- *cycle*1: the first cycle instance in *listOfUncombinedCycles*.
- *cycle*2: the cycle instance after *cycle*1 in *listOfUncombinedCycles*.
- *tmpCycle*: a temporary cycle instance composed of *combinedCycle* and *cycle*2.
- *oldNode*: a node in *combinedCycle*.

Algorithm 14.11 goes through all cycles in the list of all cycles and combines cycles with shared nodes. In Figure 12.2(b), the algorithm finds cycle $C1$ having nodes $\{0, 1, 2, 3\}$. Next, it find nodes $\{2, 4, 5, 6\}$ combing them with $C1$ because of shared node 2.

Algorithm 14.1 *FindingAllCycle*
Description: Finding all cycles
Input: $G = (N, E)$
Output: *counterCycle*

```
1:  for all edge ∈Edges do
2:      Mark edge as not visited
3:  end for
4:  for all Nodes do
5:      Mark node as not visited with value -1
6:  end for
7:  for all All nodes do
8:      if node is not visited then
9:          Call findCyclesStart(node) (Algorithm 14.2)
10:     end if
11: end for
```

Algorithm 14.2 *findCyclesStart*
Description: Start find cycles
Input: *node*
Output: *listOfAllCycles*

```
1: Create new list of all cycles
2: Create new list of path nodes
3: Create new hash table of path edges
4: Add node into path nodes list
5: Increment counterPath by 1
6: Call find_Cycle (list of all cycles, path nodes list, path edges table, node) (Algorithm 14.3)
7: Clear the list of the path nodes
8: Clear the hash table of path edges
```

Algorithm 14.3 *find_Cycle*

Description: Find all nodes of cycles in the graph using DFS
Input: *listOfAllCycles*, *pathNodes*, *pathEdges*, *lastNodeInPath*
Output: *listOfAllCycles*

```
1: Increment $lastNodeInPath.iEdgeCounter$ by one
2: if $lastNodeInPath$ is marked with value = $counterPath$ then
3:     $isCycle \leftarrow Check\_Cycles$ ($listOfAllCycles$, $pathNodes$, $pathEdges$) Algorithm 14.5
4: end if
5: Mark $lastNodeInPath$ with $counterPath$
6: Get $allNodeBackEdges$ Hashset of $lastNodeInPath$ if it was created before
7: for all $outgoingEdge \in$ outgoing edges of $lastNodeInPath$ do
8:     if $outgoingEdge$ is marked as not VISITED then
9:         if $isBackEdge(pathEdges, outgoingEdge)$ (Algorithm 14.6) then
10:            If $allNodeBackEdges$ was not created before, create new one
11:            Add $outgoingEdge$ into $allNodeBackEdges$
12:            Put current node as key and $allNodeBackEdges$ as value in $backEdges$ hashtable
13:        else
14:            $targetNodeOfOutgoingEdge \leftarrow$ target node of $outgoingEdge$
15:            Call $recursive\_find\_cycle\_call$ ($outgoingEdge$, $pathNodes$, $targetNodeOfOutgoingEdge$, $pathEdges$, $listOfAllCycles$, $lastNodeInPath$) (Algorithm 14.4)
16:        end if
17:    end if
18: end for
19: if $lastNodeInPath.iEdgeCounter > lastNodeInPath$ number of incoming edges then
20:     if $allNodebackEdges \neq$ null then
21:         for all $backEdge \in allNodebackEdges$ do
22:             if $backEdge$ edge is marked as not VISITED then
23:                 $targetNodeOfOutgoingEdge \leftarrow$ target node of $backEdge$
24:                 $outgoingEdge \leftarrow backEdge$
25:                 Call $recursive\_find\_cycle\_call$ ($outgoingEdge$, $pathNodes$, $targetNodeOfOutgoingEdge$, $pathEdges$, $listOfAllCycles$, $lastNodeInPath$) Algorithm 14.4
26:             end if
27:         end for
28:     end if
29: end if
```

Algorithm 14.4 *recursive_find_cycle_call*

Description: Recursively call find cycle
Input: *outgoingEdge*, *pathNodes*, *targetNodeOfOutgoingEdge*, *pathEdges*, *listOfAllCycles*, *nodeNumber*
Output: *listOfAllCycles*

```
1: Mark outgoingEdge as VISITED
2: Add targetNodeOfOutgoingEdge to pathNodes
3: Put the outgoingEdge key and the position of the target node in pathEdges
4: Call find_Cycle (listOfAllCycles, pathNodes, pathEdges, targetNodeOfOutgoingEdge)
   recursively (Algorithm 14.3)
5: Remove last occurrence of targetNodeOfOutgoingEdge from pathNodes
6: Remove the outgoingEdge from pathEdges
```

Algorithm 14.5 *Check_Cycles*

Description: Check if a part of the path is a cycle
Input: *listOfAllCycles*, *pathNodes*, *pathEdges*
Output: true iff cycle was found

```
1: $pathSize \leftarrow$ size of $pathNodes$
2: $lastNodeInPath \leftarrow$ last node in $pathNodes$
3: $lastNodeOccurrenceinPathNodes \leftarrow pathSize - 1$
4: $firstNodeOccurrenceinPathNodes \leftarrow$ -1
5: $nodeLastCycle \leftarrow$ the cycle containing $lastNodeInPath$
       {Store in $firstNodeOccurrenceinPathNodes$ the first position of node occurrence for
   ($lastNodeInPath$) in $pathNodes$ from end}
6: for $nodeIndex \leftarrow pathSize - 2$ down to 0 do
7:     if node having index $nodeIndex$ equals $lastNodeInPath$ then
8:         $firstNodeOccurrenceinPathNodes \leftarrow nodeIndex$
9:         Break
10:    end if
11: end for
12: if $lastNodeOccurrenceinPathNodes - firstNodeOccurrenceinPathNodes = 2$ then
        {path = double edge}
13:    return false
14: end if
15: $node\_1Cycle \leftarrow$ cycle containing node having index $pathSize - 2$
16: if $node\_1Cycle \neq null \wedge nodeLastCycle \neq null \wedge node\_1Cycle = nodeLastCycle$ then
        {Two sequence nodes in same cycle}
17:    return false
18: else if $firstNodeOccurneceinPathNodes > -1 \wedge firstNodeOccurrenceinPathNodes <$
    $pathSize\ -1$ then
19:    $nodesCycleList \leftarrow$ call $Compute\_ReducedPath$ ($pathNodes$, $pathEdges$,
    $firstNodeOccurrenceinPathNodes$, $oneCycleEdges$) (Algorithm 14.7)
20:    if $nodesCycleList \neq null$ then
           {Make a copy of found cycle}
21:        Create new cycle $cycle$
22:        Add into $cycle$ all nodes and edges of $nodesCycleList$
           {To check if the cycle is not in the list of all cycles yet}
23:        Call $subCycle$ ($listOfAllCycles$, $cycle$) (Algorithm 14.8)
24:        Call $merge\_Cycles$ ($listOfAllCycles$) (Algorithm 14.11)
25:        Return true
26:    end if
27: else if $nodeLastCycle \neq null$ then
28:    return $partialCycle$ ($pathNodes$, $listOfAllCycles$, $lastNodeInPath$) (Algorithm 14.10)
29: end if
```

Algorithm 14.6 *isBackEdge*

Description: Check if *edge* **is a back edge**
Input: *pathEdges* , *edge*
Output: true iff *edge* **is a back edge**

```
1: Create $testEdge$ as reverse edge of start and end nodes
2: Get the position of $testEdge$ from $pathEdges$ and store it in $endEdgePosition$
3: if $endEdgePosition \neq null$ then
4:    return $True$
5: else
6:    return $False$
7: end if
```

Algorithm 14.7 *compute_ReducedPath*

Description: Reduce a part of the path from the potential cycle through searching over edges in edges path
Input: *pathNodes*, *pathEdges*, *startNode*, *oneCycleEdges*
Output: *finalExtractedNodeslist*

```
1: $pathSize \leftarrow$ size of $pathNodes$
2: Create $potentialCycleNodesList$ as new list for potential cycle nodes
3: for $indexIntermediateNode \leftarrow startNode$ to $pathSize$ do
4:    Add to $potentialCycleNodesList$ a node having index $indexIntermediateNode$ in $pathNodes$
5: end for
6: Create $finalCycleNodeSet$
7: if Second node in $potentialCycleNodesList$ equals a node having index $pathSize - 2$ in $pathNodes \wedge$ first node in $potentialCycleNodesList$ equals last node in $pathNodes$ then
       {Start and end edge are reverse to each other.}
       {No cycle, reduced path is empty}
8:    Return null
9: else
10:   for $indexIntermediateNode \leftarrow 0$ to size of $potentialCycleNodesList - 1$ do
11:      $testEdge \leftarrow$ linking ids of two nodes in $potentialCycleNodesList$ having an index $indexIntermediateNode + 1$ and an index $indexIntermediateNode$ respectively
12:      Add a node having an index $indexIntermediateNode$ in $potentialCycleNodesList$ to $finalCycleNodeSet$
13:      $endEdgePosition \leftarrow$ get the position of $testEdge$ in $pathEdges$
14:      if $endEdgePosition \neq null$ then
15:         $endEdgePosition \leftarrow endEdgePosition - startNode$
16:      end if
17:      if $endEdgePosition \neq null \wedge endEdgePosition > indexIntermediateNode$ then
18:         $indexIntermediateNode \leftarrow endEdgePosition$
19:      end if
20:   end for
21: end if
22: Create $finalExtractedNodeslist$ as new list for $finalCycleNodeSet$
23: return $finalExtractedNodeslist$
```

Algorithm 14.8 *subCycle*

Description: Add the cycle to *listOfAllCycles* **if it was not added**
Input: *listOfAllCycles*, *cycle*
Output: *listOfAllCycles*

```
1: for all $tmpCycle \in listOfAllCycles$ do
2:    if $tmpCycle$ contains all nodes of input $cycle$ then
3:       return
4:    end if
5: end for
6: Add $cycle$ into list of all cycles
7: Increment $counterCycle$ by 1
```

Algorithm 14.9 *addToCycle*

Description: Add to a cycle an intermediate node in a path if the node is not contained in the cycle
Input: *indexCurrentNode*, *pathNodes*, *foundCycle*, *lastNodeInPath*
Output: Add intermediate nodes to *foundCycle*

```
1: for $indexIntermediateNode \leftarrow indexCurrentNode + 1$ to size of $pathNodes$ do
2:    $intermediateNode \leftarrow$ Retrieve the node having $indexIntermediateNode$ index from $pathNodes$
3:    if $foundCycle$ does not contain $intermediateNode$ then
4:       Add $intermediateNode$ to $foundCycle$
5:       Put $intermediateNode$ and $foundCycle$ as key and value in $nodeToCycle$ hashmap
6:    end if
7: end for
```

Algorithm 14.10 *partialCycle*

Description: Search over all nodes in path nodes in backward direction finding the node, which is contained in the same cycle as the last node in the path nodes and adds all intermediate nodes to the cycle
Input: *pathNodes*, *listOfAllCycles*, *lastNodeInPath*
Output: true iff a Path is part of a cycle

```
1: $cycle \leftarrow$ Retrieve the cycle having $lastNodeInPath$ node from $nodeToCycle$
2: if $cycle \neq null$ then
3:    for $nodeIndex \leftarrow pathSize - 2$ down to 0 do
4:       $currentNode \leftarrow$ node having $nodeIndex$ in $pathNodes$
5:       if $currentNode \neq lastNodeInPath$ then
6:          $cyclePathNode \leftarrow$ Retrieve the cycle having current node from $nodeToCycle$
7:          if $cyclePathNode \neq null \wedge cycle = cyclePathNode$ then
8:             Call $addToCycle$ ($indexCurrentNode$, $pathNodes$, $cyclePathNode$, $lastNodeInPath$) (Algorithm 14.9)
9:             Call $merge\_Cycles$ ($listOfAllCycles$) (Algorithm 14.11)
10:            Return true
11:         end if
12:      end if
13:   end for
14: end if
15: Return false
```

Algorithm 14.11 *merge_Cycles*

Description: Combine cycles with shared nodes
Input: *listOfAllCycles*
Output: merging shared cycles in *listOfAllCycles*

```
1: for all $repeatedCycle \in finallistOfallCyclesNodes$ Hashtable do
2:    for all $cycle \in listOfAllCycles$ do
3:       if $repeatedCycle$ contains $cycle$ nodes then
         {before merging, remove repeated cycles in final hashtable}
4:          remove $repeatedCycle$ from $finallistOfallCyclesNodes$ Hashtable
5:       end if
6:    end for
7: end for
8: Create $listOfUncombinedCycles$ initializing it by list of all cycles
9: Initialize $combined$ flag to false
10: while Size of $listOfUncombinedCycles > 0$ do
11:    Remove first cycle from $listOfUncombinedCycles$
12:    Store removed cycle in $cycle1$
13:    Create $combinedCycle$ cycle initialized by $cycle1$
14:    Change $combined$ flag to true
15:    while $combined$ do
16:       Create $tmpListOfUncombinedCycles$
17:       Change $combined$ flag to false
18:       while $listOfUncombinedCycles$ is not empty do
19:          Remove first cycle from $listOfUncombinedCycles$
20:          Store removed cycle in $cycle2$
21:          Create temporary cycle $tmpCycle$ initializing it by $combinedCycle$
22:          Add all nodes of $cycle2$ into $tmpCycle$
23:          if Nodes size of $tmpCycle$ < nodes size of $combinedCycle$+ nodes size of $cycle2$
   then
24:             Change $combined$ flag to true
25:             Add all nodes and edges of $cycle2$ into $combinedCycle$
26:          else
27:             Add $cycle2$ to $tmpListOfUncombinedCycles$
28:          end if
29:       end while
30:       Store $tmpListOfUncombinedCycles$ in $listOfUncombinedCycles$
31:    end while
32:    Put $combinedCycle$ in $finallistOfallCyclesNodes$ Hashtable
33:    for all $node \in combinedCycle$ nodes do
34:       Put key $node$ having $combinedCycle$ as value in $nodeToCycle$
35:    end for
36: end while
37: Clear $listOfAllCycles$
38: Add all cycles in $finallistOfallCyclesNodes$ to $listOfAllCycles$
39: for all cycles in $finallistOfallCyclesNodes$ do
40:    Add all cycle nodes in one hashtable
41: end for
```

14.1.2 Algorithms of Split

The class diagram in Figure 14.3 shows an overview of the *SplitConnectedComponentAtCyclePoint* class. All global data attributes (variables) and data structures, which are related to split are listed in this diagram. The *SplitConnectedComponentAtCyclePoint* class on the left side of the Figure 14.3 lists the names of all algorithms (operations) used to split wCCs at nodes of ntCSs while for the other classes no algorithms are shown because they have only utilities operations like gets and sets.

The global data attributes and data structures of the class *SplitConnectedComponentAtCyclePoint* are:

- *parentGraph*: contains the input directed graph.
- *cyclesNodes*: is a HashMap (Table 14.2) containing cycle nodes mapped to their cycles.
- *marked*: is a hashtable (Table 14.2) containing the ids of all marked nodes.
- *allWCC*: is an ArrayList (Table 14.1) of all subgraphs of all wCCs after split.

A call graph of all algorithms provided by *SplitConnectedComponentAtCyclePoint* is shown in Figure 14.4.

The split algorithm consists of four parts:

- *SplitConnectedComponentAtCyclePoint* (Algorithm 14.12)

  It marks all nodes as not visited (Lines 1-3) and it calls *check_Start* (Algorithm 14.13) for all *cyclesNodes* that have edges (Lines 4-8).

- *check_Start* (Algorithm 14.13)

  The local data attributes and data structures are:

  - *listOfCyclesNodes*: a HashSet (Table 14.4) of all cycles found.
  - *listOfAllNodes*: a HashSet (Table 14.4) of current path nodes.
  - *allEdges*: a HashSet (Table 14.4) of edges of a cycle node.
  - *wcc*: a graph.

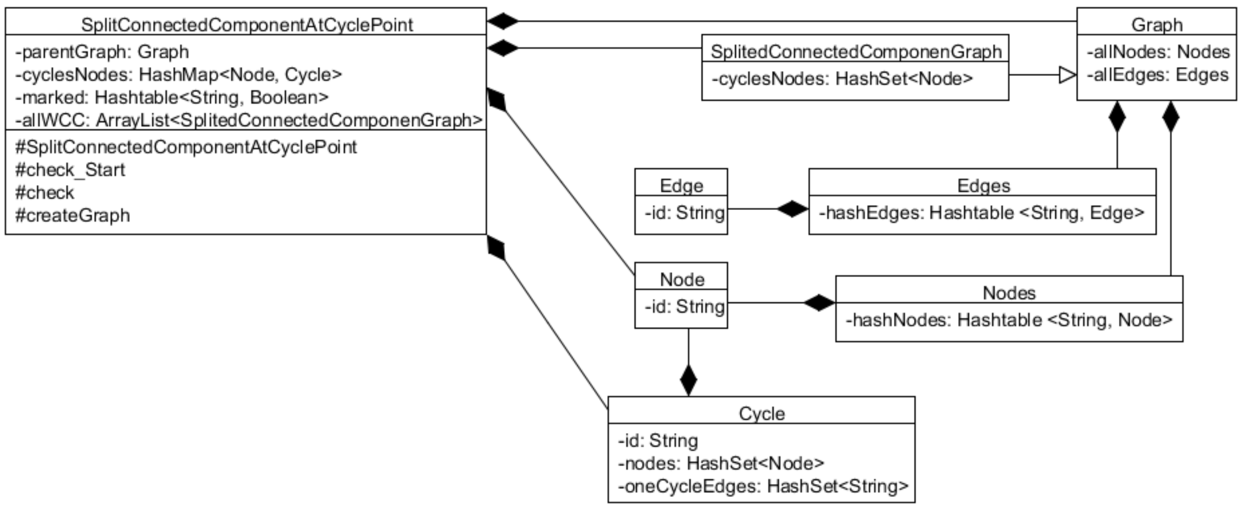

SplitConnectedComponentAtCyclePoint
-parentGraph: Graph
-cyclesNodes: HashMap<Node, Cycle>
-marked: Hashtable<String, Boolean>
-allWCC: ArrayList<SplitedConnectedComponenGraph>
#SplitConnectedComponentAtCyclePoint
#check_Start
#check
#createGraph
SplitedConnectedComponenGraph
-cyclesNodes: HashSet<Node>
Graph
-allNodes: Nodes
-allEdges: Edges
Edge
-id: String
Edges
-hashEdges: Hashtable <String, Edge>
Node
-id: String
Nodes
-hashNodes: Hashtable <String, Node>
Cycle
-id: String
-nodes: HashSet<Node>
-oneCycleEdges: HashSet<String>

Figure 14.3: Class Diagram for Split

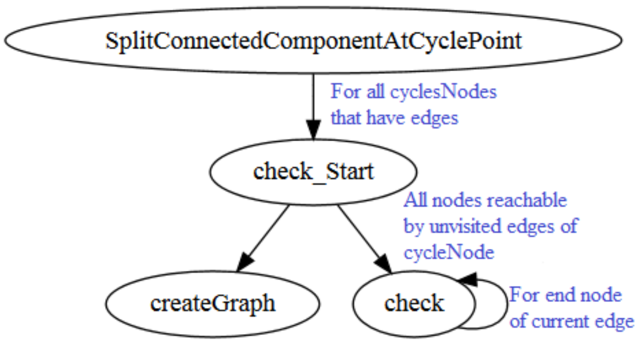

SplitConnectedComponentAtCyclePoint
For all cyclesNodes that have edges
check_Start
All nodes reachable by unvisited edges of cycleNode
createGraph
check
For end node of current edge

Figure 14.4: Call graph for all algorithms provided by the *SplitConnectedComponentAtCyclePoint* class (Figure 14.3).

It initializes *listOfCyclesNodes* and *allEdges* (Lines 1-3). After that, it goes over all nodes reachable by unvisited edges of *cycleNode* (Lines 4-25) calling *check* (Algorithm 14.14) (Line 22). Then, it calls *create_Graph* (Algorithm 14.15) to create a subgragh from *listOfAll-Nodes* (Line 26).

- *check* (Algorithm 14.14)

  The local data attributes and data structures are:

  - *allEdges*: a HashSet (Table 14.4) of edges of a non cycle node.

  If a non cycle node is not visited, it goes over all edges of current non cycle node (Lines 4-23). If the edge is not visited (Line 5):

  - Mark edge as visited (Line 6).
  - Then, if end node of current edge is a cycle node, add it to *listOfCyclesNodes* (Lines 17-18).
  - Otherwise, call *check* (Algorithm 14.14) for end node of current edge recursively (Line 20).

- *create_Graph* (Algorithm 14.15)

  The local data attributes and data structures are:

  - *wcc*: a graph.
  - *listOfAllNodesKey*: a HashSet (Table 14.4) containing ids of all nodes in *listOfAll Nodes*.

  It creates a subgraph consisting of all nodes from *listOfAllNodes* set and edges contained in graph without cycles edges (Lines 1-20).

Algorithm 14.12 *SplitConnectedComponentAtCyclePoint*

Description: Split wCCs at nodes of ntCSs
Input: *parentGraph*, *cyclesNodes*
Output: Split wCCs

```
1: for all $node \in parentGraph$ nodes do
2:    Put in $marked$ hashtable id of $node$ having value $False$
3: end for
4: for all $cycleNode \in cyclesNodes$ do
5:    if # Neighbors of $cycleNode > 0$ then
6:       Call $check\_Start(cycleNode)$ Algorithm 14.13
7:    end if
8: end for
```

Algorithm 14.13 *check_Start*

Description: Start check
Input: *cycleNode*
Output: *allWCC*

```
1: Create $listOfCyclesNodes$ set
2: Add $cycleNode$ to $listOfCyclesNodes$
3: Add all edges of $cycleNode$ to $allEdges$ set
4: for all $edge \in allEdges$ do
5:    if $edge$ is marked as not visited then
6:       Mark $edge$ as visited
7:       if end node of $edge \neq cycleNode$ then
8:          $node2 \leftarrow$end node of $edge$
9:       else
10:         $node2 \leftarrow$start node of $edge$
11:      end if
12:      $reverseEdge \leftarrow$ get reverse edge from graph edges
13:      if $reverseEdge \neq null$ then
14:         Mark $reverseEdge$ as visited
15:      end if
16:      Create $listOfAllNodes$ set
17:      Add $cycleNode$ to $listOfAllNodes$
18:      Add $node2$ to $listOfAllNodes$
19:      if $node2 \in cyclesNodes$ then
20:         Add $node2$ to $listOfCyclesNodes$
21:      else
22:         Call $check(node2, listOfCyclesNodes, listOfAllNodes)$ Algorithm 14.14
23:      end if
24:   end if
25: end for
26: $wcc \leftarrow$ a splited subgraph created out of $listOfAllNodes$ by calling
    $create\_Graph(listOfCyclesNodes, listOfAllNodes)$ Algorithm 14.15
27: if # nodes of $wcc > 0$ then
28:    Add $wcc$ to $allWCC$ list of all subgraphs
29: end if
```

Algorithm 14.14 *check*

Description: Go over all nodes reachable by unvisited edges of cycle node
Input: $nonCycleNode$, $listOfCyclesNodes$, $listOfAllNodes$
Output: $listOfAllNodes$, $listOfCyclesNodes$

```
if $nonCycleNode$ is not marked then
    Mark $nonCycleNode$
    Add all edges of $nonCycleNode$ to $allEdges$ set
    for all $edge \in allEdges$ do
        if $edge$ is marked as not visited then
            Mark $edge$ as visited
            if end node of $edge \neq nonCycleNode$ then
                $node2 \leftarrow$end node of $edge$
            else
                $node2 \leftarrow$start node of $edge$
            end if
            $reverseEdge \leftarrow$ get reverse edge from graph edges
            if $reverseEdge \neq null$ then
                Mark $reverseEdge$ as visited
            end if
            Add $node2$ to $listOfAllNodes$ list
            if $node2 \in cyclesNodes$ then
                Add $node2$ to $listOfCyclesNodes$ list
            else
                Call $check(node2, listOfCyclesNodes, listOfAllNodes)$ Algorithm 14.14
            end if
        end if
    end for
end if
```

Algorithm 14.15 *create_Graph*

Description: Create graph

Input: $listOfCyclesNodes$, $listOfAllNodes$

Output: WCC

```
1: Create $wcc$ an empty subgraph
2: for all $node \in listOfAllNodes$ do
3:     $n1 \leftarrow$ a node having same $id$ of $node$ in original graph $parentGraph$
4:     if $n1 \neq null$ then
5:         Put $n1$ in $wcc$
6:     end if
7: end for
8: for all $node \in listOfAllNodes$ do
9:     $n1 \leftarrow$ a node having same $id$ of $node$ in original graph $parentGraph$
10:    if $n1 \neq null$ then
11:        for all $edge \in$ incoming edges of $n1$ do
12:            $source \leftarrow$ a node having same $id$ of $edge$ start node in $wcc$
13:            $target \leftarrow$ a node having same $id$ of $edge$ end node in $wcc$
14:            if $source \neq null \wedge target \neq null$ then
15:                Create an edge $me$ having $source$ and $target$ as start and end nodes respectively
16:                Put $me$ in $wcc$
17:            end if
18:        end for
19:    end if
20: end for
```

14.1.3 Algorithms of Detect DAGs and Trees

The class diagram in Figure 14.5 shows an overview of the classes for decomposing wCCs into ntCSs, trees, and DAGs in addition to constructing the hierarchy (Section 12.5). The class "Cycle" was discussed in Section 12.2, while the hierarchy construction will be discussed in Section 12.5. All global data attributes (variables) and data structures that are related to trees and DAGs detection are listed in these diagrams. Decomposition, Trees, and DAGs classes show the names of the algorithms (operations) used to detect trees and DAGs.

The global data attributes and data structures of the class "Tree" are:

- id: the id of a tree.
- $root$: the root node of the tree.
- $finallistOfallTreesNodes$: a hashtable (Table 14.2) containing all nodes of the tree.
- $flag$: a numeric variable to mark the tree as down-tree (1) or up-tree (2).

The global data attributes and data structures of the class "Trees" are:

- $finallistOfallTreesNodes$: a hashtable (Table 14.2) containing trees.
- $counterPath$: a counter for paths.

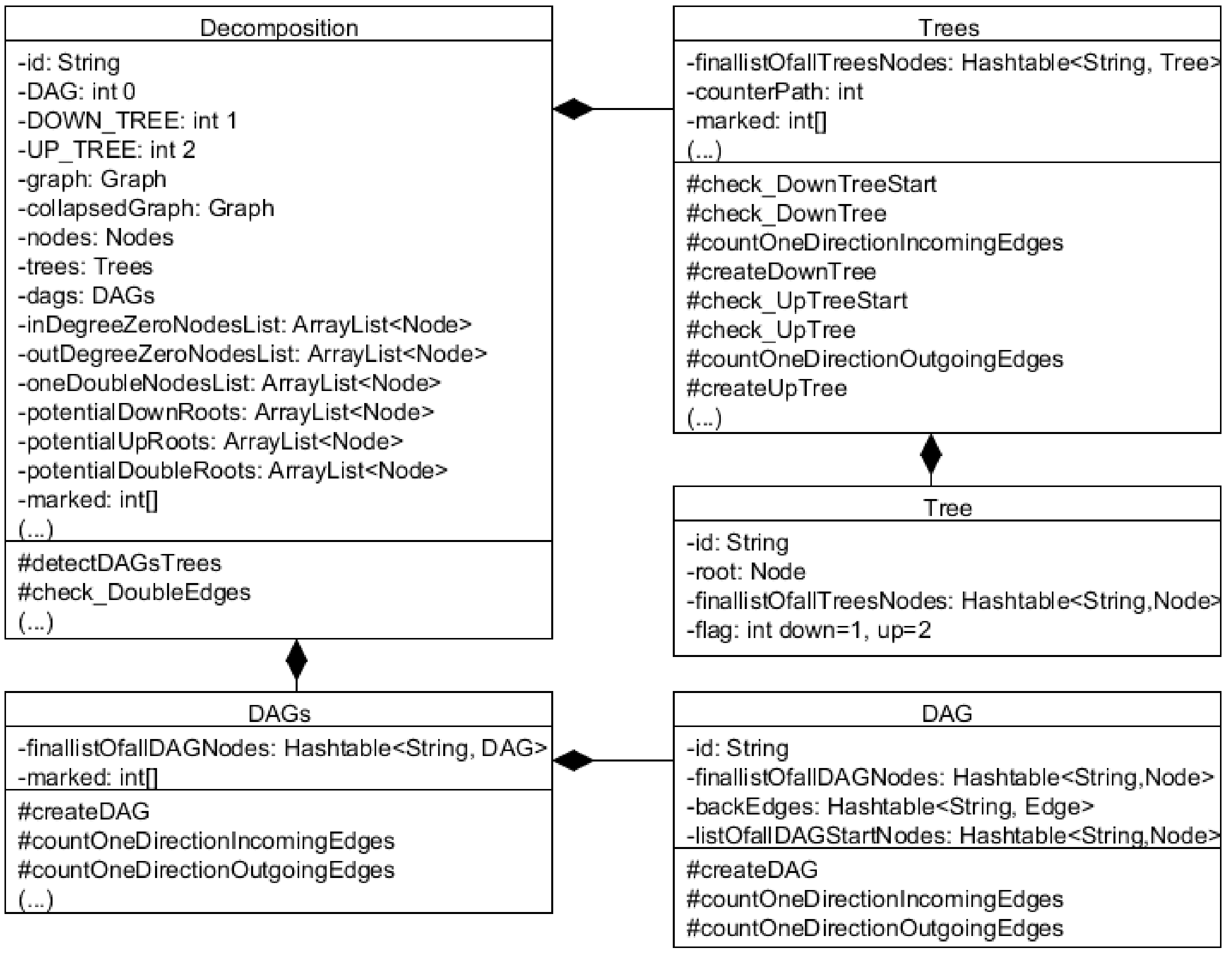

Decomposition
-id: String
-DAG: int 0
-DOWN_TREE: int 1
-UP_TREE: int 2
-graph: Graph
-collapsedGraph: Graph
-nodes: Nodes
-trees: Trees
-dags: DAGs
-inDegreeZeroNodesList: ArrayList<Node>
-outDegreeZeroNodesList: ArrayList<Node>
-oneDoubleNodesList: ArrayList<Node>
-potentialDownRoots: ArrayList<Node>
-potentialUpRoots: ArrayList<Node>
-potentialDoubleRoots: ArrayList<Node>
-marked: int[]
(...)
#detectDAGsTrees
#check_DoubleEdges
(...)
Trees
-finallistOfallTreesNodes: Hashtable<String, Tree>
-counterPath: int
-marked: int[]
(...)
#check_DownTreeStart
#check_DownTree
#countOneDirectionIncomingEdges
#createDownTree
#check_UpTreeStart
#check_UpTree
#countOneDirectionOutgoingEdges
#createUpTree
(...)
Tree
-id: String
-root: Node
-finallistOfallTreesNodes: Hashtable<String,Node>
-flag: int down=1, up=2
DAGs
-finallistOfallDAGNodes: Hashtable<String, DAG>
-marked: int[]
#createDAG
#countOneDirectionIncomingEdges
#countOneDirectionOutgoingEdges
(...)
DAG
-id: String
-finallistOfallDAGNodes: Hashtable<String,Node>
-backEdges: Hashtable<String, Edge>
-listOfallDAGStartNodes: Hashtable<String,Node>
#createDAG
#countOneDirectionIncomingEdges
#countOneDirectionOutgoingEdges

Figure 14.5: Class Diagram for detecting trees and DAGs.

- *marked*: an array for marking traversed nodes with path counter.

The global data attributes and data structures of the class “DAGs” are:

- *finallistOfallDAGNodes*: a hashtable (Table 14.2) containing DAGs.
- *marked*: an array for marking traversed nodes with path counter.

The global data attributes and data structures of the class “DAG” are:

- *id*: an id for the DAG.
- *finallistOfallDAGNodes*: a hashtable (Table 14.2) containing all nodes of the DAG.
- *backEdges*: a hashtable (Table 14.2) containing all back edges of the DAG.
- *listOfallDAGStartNodes*: a hashtable (Table 14.2) containing all start nodes of the DAG (roots).

The global data attributes and data structures of the class “Decomposition” are:

- *graph*: a directed graph.
- *trees*: all trees
- *dags*: all DAGs
- *inDegreeZeroNodesList*: an ArrayList (Table 14.1) containing all nodes with in-degree 0.
- *outDegreeZeroNodesList*: an ArrayList (Table 14.1) containing all nodes with out-degree 0.
- *oneDoubleNodesList*: an ArrayList (Table 14.1) containing all nodes with one double edge.
- *potentialDoubleRoots*: an ArrayList (Table 14.1) containing all potential root nodes having only double edges.
- *potentialDownRoots*: an ArrayList (Table 14.1) containing all potential root nodes having only outgoing edges ignoring double edges (in-degree 0 by forgetting double edges).
- *potentialUpRoots*: an ArrayList (Table 14.1) containing all potential root nodes having only incoming edges ignoring double edges (out-degree 0 by forgetting double edges).
- *DAG*: constant (0).
- *DOWN_TREE*: constant (1).
- *UP_TREE*: constant (2).
- *collapsedGraph*: a graph containing meta-nodes and meta-edges representing the decomposed subgraphs from this wCC and edges between them, respectively (related to build two-level hierarchy in Section 12.5).

A call graph of the primary algorithms provided by the “Trees” and “DAGs” classes and *detectDAGsTrees* (Algorithm 14.16) provided by the class “Decomposition” (Figure 14.5) is shown in Figure 14.6.

The *detectDAGsTrees* algorithm consists of several parts handling different cases of potential graph types:

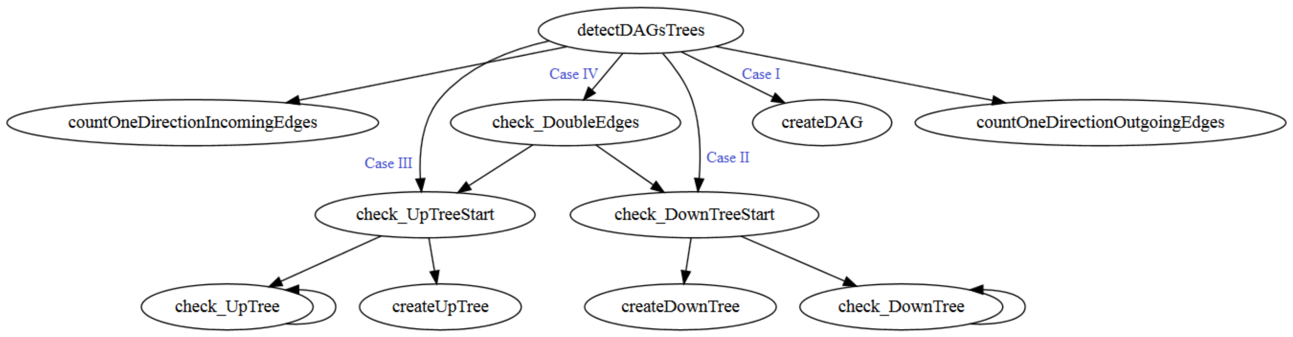


Figure 14.6: Call graph for the primary trees and DAGs detection algorithms.

- *detectDAGsTrees* (Algorithm 14.16)

  The local data attributes and data structures are:

  - *ccGraph*: a weakly connected component graph.

  It checks if the graph is a down-tree, an up-tree, or a DAG. It starts counting all nodes with in-degree 0, out-degree 0, only double edges, only outgoing edges, and only incoming edges (Lines 1-17). Then, *check_DownTreeStart* (Algorithm 14.17), *check_UpTreeStart* (Algorithm 14.20), or *check_DoubleEdges* (Algorithm 14.23) are called based on the counter values according to the logic shown in Table 12.1 (Lines 18-29).

- *check_DownTreeStart* (Algorithm 14.17)

  The local data attributes and data structures are:

  - *pathNodes*: an ArrayList (Table 14.1) containing path nodes.
  - *node*: the first node of *inDegreeZeroNodesList*.
  - *root*: the root node of the tree.
  - *result*: a boolean variable, true iff down-tree.

  It checks if the graph is a down-tree or a DAG. If the counter of in-degree 0 nodes equals one, then it marks all nodes as not visited storing the node with in-degree 0 as root, adding the node to the path nodes list, and calling *check_DownTree* (Algorithm 14.18) to check if the graph is a down-tree. Then, it calls *createDownTree* (Algorithm 14.24) to create a down-tree instance if one is found and it returns down-tree. Otherwise, it returns DAG (Lines 3-16). If the counter of in-degree 0 nodes does not equal one, it goes over all potential root nodes marking all nodes as not visited, clearing the path nodes list, storing the current node as root, adding the node to the path nodes list, and calling *check_DownTree* (Algorithm 14.18) to check if the graph is down-tree. Then, it calls *createDownTree* (Algorithm 14.24) to create down-tree instance if one is found and it returns down-tree. Finally, it returns DAG, if no down-tree is found (Lines 17-35).

- *check_DownTree* (Algorithm 14.18)

  The local data attributes and data structures are:

  - *result*: a boolean variable, true iff down-tree.
  - *outgoingEdge*: outgoing edge.

For the current node, it returns DAG if it is marked with *counterPath* (Lines 1-3). Then, for early termination, it checks if the incoming edges number of the current node (excluding double edges and incoming edge from its parent) is greater than 0 by calling Algorithm 14.19 (Lines 4-6). After marking the node with *counterPath*, it goes over all outgoing edges of the node (applying depth first search) adding the target node to the path nodes list, calling *check_DownTree* Algorithm 14.18, and removing the target node from path nodes list (Lines 9-22).

- *countOneDirectionIncomingEdges* (Algorithm 14.19)

  The local data attributes and data structures are:

  - *count*: numeric variable storing the number of one direction incoming edges for a node.
  - *incomingEdge*: incoming edge.

  It counts the incoming edges (excluding double edges and incoming edge from its parent) of current node.

- *createDownTree* (Algorithm 14.24)

  The local data attributes and data structures are:

  - *root*: the root of the tree.
  - *nodes*: the list of tree nodes.
  - *htableDownTree*: a hashtable (Table 14.2) containing all tree nodes.
  - *downTree*: Tree instance.

  It creates a down-tree instance storing all nodes as a hashtable.

- *check_UpTreeStart* (Algorithm 14.20)

  The local data attributes and data structures are:

  - *pathNodes*: an ArrayList (Table 14.1) containing path nodes.
  - *node*: the first node of *outDegreeZeroNodesList*.
  - *root*: the root node of the tree.
  - *result*: a boolean variable, true iff up-tree.

  It checks if the graph is an up-tree or a DAG analogously to *check_DownTreeStart* (Algorithm 14.17). If the counter of nodes with out-degree 0 equals one, then it marks all nodes as not visited storing the node with out-degree 0 as root, adding the node to the path nodes list, and calling *check_UpTree* (Algorithm 14.21) to check if the graph is an up-tree. Then, it calls *createUpTree* (Algorithm 14.25) to create an up-tree instance if one is found and it returns up-tree. Otherwise, it returns DAG (Lines 3-16).

  If the counter of nodes with out-degree 0 does not equal one, it goes over all potential root nodes marking all nodes as not visited, clearing the path nodes list, storing the current node as root, adding the node to the path nodes list, and calling *check_UpTree* (Algorithm 14.21) to check if the graph is up-tree. Then, it calls *createUpTree* (Algorithm 14.25) to create up-tree instance if one is found and it returns up-tree. Finally, it returns DAG, if no up-tree is found (Lines 17-35).

- *check_UpTree* (Algorithm 14.21)

  The local data attributes and data structures are:

  - *result*: a boolean variable, true iff up-tree.
  - *incomingEdge*: an incoming edge.

  Analogously to *check_DownTree* (Algorithm 14.18), for the current node, it returns DAG if it is marked with *counterPath* (Lines 1-3). Then, for early termination, it checks if the Outgoing edges number of the current node (excluding double edges and Outgoing edge from its parent) is greater than 0 by calling Algorithm 14.22 (Lines 4-6). After marking the node with *counterPath*, it goes over all Incoming edges of the node (applying depth first search) adding the source node to the path nodes list, calling *check_UpTree* Algorithm 14.21, and removing the source node from path nodes list (Lines 9-22).

- *countOneDirectionOutgoingEdges* (Algorithm 14.22)

  The local data attributes and data structures are:

  - *count*: a numeric variable storing the number of one direction outgoing edges for a node.
  - *outgoingEdge*: an outgoing edge.

  It counts the Outgoing edges (excluding double edges and Outgoing edge from its parent) of current node.

- *createUpTree* (Algorithm 14.25)

  The local data attributes and data structures are:

  - *root*: the root of the tree.
  - *nodes*: the list of tree nodes.
  - *htableUpTree*: a hashtable (Table 14.2) containing all tree nodes.
  - *upTree*: Tree instance.

  It creates an up-tree instance storing all nodes as hashtable.

- *check_DoubleEdges* (Algorithm 14.23)

  The local data attributes and data structures are:

  - *result*: a number giving the graph type.

  It calls *check_DownTreeStart* (Algorithm 14.17) to check if the graph is a down-tree or a DAG. If a DAG is found, it calls *check_UpTreeStart* (Algorithm 14.20) to finally check if the graph is an up-tree or a DAG.

- *createDAG* (Algorithm 14.26)

  The local data attributes and data structures are:

  - *htableDAG*: a hashtable (Table 14.2) containing all DAG nodes.
  - *dag*: a DAG instance.
  - *htableStartRootNodesDAG*: a hashtable (Table 14.2) containing all roots of the DAG.

  It creates a DAG instance storing all DAG nodes in a hashtable.

Algorithm 14.16 *detectDAGsTrees*

Description: detect DAGs Trees Algorithm
Input: *ccGraph* **weakly connected component**
Output: Determine type of wCC (up-tree, down-tree, DAG)

```
for all $node \in$ all nodes of $ccGraph$ having in-degree 0 do
    Increment $inDegreeZeroNumber$ by 1
    Add $node$ to $inDegreeZeroNodesList$ ArrayList
end for
for all $node \in$ all nodes of $ccGraph$ having out-degree 0 do
    Increment $outDegreeZeroNumber$ by 1
    Add $node$ to $outDegreeZeroNodesList$ ArrayList
end for
for all $node \in$ all nodes of $ccGraph$ having only double edges do
    Add $node$ to $potentialDoubleRoots$ ArrayList
end for
for all $node \in$ all nodes of $ccGraph$ having only outgoing edges do
    Add $node$ to $potentialDownRoots$ ArrayList
end for
for all $node \in$ all nodes of $ccGraph$ having only incoming edges do
    Add $node$ to $potentialUpRoots$ ArrayList
end for
if $inDegreeZeroNumber >= 2 \wedge outDegreeZeroNumber >= 2$ then
    Mark $result$ as DAG
else if $inDegreeZeroNumber < 2 \wedge outDegreeZeroNumber >= 2$ then
    $result \leftarrow check\_DownTreeStart\ (ccGraph)$ (Algorithm 14.17)
else if $inDegreeZeroNumber >= 2 \wedge outDegreeZeroNumber < 2$ then
    $result \leftarrow check\_UpTreeStart(ccGraph)$ (Algorithm 14.20)
else if $inDegreeZeroNumber < 2 \wedge outDegreeZeroNumber < 2$ then
    $result \leftarrow check\_DoubleEdges(ccGraph)$ (Algorithm 14.23)
end if
if $result = DAG$ then
    Call $createDAG(ccGraph)$(Algorithm 14.26)
end if
```

Algorithm 14.17 *check_DownTreeStart*

Description: Start check down tree
Input: *graph*
Output: 1 iff **DOWN TREE , 0** iff **DAG**

```
 1: Increment $counterPath$ by 1
 2: Create $pathNodes$ list
 3: if $inDegreeZeroNumber == 1$ then
 4:    for all $node \in graph$ do
 5:       Mark node as not visited with value $-1$
 6:    end for
 7:    $node \leftarrow$ first node of $inDegreeZeroNumber$
 8:    $root \leftarrow node$
 9:    Add $node$ to $pathNodes$
10:    $result \leftarrow check\_DownTree(pathNodes, node, marked)$ (Algorithm 14.18)
11:    if $result$ then
12:       Call $createDownTree(root, nodes)$ (Algorithm 14.24)
13:       return $DOWN\_TREE$
14:    else
15:       return $DAG$
16:    end if
17: else
18:    add $inDegreeZeroNodesList$ to $potentialOrderedRootsNodesList$
19:    add $potentialDownRoots$ at end of $potentialOrderedRootsNodesList$
20:    add $potentialDoubleRoots$ at end of $potentialOrderedRootsNodesList$
21:    for all $node \in potentialOrderedRootsNodesList$ do
22:       for all $node \in graph$ do
23:          Mark node as not visited with value $-1$
24:       end for
25:       Clear $pathNodes$
26:       $root \leftarrow$ current node
27:       Add node to $pathNodes$
28:       $result \leftarrow check\_DownTree(pathNodes, node, marked)$ (Algorithm 14.18)
29:       if $result$ then
30:          Call $createDownTree(root, nodes)$ (Algorithm 14.24)
31:          return $DOWN\_TREE$
32:       end if
33:    end for
34:    return $DAG$
35: end if
```

Algorithm 14.18 *check_DownTree*
Description: Check down-tree
Input: *pathNodes*, *node*, *marked*
Output: True *iff* **DOWN TREE, False** *iff* **DAG**

```
1: if $node$ is marked with $counterPath$ flag then
2:     return $False$ {DAG}
3: end if
       {For early termination, it checks if $node$ has many Incoming edges (excluding double edges and Incoming edge from its parent)}
4: if size of $pathNodes > 1 \wedge countOneDirectionIncomingEdges(node, parentnode) > 0$ (Algorithm 14.19) then
5:     return $False$ {DAG}
6: end if
7: Mark $node$ with value equal to $counterPath$ flag
8: $result \leftarrow True$
9: for all $outgoingEdge \in$ Outgoing edges of $node$ do
10:    $targetNodeOfOutgoingEdge \leftarrow$ target node of $outgoingEdge$
11:    Add the target node of $outgoingEdge$ to $pathNodes$
12:    $first \leftarrow$ the first occurrence position (index) for last node in the path
13:    $last \leftarrow$ the last occurrence position (index) for last node in the path
14:    if $first > -1 \wedge last > -1 \wedge last - first = 2$ then{Ignore double edge}
15:        continue
16:    end if
17:    $result \leftarrow check\_DownTree$ ($pathNodes$, the target node of $outgoingEdge$, $marked$) (Algorithm 14.18)
18:    Remove last occurrence of the target node of $outgoingEdge$ from $pathNodes$
19:    if $\neg result$ then
20:        return $False$ {DAG}
21:    end if
22: end for
23: return $True$
```

Algorithm 14.19 *countOneDirectionIncomingEdges*
Description: Count one direction incoming edges
Input: *node*, *parentNode*
Output: one direction incoming Edges number of *node*

```
1: $count \leftarrow 0$
2: for all $incomingEdge \in$ all incoming edges of $node$ do
3:     if start node of $incomingEdge \neq parentNode \wedge \neg node \in$ all In Neighbors of start node of $incomingEdge$ then
4:         Increment $count$ by 1
5:     end if
6: end for
7: return $count$
```

Algorithm 14.20 *check_UpTreeStart*

Description: Start check up-tree
Input: *graph*
Output: 1 iff **UP TREE , 0** iff **DAG**

```
if $outDegreeZeroNumber == 1$ then
```

```
Increment $counterPath$ by 1
Create $pathNodes$
if $outDegreeZeroNumber == 1$ then
    for all $node \in graph$ do
        Mark $node$ as not visited with value $-1$
    end for
    $node \leftarrow$ first node of $outDegreeZeroNumber$
    $root \leftarrow node$
    Add $node$ to $pathNodes$
    $result \leftarrow check\_UpTree(pathNodes, node, marked)$ (Algorithm 14.21)
    if $result$ then
        Call $createUpTree(root, nodes)$ (Algorithm 14.25)
        return $UP\_TREE$
    else
        return $DAG$
    end if
else
    add $outDegreeZeroNodesList$ to $potentialOrderedRootsNodesList$
    add $potentialUpRoots$ at end of $potentialOrderedRootsNodesList$
    add $potentialDoubleRoots$ at end of $potentialOrderedRootsNodesList$
    for all $node \in potentialOrderedRootsNodesList$ do
        for all $node \in graph$ do
            Mark $node$ as not visited with value $-1$
        end for
        Clear $pathNodes$
        $root \leftarrow node$
        Add $node$ to $pathNodes$
        $result \leftarrow check\_UpTree(pathNodes, node, marked)$ (Algorithm 14.21)
        if $result$ then
            Call $createUpTree(root, nodes)$ (Algorithm 14.25)
            return $UP\_TREE$
        end if
    end for
    return $DAG$
end if
```

Algorithm 14.21 *check_UpTree*

Description: Check up-tree
Input: *pathNodes*, *node*, *marked*
Output: True *iff* **UP TREE, False** *iff* **DAG**

```
1: if $node$ is marked with $counterPath$ flag then
2:     return $False$ {DAG}
3: end if
       {For early termination, it checks if $node$ has many Outgoing edges (excluding double edges and Outgoing edge to its parent)}
4: if size of $pathNodes > 1 \wedge countOneDirectionOutgoingEdges(node, parentnode) > 0$ (Algorithm 14.22) then
5:     return $False$ {DAG}
6: end if
7: Mark $node$ with value equal to $counterPath$ flag
8: $result \leftarrow True$
9: for all $incomingEdge \in$ Incoming edges of $node$ do
10:    $sourceNodeOfIncomingEdge \leftarrow$ source node of $incomingEdge$
11:    Add the source node of $incomingEdge$ to $pathNodes$
12:    $first \leftarrow$ the first occurrence position (index) for last node in the path
13:    $last \leftarrow$ the last occurrence position (index) for last node in the path
14:    if $first > -1 \wedge last > -1 \wedge last - first = 2$ then {Ignore Double Edge}
15:        continue
16:    end if
17:    $result \leftarrow check\_UpTree$ ($pathNodes$, the source node of $incomingEdge$, $marked$) (Algorithm 14.21)
18:    Remove last occurrence of the source node of incoming edge from $pathNodes$
19:    if $\neg result$ then
20:        return $False$ {DAG}
21:    end if
22: end for
23: return $True$
```

Algorithm 14.22 *countOneDirectionOutgoingEdges*

Description: Count one direction outgoing edges
Input: *node*, *parentNode*
Output: one direction outgoing Edges number of *node*

```
1: $count \leftarrow 0$
2: for all $outgoingEdge \in$ all outgoing edges of $node$ do
3:     if end node of $outgoingEdge \neq parentNode \wedge \neg node \in$ all Out Neighbors of end node of $outgoingEdge$ then
4:         Increment $count$ by 1
5:     end if
6: end for
7: return $count$
```

Algorithm 14.23 *check_DoubleEdges*

Description: Check double edges
Input: *graph*
Output: return graph type (up-tree, down-tree, DAG)

```
1: $result \leftarrow check\_DownTreeStart\ (graph)$ (Algorithm 14.17)
2: if $result = DAG$ then
3:     $result \leftarrow check\_UpTreeStart\ (graph)$ (Algorithm 14.20)
4: end if
5: return $result$
```

Algorithm 14.24 *createDownTree*

Description: Create down-tree
Input: *root*, *nodes*
Output: down-tree

```
1: Initialize $htableDownTree$ hash table
2: Add tree $nodes$ to $htableDownTree$
3: Create $downTree$ Tree instance
4: Put $downTree$ in $finallistOfallTreesNodes$
```

Algorithm 14.25 *createUpTree*

Description: Create up-tree
Input: *root*, *nodes*
Output: up-tree

```
1: Initialize $htableUpTree$ hash table
2: Add tree $nodes$ to $htableUpTree$
3: Create $upTree$ instance
4: Put $upTree$ in $finallistOfallTreesNodes$
```

Algorithm 14.26 *createDAG*

Description: Create DAG
Input: *dagGraph*
Output: DAG

```
1: Initialize $htableDAG$ hash table
2: Add the nodes of $dagGraph$ to $htableDAG$
3: Create DAG instance
4: Put DAG in $finallistOfallDAGNodes$
```

14.1.4 Algorithms of Hierarchy Construction

The class diagram in Figure 14.7 shows an overview of the implementation part of the hierarchy construction. The Decomposition, Trees, and DAGs classes list the names of algorithms (operations) used to build two-level hierarchy.

The call graph of the primary algorithms to create meta-nodes and meta-edges provided by the class Decomposition is shown in Figure 14.8.

A call graph of primary algorithms to decompose the meta-graph using breadth first search is shown in Figure 14.9.

The algorithm consists of several parts:

- *startCreateMetaNodes* (Algorithm 14.27)

  The local data attributes and data structures are:

  - *allCycles*: a hashtable (Table 14.2) containing all cycles.
  - *allTrees*: a hashtable (Table 14.2) containing all trees.
  - *allDAGs*: a hashtable (Table 14.2) containing all DAGs.
  - *notCollapsedGraph*: a graph containing the original nodes and edges.
  - *newUpperLevelCollapsedFeatureGraph*: a graph for storing all collapsed components nodes.

  It starts creating a meta-node for each decomposed subgraph.

  - if the graph has cycles, it will call *createMetaNodesByBFS* Algorithm 14.28 to start collapsing components.
  - otherwise, if it has one tree or DAG, it will create only one meta-node.

- *createMetaNodesByBFS* (Algorithm 14.28)

  The local data attributes and data structures are:

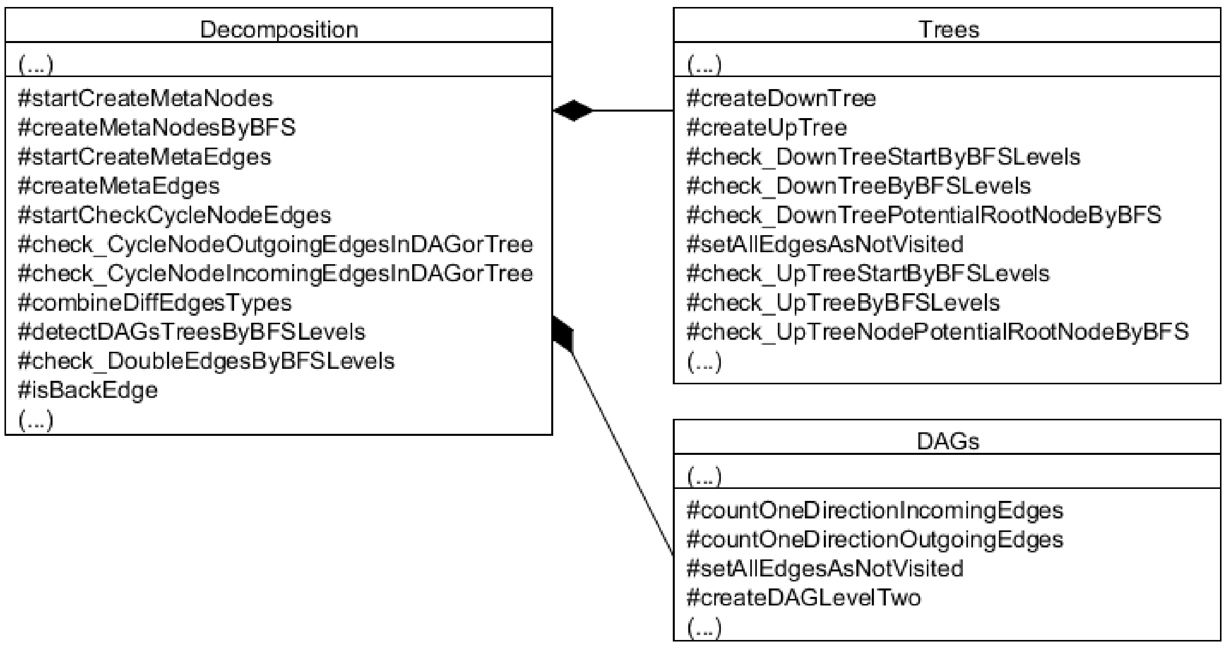
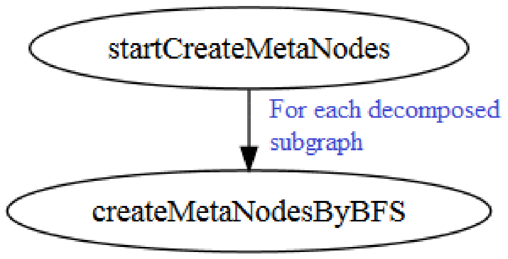
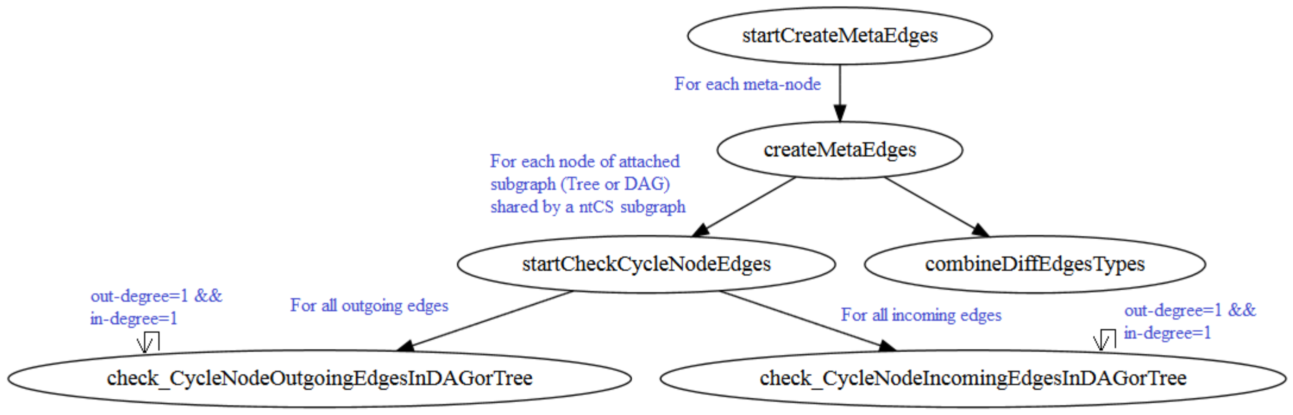

Decomposition
(...)
#startCreateMetaNodes
#createMetaNodesByBFS
#startCreateMetaEdges
#createMetaEdges
#startCheckCycleNodeEdges
#check_CycleNodeOutgoingEdgesInDAGorTree
#check_CycleNodeIncomingEdgesInDAGorTree
#combineDiffEdgesTypes
#detectDAGsTreesByBFSLevels
#check_DoubleEdgesByBFSLevels
#isBackEdge
(...)
Trees
(...)
#createDownTree
#createUpTree
#check_DownTreeStartByBFSLevels
#check_DownTreeByBFSLevels
#check_DownTreePotentialRootNodeByBFS
#setAllEdgesAsNotVisited
#check_UpTreeStartByBFSLevels
#check_UpTreeByBFSLevels
#check_UpTreeNodePotentialRootNodeByBFS
(...)
DAGs
(...)
#countOneDirectionIncomingEdges
#countOneDirectionOutgoingEdges
#setAllEdgesAsNotVisited
#createDAGLevelTwo
(...)

Figure 14.7: Class diagram for two-level hierarchy construction.



(a) Call graph for creating meta-nodes


startCreateMetaEdges
For each meta-node
createMetaEdges
For each node of attached subgraph (Tree or DAG) shared by a ntCS subgraph
startCheckCycleNodeEdges
combineDiffEdgesTypes
out-degree=1 && in-degree=1
For all outgoing edges
For all incoming edges
out-degree=1 && in-degree=1
check_CycleNodeOutgoingEdgesInDAGorTree
check_CycleNodeIncomingEdgesInDAGorTree

(b) Call graph for creating meta-edges

Figure 14.8: Call graphs for creating meta-graph

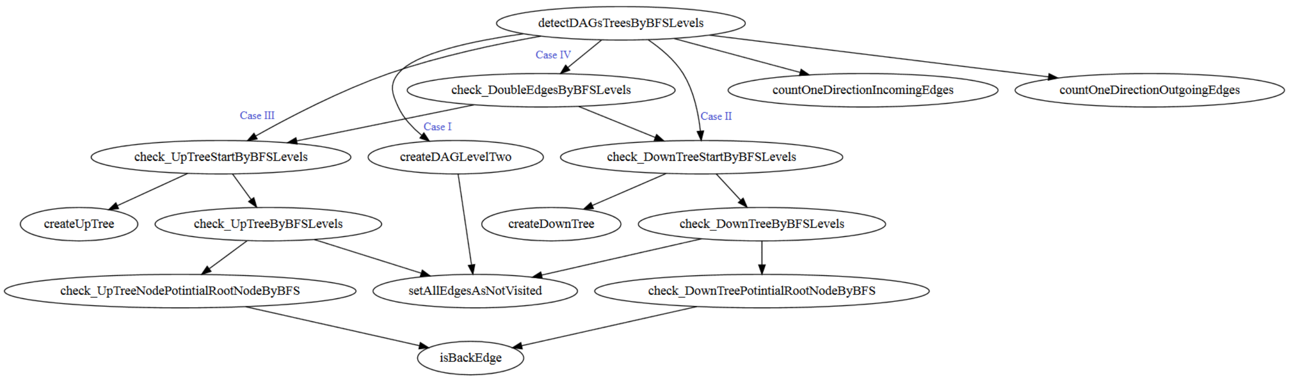

detectDAGsTreesByBFSLevels
Case IV
check_DoubleEdgesByBFSLevels
countOneDirectionIncomingEdges
countOneDirectionOutgoingEdges
Case III
Case I
Case II
check_UpTreeStartByBFSLevels
createDAGLevelTwo
check_DownTreeStartByBFSLevels
createUpTree
check_UpTreeByBFSLevels
createDownTree
check_DownTreeByBFSLevels
check_UpTreeNodePotintialRootNodeByBFS
setAllEdgesAsNotVisited
check_DownTreePotintialRootNodeByBFS
isBackEdge

Figure 14.9: Call graph for detecting tree and DAG using breadth first search.

– *newUpperLevelCollapsedFeatureGraph*: a graph for storing all collapsed components nodes.
– *notCollapsedGraph*: a graph containing the original nodes and edges.
– *key*: the key of a component.
– *q*: a Deque (Table 14.3) containing keys of components.
– *connections*: all decomposed subgraphs sharing nodes with a specific decomposed subgraph.
– *connectedTo*: a hashtable (Table 14.2) containing a pair of decomposed subgraphs sharing nodes.
– *visited*: a hashset (Table 14.4) of visited nodes.
– *nodeCounter*: a counter of the meta-nodes that were created.

It performs breadth first search over connections (shared nodes with other topological components) of topological components.

  - it will add the current topological component to q (Line 6).
  - it goes over all component keys removing its elements until it becomes empty (Lines 8-41).
    * if the current topological component is cycle, it will get current cycle connections, it will create a new meta-node adding it to $newUpperLevelCollapsedFeature Graph$.
    * if the current topological component is tree, it will get current tree connections, it will create a new meta-node adding it to $newUpperLevelCollapsedFeatureGraph$.
    * if the current topological component is DAG, it will get current DAG connections, it will create a new meta-node adding it to $newUpperLevelCollapsedFeature Graph$.
    * if current topological component has connections, it will visit them adding them to q.

- $startCreateMetaEdges$ (Algorithm 14.29)

  The local data attributes and data structures are:

  - $newUpperLevelCollapsedFeatureGraph$: a graph containing the collapsed components as meta-nodes but it still has no meta-edges.
  - $notCollapsedGraph$: a graph containing the original nodes and edges.

  It starts creating meta-edges between meta-nodes.

  - it goes over all nodes of $newUpperLevelCollapsedFeatureGraph$ (Lines 1-6).
  - if the current topological component is a tree or a DAG, it will call $createMetaEdges$ (Algorithm 14.30) to construct new meta-edges (Lines 3-5).

- $createMetaEdges$ (Algorithm 14.30)

  The local data attributes and data structures are:

  - key: the key of a component.
  - $notCollapsedGraph$: a graph containing the original nodes and edges.
  - $newUpperLevelCollapsedFeatureGraph$: a graph containing the collapsed components as nodes without edges.
  - $treeOrDAGCollapsedNode$: a collapsed node of a tree or DAG component.
  - $listOfConnectionsKeys$: a Deque (Table 14.3) containing the tree or DAG nodes shared by cycles.
  - $incomingEdges$: a hashtable (Table 14.2) to store single incoming edges.
  - $outgoingEdges$: a hashtable (Table 14.2) to store single outgoing edges.
  - $doubleEdgesType1$: a hashtable (Table 14.2) to store double edges (type 1).
  - $doubleEdgesType2\_3$: a hashtable (Table 14.2) to store double edges representing two different direction paths between DAG and cycle (type2) and blocking edges (type3).

  It creates new meta-edges of the collapsed component.

  - it performs depth first search using $listOfConnectionsKeys$ of component (Lines 10-29).
    - for each node of tree component (DAG) shared by a certain cycle, it calls $startCheckCycleNodeEdges$ (Algorithm 14.31) to check edges of current node (Lines 20-27).
    - it considers different combination for found edges by calling $combineDiffEdges$ $Types$ (Algorithm 14.32) (Line 28).

- $startCheckCycleNodeEdges$ (Algorithm 14.31)

  The local data attributes and data structures are:

  - $nodeConnCycle$: a node of a tree or DAG component shared by cycle.
  - $treeOrDAGCollapsedNode$: a collapsed node of a tree or DAG component.
  - $cycleCollapsedNode$: a meta-node representing a cycle.
  - $tmpoutgoingEdges$: a temporary hashtable (Table 14.2) to store single outgoing edges.
  - $tmpincomingEdges$: a temporary hashtable (Table 14.2) to store single incoming edges.
  - $doubleEdgesType1$: a hashtable (Table 14.2) to store double edges (type 1).
  - $doubleEdgesType2\_3$: a hashtable (Table 14.2) to store double edges representing two different direction paths between DAG and cycle (type2) and blocking edge (type3).

  It checks edges of current node.

  - It goes over all outgoing edges calling $check\_CycleNodeOutgoingEdgesInDAGorTree$ Algorithm 14.33 (Lines 2-11).
  - It goes over all incoming edges calling $check\_CycleNodeIncomingEdgesInDAGorTree$ Algorithm 14.34 (Lines 14-23).

- $check\_CycleNodeOutgoingEdgesInDAGorTree$ (Algorithm 14.33)

  The local data attributes and data structures are:

  - $firstNode$: first node (node shared by component and cycle).
  - $currentNode$: current node along a path in component.
  - $treeOrDAGCollapsedNode$: a collapsed node of component.
  - $cycleCollapsedNode$: a collapsed node of cycle component.
  - $tmpoutgoingEdges$: a temporary hashtable (Table 14.2) to store single outgoing edges.
  - $doubleEdgesType2\_3$: a hashtable (Table 14.2) to store double edges representing two different direction paths between component and cycle (type2) and blocking edges (type3).

  It checks the outgoing edges starting from $firstNode$ to determine the type of the meta-edge connected to DAGs and trees, if it is outgoing edge or blocking edge.

  - if $currentNode$ has out-degree > 1 or is cycle node or out-degree $= 0$ and in-degree $= 1$, it returns outgoing edge (Lines 1-5).

  - if *currentNode* has in-degree >= 2, it returns block edge if out-degree = 0 and it returns outgoing edge if out-degree > 0 (Lines 6-16).
  - if *currentNode* has out-degree = 1 and in-degree = 1, it continues (Lines 17-21).

- *check_CycleNodeIncomingEdgesInDAGorTree* (Algorithm 14.34)

  The local data attributes and data structures are:

  - *firstNode*: first node (node shared by component and cycle).
  - *currentNode*: current node along a path in component.
  - *treeOrDAGCollapsedNode*: a collapsed node of DAG component.
  - *cycleCollapsedNode*: a collapsed node of cycle component.
  - *tmpincomingEdges*: a temporary hashtable (Table 14.2) to store single incoming edges.
  - *doubleEdgesType*2_3: a hashtable (Table 14.2) to store double edges representing two different direction paths between component and cycle (type2) and blocking edges (type3).

  It checks the incoming edges starting from *firstNode* to determine the type of the meta-edge connected to DAGs and trees, if it is incoming edge or blocking edge.

  - if *currentNode* has in-degree > 1 or is cycle node or in-degree= 0 and out-degree= 1, it returns incoming edge.
  - if *currentNode* has out-degree >= 2, it returns block edge if in-degree = 0 and it returns incoming edge if in-degree > 0.
  - if *currentNode* has out-degree = 1 and in-degree = 1, it continues.

- *combineDiffEdgesTypes* (Algorithm 14.32)

  The local data attributes and data structures are:

  - *tmpoutgoingEdges*: a temporary hashtable (Table 14.2) to store single outgoing edges.
  - *tmpincomingEdges*: a temporary hashtable (Table 14.2) to store single incoming edges.
  - *doubleEdgesType*1: a hashtable (Table 14.2) to store double edges (type 1).
  - *doubleEdgesType*2_3: a hashtable (Table 14.2) to store double edges representing two different direction paths between DAG and cycle (type2) and blocking edges (type3).
  - *newUpperLevelCollapsedFeatureGraph*: a graph containing the collapsed components as nodes without edges.

  It considers different combinations for found edges.

  - It creates double edge, if number of double edges in *doubleEdgesType*1 is larger than zero (Lines 4-6).
  - If the number of *tmpincomingEdges* is larger than zero
    * it creates double path edge, if the number of *tmpoutgoingEdges* is larger than zero (Lines 8-10).

    * it creates incoming edge, otherwise (Lines 11-13).
  - It creates outgoing edge, if number of edges in *tmpoutgoingEdges* is larger than zero (Lines 15-17).
  - Otherwise, it creates blocking edge (Lines 18-20).

The detection of tree and DAG in level two is similar to level one except it follows breadth first search instead of depth first search with the following modifications in the detection algorithm.

- *check_DownTreeByBFSLevelsAlgorithm* (Algorithm 14.35)

  The local data attributes and data structures are:

  - *pathEdges*: a hashtable (Table 14.2) to store visited edges.
  - *htableStartRootNodes*: a hashtable (Table 14.2) to store root nodes.
  - *graph*: the directed graph to be checked.
  - *visited*: a hashset (Table 14.4) containing the visited nodes.
  - *potentialOrderedRootsNodesList*: an Arraylist (Table 14.1) containing potential root nodes.

  It checks if a component is a down-tree using breadth first search (Lines 4-29):

  - if there is only one node having in-degree equals zero, than call *check_DownTreePotentialRootNodeByBFS* (Algorithm 14.36) using this node as root (Lines 4-12).
  - otherwise call *check_DownTreePotentialRootNodeByBFS* (Algorithm 14.36) for all nodes of *potentialOrderedRootsNodesList* (Lines 14-29).

- *check_DownTreePotentialRootNodeByBFS* (Algorithm 14.36)

  The local data attributes and data structures are:

  - *potentialRootNode*: a potential root node.
  - *pathEdges*: a hashtable (Table 14.2) to store visited edges.
  - *visited*: a hashset (Table 14.4) containing visited nodes.
  - *prevLevel*: a hashtable (Table 14.2) storing visited nodes of previous level.
  - *nextLevelVisited*: a hashtable (Table 14.2) storing visited nodes of current level.

  It performs breadth first search starting from *potentialRootNode* to check if a current node is visited.

  - it adds *potentialRootNode* to *prevLevel* (Line 2).
  - while *prevLevel* has elements:
    * it creates new *nextLevelVisited* (Line 5).
    * while *prevLevel* has elements:
      · it removes one node from *prevLevel* (Line 8).
      · it goes through all unvisited outgoing edges of current node (Line 9-25).
      · if target node of current node is visited, it return false (Lines 16-17).
      · otherwise it adds target node to *visited* and *nextLevelVisited* (Lines 18-22).

        ∗ *prevLevel* is assigned *nextLevelVisited* (Line 27).

    – it returns true (Line 29).

- *setAllEdgesAsNotVisited* (Algorithm 14.37)

  The local data attributes and data structures are:

    – *graph*:the directed graph to be changed.

  It sets all edges in the graph as visited.

Algorithm 14.27 *startCreateMetaNodes*

Description: Start creating meta-nodes of meta-graph
Input: *notCollapsedGraph*
Output: Collapsed topological components as meta-nodes

```
1: $allCycles \leftarrow$ hashtable of all cycles
2: $allTrees \leftarrow$hashtable of all trees
3: $allDAGs \leftarrow$ hashtable of all DAGs
4: $newUpperLevelCollapsedComponentGraph \leftarrow$ an empty Graph
5: if # of $allCycles$ elements $> 0$ then
6:     $key \leftarrow$ first cycle key
7:     Call $createMetaNodesByBFS(newUpperLevelCollapsedComponentGraph,$ $notCollapsedGraph$, $key$, null) (Algorithm 14.28)
8: else
9:     # elements of $allTrees \vee allDAGs = 1$
10:    Create one node $parentNode$
11:    if # elements of $allTrees = 1$ then
12:        for all $childNode \in$ all nodes of tree do
13:            add $childNode$ as child for $parentNode$
14:        end for
15:    else if # elements of $allDAGs = 1$ then
16:        for all $childNode \in$ all nodes of DAG do
17:            add $childNode$ as child for $parentNode$
18:        end for
19:    end if
20: end if
```

Algorithm 14.28 *createMetaNodesByBFS*

Description: Create meta-nodes by BFS
Input: *newUpperLevelCollapsedFeatureGraph*, *notCollapsedGraph*, *key*
Output: Collapsed topological components as meta-nodes in *newUpperLevelCollapsedFeatureGraph*

```
1:  connections ← null
2:  connectedTo ← a hashtable
3:  q ← a deque
4:  visited ← a hashset
5:  nodeCounter = −1
6:  add key to q
7:  add key to visited
8:  while q is not empty do
9:      key ← remove first element from q
10:     if key is cycle then
11:         connections ← the cycle nodes shared by trees and DAGs
12:         Create one meta-node parentNode
13:         for all childNode ∈ all nodes of cycle do
14:             add childNode as child for parentNode
15:         end for
16:     else if key is tree then
17:         connections ← the tree nodes shared by cycles
18:         Create one meta-node parentNode
19:         for all childNode ∈ all nodes of tree do
20:             add childNode as child for parentNode
21:         end for
22:     else if key is DAG then
23:         connections ← the DAG nodes shared by cycles
24:         Create one meta-node parentNode
25:         for all childNode ∈ all nodes of DAG do
26:             add childNode as child for parentNode
27:         end for
28:     end if
29:     add parentNode to newUpperLevelCollapsedFeatureGraph
30:     if connections is not null then
31:         for all childKey ∈ connections do
32:             if childKey ∉ visited then
33:                 add childKey to visited
34:                 add childKey to q
35:                 add childKey and key to connectedTo
36:             else
37:                 add childKey and key to connectedTo
38:             end if
39:         end for
40:     end if
41: end while
```

Algorithm 14.29 *startCreateMetaEdges*

Description: Start creating meta-edges
Input: *newUpperLevelCollapsedFeatureGraph*, *notCollapsedGraph*
Output: *newUpperLevelCollapsedFeatureGraph* **with new meta-edges**

```
1: for all $currentCollapsedNode \in newUpperLevelCollapsedFeatureGraph$ do
2:     $key \leftarrow$ key of $currentCollapsedNode$
3:     if $key$ is tree $\vee$ DAG then
4:         call $createMetaEdges(key, notCollapsedGraph, newUpperLevelCollapsedFeatureGraph, currentCollapsedNode)$ Algorithm 14.30
5:     end if
6: end for
```

Algorithm 14.30 *createMetaEdges*

Description: Create meta-edges of tree component (DAG)
Input: key, $notCollapsedGraph$, $newUpperLevelCollapsedFeatureGraph$, $treeOrDAGCollapsedNode$
Output: $newUpperLevelCollapsedFeatureGraph$ **with meta-edges**

```
1: if $key$ is tree then
2:    $connections$ ← the tree nodes shared by cycles
3:    $currentTree$ ← the tree of $key$
4: else if $key$ is DAG then
5:    $connections$ ← the DAG nodes shared by cycles
6:    $currentDAG$ ← the DAG of $key$
7: end if
8: if $connections$ is not null $\wedge \#$ elements of $connections > 0$ then
9:    $listOfConnectionsKeys$ ← all keys of $connections$
10:   while $listOfConnectionsKeys$ not empty do
11:      $incomingEdges$ ← all single incoming edges (still empty)
12:      $outgoingEdges$ ← all single outgoing edges (still empty)
13:      $doubleEdgesType1$ ← double edges type 1 (still empty)
14:      $doubleEdgesType2\_3$ ← double edges type 2 and blocking edges type 3 (still empty)
15:      $connKey$ ← remove first element from $listOfConnectionsKeys$
16:      $conn$ ← the tree component (DAG) nodes shared by cycle of key $connKey$
17:      $cycleCollapsedNode$ ← a node with key $connKey$ in $newUpperLevelCollapsedFeatureGraph$
18:      $tmpincomingEdges$ ← all potential incoming edges (still empty)
19:      $tmpoutgoingEdges$ ← all potential outgoing edges (still empty)
20:      for all $nodeConn \in conn$ do
21:         if $key$ is tree then
22:            $nodeConnCycle$ ← get from $currentTree$ nodes the node of $nodeConn$
23:         else if $key$ is DAG then
24:            $nodeConnCycle$ ← get from $currentDAG$ nodes the node of $nodeConn$
25:         end if
26:         Call $startCheckCycleNodeEdges$ ($nodeConnCycle$, $treeOrDAGCollapsedNode$, $cycleCollapsedNode$, $tmpoutgoingEdges$, $doubleEdgesType2\_3$, $tmpincomingEdges$) (Algorithm 14.31)
27:      end for
28:      Call $combineDiffEdgesTypes$ ($tmpoutgoingEdges$, $tmpincomingEdges$, $doubleEdgesType1$, $doubleEdgesType2\_3$, $newUpperLevelCollapsedFeatureGraph$) (Algorithm 14.32)
29:   end while
30: end if
```

Algorithm 14.31 *startCheckCycleNodeEdges*

Description: Start check cycle node edges

Input: *nodeConnCycle*, *treeOrDAGCollapsedNode*, *cycleCollapsedNode*, *tmpoutgoingEdges*, *tmpincomingEdges*, *doubleEdgesType1*, *doubleEdgesType2_3*

Output: *tmpoutgoingEdges*, *tmpincomingEdges*, *doubleEdgesType1*, *doubleEdgesType2_3*

```
1: if out-degree of $nodeConnCycle > 0$ then
2:   for all $outEdge \in$ outgoing edges of $nodeConnCycle$ do
3:     $isDoubleEdge \leftarrow$ Call $isBackEdge$(incoming edges of $nodeConnCycle$, $outEdge$) (Algorithm 14.38)
4:     if $isDoubleEdge$ is true then
5:       Create double edge (doubleTyp1)
6:       Add the double edge to $doubleEdgesType1$
7:     else
8:       $targetNode \leftarrow$ end node of $outEdge$
9:       $check\_CycleNodeOutgoingEdgesInDAGorTree$($nodeConnCycle$, $targetNode$, $treeOrDAGCollapsedNode$, $cycleCollapsedNode$, $tmpoutgoingEdges$, $doubleEdgesType2\_3$) (Algorithm 14.33)
10:    end if
11:  end for
12: end if
13: if in-degree of $nodeConnCycle > 0$ then
14:   for all $inEdge \in$ incoming edges of $nodeConnCycle$ do
15:     $isDoubleEdge \leftarrow$ Call $isBackEdge$(outgoing edges of $nodeConnCycle$, $inEdge$) (Algorithm 14.38)
16:     if $isDoubleEdge$ is true then
17:       Create double edge (doubleTyp1)
18:       Add the double edge to $doubleEdgesType1$
19:     else
20:       $sourceNode \leftarrow$ start node of $inEdge$
21:       $check\_CycleNodeIncomingEdgesInDAGorTree$($nodeConnCycle$, $sourceNode$, $treeOrDAGCollapsedNode$, $cycleCollapsedNode$, $tmpincomingEdges$, $doubleEdgesType2\_3$) (Algorithm 14.34)
22:     end if
23:   end for
24: end if
```

Algorithm 14.32 *combineDiffEdgesType*

Description: Combine different edge types

Input: *tmpoutgoingEdges*, *tmpincomingEdges*, *doubleEdgesType*1, *doubleEdgesType*2_3, *newUpperLevelCollapsedFeatureGraph*

Output: combined edge added to *newUpperLevelCollapsedFeatureGraph*

```
1: $numberOfDoubleEdgesType1 \leftarrow$ number of double edges in $doubleEdgesType1$
2: $numberOfOutgoingEdges \leftarrow$ number of outgoing edges in $tmpoutgoingEdges$
3: $numberOfIncomingEdges \leftarrow$ number of incoming edges in $tmpincomingEdges$
4: if $numberOfDoubleEdgesType1 > 0$ then
5:    Get double edge from $doubleEdgesType1$
6:    Add the double edge to $newUpperLevelCollapsedFeatureGraph$
7: else if $numberOfIncomingEdges > 0$ then
8:    if $numberOfOutgoingEdges > 0$ then
9:       Create double path edge
10:      Add the double path edge to $newUpperLevelCollapsedFeatureGraph$
11:   else
12:      Get incoming edge from $tmpincomingEdges$
13:      Add the incoming edge to $newUpperLevelCollapsedFeatureGraph$
14:   end if
15: else if $numberOfOutgoingEdges > 0$ then
16:   Get outgoing edge from $tmpoutgoingEdges$
17:   Add the outgoing edge to $newUpperLevelCollapsedFeatureGraph$
18: else
19:   Get blocking edge from $doubleEdgesType2\_3$
20:   Add the blocking edge to $newUpperLevelCollapsedFeatureGraph$
21: end if
```

Algorithm 14.33 *check_CycleNodeOutgoingEdgesInDAGorTree*

Description: Check cycle node outgoing edges in DAG or tree component

Input: $firstNode$, $currentNode$, $treeOrDAGCollapsedNode$, $cycleCollapsedNode$, $tmpoutgoingEdges$, $doubleEdgesType2\_3$

Output: $tmpoutgoingEdges$, $doubleEdgesType2\_3$

```
1: if out-degree of $currentNode$ $>$ 1 $\vee$ $currentNode$ $\in$ cycles nodes $\vee$ (out-degree of $currentNode$ $= 0 \wedge$ in-degree of $currentNode$ $= 1$) then
2:   Create outgoing edge $newOutgoingEdge$ having $cycleCollapsedNode$ and $treeOrDAGCollapsedNode$ as start and end nodes
3:   Add $newOutgoingEdge$ to $tmpoutgoingEdges$
4:   Return
5: end if
6: if in-degree of $currentNode$ $>= 2$ then
7:   if out-degree of $currentNode$ $= 0$ then
8:     Create blocking edge (doubleTyp3)
9:     Add the blocking edge to $doubleEdgesType2\_3$
10:    Return
11:  else if out-degree of $currentNode$ $> 0$ then
12:    Create outgoing edge $newOutgoingEdge$
13:    Add $newOutgoingEdge$ to $tmpoutgoingEdges$
14:    Return
15:  end if
16: end if
17: if in-degree of $currentNode$ = out-degree of $currentNode \wedge$ out-degree of $currentNode$ $= 1$ then
18:   $outEdge \leftarrow$ outgoing edges of $currentNode$
19:   $targetNode \leftarrow$ target node of $outEdge$
20:   Call $check\_CycleNodeOutgoingEdgesInDAGorTree(firstNode, targetNode, treeOrDAGCollapsedNode, cycleCollapsedNode, tmpoutgoingEdges, doubleEdgesType2\_3)$ (Algorithm 14.33)
21: end if
```

Algorithm 14.34 *check_CycleNodeIncomingEdgesInDAGorTree*

Description: Check cycle node incoming edges in DAG or tree component

Input: $firstNode$, $currentNode$, $treeOrDAGCollapsedNode$, $cycleCollapsedNode$, $tmpincomingEdges$, $doubleEdgesType2\_3$

Output: $tmpincomingEdges$, $doubleEdgesType2\_3$

```
1: {This algorithm works analogously to Algorithm 14.33}
```

Algorithm 14.35 *check_DownTreeByBFSLevels*

Description: Check down-tree by BFS levels
Input: *pathEdges*, *htableStartRootNodes*, *graph*
Output: *htableStartRootNodes*, **return True iff** *graph* **is Down Tree**

```
1: $componentSize \leftarrow$ # of $graph$ nodes
2: $visited \leftarrow$ new HashSet
3: $result \leftarrow$ true
4: if # of nodes of in-degree zero = 1 then
5:     Clear $htableStartRootNodes$
6:     Add the node of in-degree zero to $htableStartRootNodes$ as root
7:     Add the root node to $visited$
8:     Call $setAllEdgesAsNotVisited(graph)$ (Algorithm 14.37)
9:     $result \leftarrow$ Call $check\_DownTreePotintialRootNodeByBFS(node, pathEdges, visited)$
10:    if $result \wedge$ # nodes in $visited = componentSize$ then
11:        Return true
12:    end if
13: else
14:    $potentialOrderedRootsNodesList \leftarrow$ all nodes of in-degree zero
15:    Add all nodes having # one direction incoming edges = zero at the end of $potentialOrderedRootsNodesList$
16:    Add all nodes having only double edges at the end of $potentialOrderedRootsNodesList$
17:    for all $node \in potentialOrderedRootsNodesList$ do
18:        Clear $htableStartRootNodes$
19:        Clear $visited$
20:        Clear $pathEdges$
21:        Add $node$ id to $visited$
22:        Call $setAllEdgesAsNotVisited(graph)$ (Algorithm 14.37)
23:        Add $node$ to $htableStartRootNodes$
24:        $result \leftarrow$ Call $check\_DownTreePotentialRootNodeByBFS(node, pathEdges, visited)$
25:        if $result \wedge$ # nodes in $visited = componentSize$ then
26:            Return true
27:        end if
28:    end for
29: end if
30: Return false
```

Algorithm 14.36 *check_DownTreePotentialRootNodeByBFS*
Description: Check down-tree potential root node by BFS
Input: *potentialRootNode*, *pathEdges*, *visited*
Output: *visited*, **return True iff down-tree**

```
1: $prevLevel$ new Hashtable
2: Add $potentialRootNode$ to $prevLevel$
3: $currentNode$ =null
4: while #nodes of $prevLevel > 0$ do
5:    $nextLevelVisited$ new Hashtable
6:    while #nodes of $prevLevel > 0$ do
7:       $currentNode \leftarrow$ a node $\in prevLevel$
8:       Remove $currentNode$ from $prevLevel$
9:       for all $outgoingEdge \in$ all outgoing edges of $currentNode$ do
10:         if $outgoingEdge$ is not visited then
11:            Set $outgoingEdge$ as visited
12:            $sourceNodeOfOutgoingEdge \leftarrow$ source node of $outgoingEdge$
13:            $targetNodeOfOutgoingEdge \leftarrow$ target node of $outgoingEdge$
14:            $outgoingEdgeId \leftarrow$ id of $outgoingEdge$
15:            if Call $isBackEdge(pathEdges, outgoingEdge)$ (Algorithm 14.38) then
16:               if id of $targetNodeOfOutgoingEdge \in visited$ then
17:                  Return false
18:               else
19:                  Add id of $targetNodeOfOutgoingEdge$ to $visited$
20:                  Add $targetNodeOfOutgoingEdge$ to $nextLevelVisited$
21:                  Add $outgoingEdge$ to $pathEdges$
22:               end if
23:            end if
24:         end if
25:      end for
26:   end while
27:   $prevLevel \leftarrow nextLevelVisited$
28: end while
29: Return true
```

Algorithm 14.37 *setAllEdgesAsNotVisited*
Description: Sets all edges of *graph* **as not visited**
Input: *graph*
Output: *graph* **with all edges marked as visited**

```
1: for all $node \in graph$ do
2:    $edges \leftarrow$ new Hashset containing incoming edges of $node$
3:    for all $edge \in edges$ do
4:       Set $edge$ as not visited
5:    end for
6: end for
```

Algorithm 14.38 *isBackEdge*

Description: Check if an edge is back edge

Input: *pathEdges*, *edge*

Output: true iff *edge* **is a back edge of another edge**

```
1: $testEdge \leftarrow$ reverse id of $edge$
2: $m \leftarrow$ an edge $\in pathEdges$ having id $testEdge$
3: if $m \neq$null then
4:     Return true
5: else
6:     Return false
7: end if
```

14.2 Drawing Algorithms

14.2.1 Algorithms of Computing Size of all Meta Nodes

From the layouts computed, their bounding box (tree, DAG) or bounding sphere (ntCS) is computed. Knowing the area size for each meta-node in level one, the drawings are stored to be used later for drawing the final layout. Then, the tree or the DAG composed of the meta-nodes in level one, which is represented by a meta-node in level two, is drawn using the area-aware versions of the tree or the DAG drawing algorithms, respectively, based on the area size of each meta-node in level one. Finally, the total area of the drawing is determined and the drawing is stored to be used for drawing the final layout.

The class diagram in Figure 14.10 gives an overview over the class structure for computing the size of all meta nodes. All global data attributes (variables) and data structures that are related to computing the area size are listed in this diagram. The "TwoLevelGraphDrawing-Panel" class contains the names of the algorithms (operations) used to compute the area size of all meta-nodes.

The global data attributes and data structures of TwoLevelGraphDrawingPanel class are:

- *startPosition*: an initial point that used to start drawing.

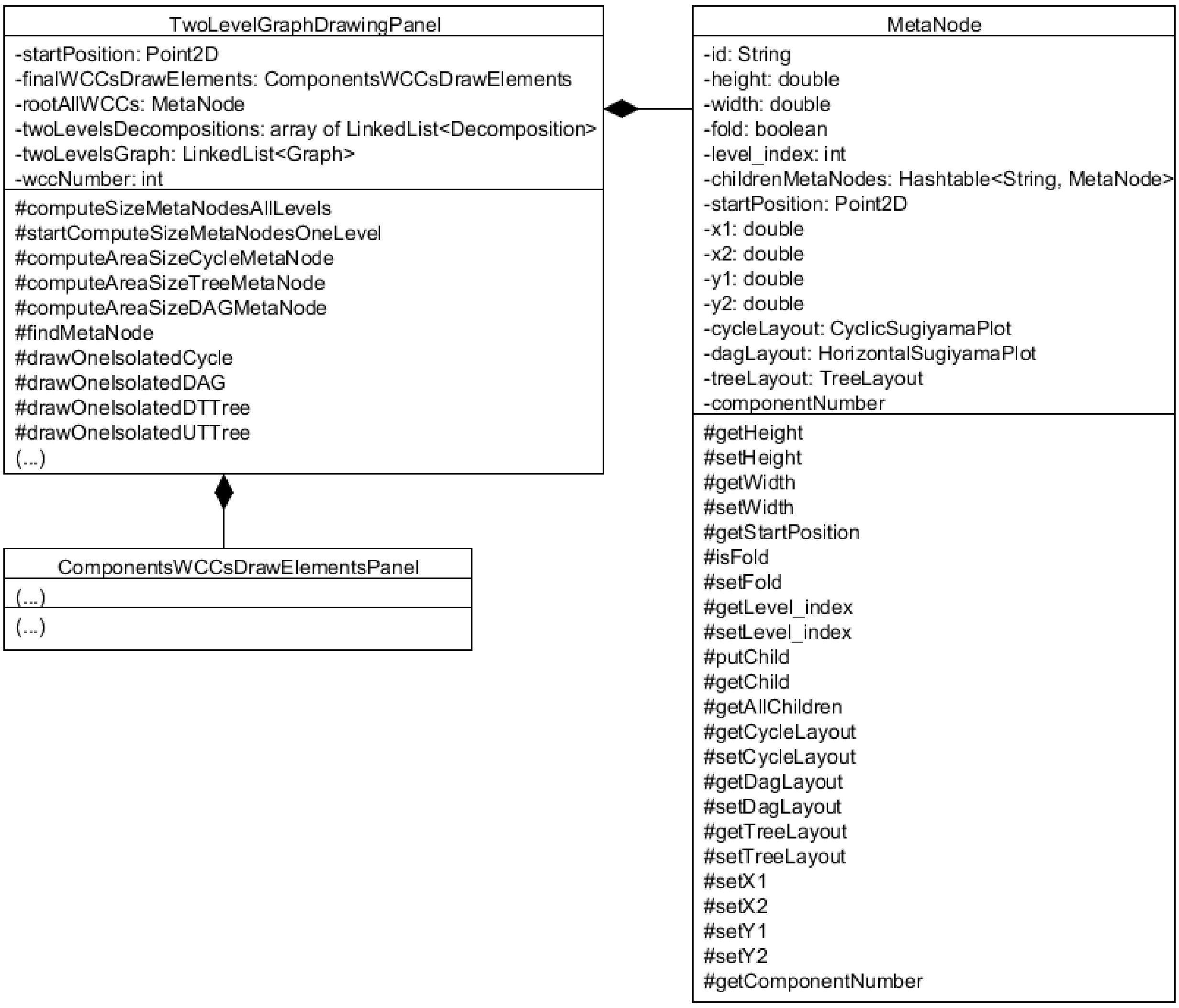


Figure 14.10: Class diagram for computing the size of all meta nodes

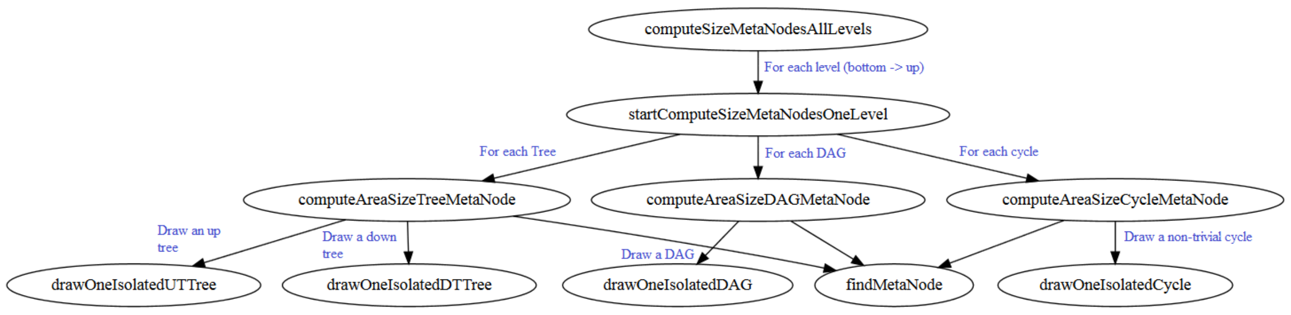


Figure 14.11: Call graph for the algorithms for computing meta-node sizes and for drawing subgraphs of the original graph provided by the TwoLevelGraphDrawingPanel class in Figure 14.10.

- *finalWCCsDrawElements*: an object from Class ComponentsWCCsDrawElementsPanel that contains final graphic elements.
- *rootAllWCCs*: a root meta-node (virtual meta-node) of all two-levels hierarchies
- *twoLevelsDecompositions*: an array of LinkedList (Deque, Table 14.3), where each list contains decompositions of one wCC (original graph) and its meta-graph.
- *twoLevelsGraph*: an array of LinkedList (Deque, Table 14.3), where each list contains one wCC (original graph) and its meta-graph.
- *wccNumber*: number of current wCC

A call graph of algorithms used for computing the area size of all meta-nodes is shown in Figure 14.11. The algorithm consists of several parts:

- *computeSizeMetaNodesAllLevels* (Algorithm 14.39)

  The local data attributes and data structures are:

  - *twoLevelsDecompositions*: an array of LinkedList (Deque, Table 14.3), where each list contains decompositions of one wCC (original graph) and its meta-graph.
  - *wccNumber*: number of current wCC
  - *twoLevelsGraph*: an array of LinkedList (Deque, Table 14.3), where each list contains one wCC (original graph) and its meta-graph.
  - *dummyRootHierarchy*: dummy root meta-node of two-levels hierarchy for current wCC

  The algorithm finds the size of meta-nodes in level one first, then it accumulates them in level two.

  - For each level, it calls *startComputeSizeMetaNodesOneLevel* (Algorithm 14.40) to compute area size of all meta-nodes (Lines 1-3).

- *startComputeSizeMetaNodesOneLevel* (Algorithm 14.40)

  The local data attributes and data structures are:

  - *decomposition*: a decomposition of current level
  - *nonEmptyGraph*: a graph of current level

- *level_index*: number of current level
- *dummyRootHierarchy*: a dummy root meta-node of two-level hierarchy
- *allCycles*: a hashtable (Table 14.2) containing all ntCS
- *allTrees*: a hashtable (Table 14.2) containing all trees
- *allDAGs*: a hashtable (Table 14.2) containing all DAGs
- *tree*: a Tree
- *dag*: a DAG

The algorithm computes the sizes of meta-nodes in a single level and drawing of the subgraphs of the original graph.

- For each meta-node, it calls Algorithm 14.41 for cycle meta-nodes (Line 5), Algorithm 14.46 for DAG meta-nodes (Line 8), and Algorithm 14.43 for tree meta-nodes (Line 11).

• *computeAreaSizeCycleMetaNode* (Algorithm 14.41)

The local data attributes and data structures are:

- *cycle*: a Cycle
- *nonEmptyGraph*: a graph of the current level
- *decomposition*: a decomposition of the current level
- *dummyRootHierarchy*: a dummy root meta-node of the two-level hierarchy
- *level_index*: number of the current level

Please note that *decomposition* and *nonEmptyGraph* provide information for internally use in the algorithm. The algorithm draws a subgraph representing a cycle and finds the radius of the smallest circle bounding it.

- Draw ntCS using Algorithm 14.42 (Line 3).
- Call Algorithm 14.48 to find the meta-node representing the ntCS in the two-level hierarchy (Line 4).
- If the meta-node is folded, use a fixed value for the size of the meta-node (Lines 5-6).
- Otherwise, get the radius of the cyclic drawing and use its radius as size of the meta-node including a border area (Lines 7-9).
- Store the size of the meta-node and the cyclic drawing in the meta-node (Lines 10-12).

• *computeAreaSizeTreeMetaNode* (Algorithm 14.43)

The local data attributes and data structures are:

- *tree*: a Tree
- *nonEmptyGraph*: a graph of the current level
- *decomposition*: a decomposition of the current level
- *dummyRootHierarchy*: a dummy root meta-node of the two-level hierarchy
- *level_index*: number of current level

Please note that *decomposition* and *nonEmptyGraph* provide information for internally use in the algorithm. The algorithm draws a subgraph representing a tree and finds the height and the width of the smallest rectangle surrounding it.

- Draw down-tree (up-tree) component using Algorithm 14.44 (Algorithm 14.45) (Lines 6-10).
- Call Algorithm 14.48 to find the meta-node representing the tree in the two-level hierarchy (Line 11).
- If the meta-node is folded, use a fixed value for the size of the meta-node (Lines 12-14).
- Otherwise, get the width and the height of the tree drawing and use them as size of the meta-node including a border area (Lines 15-20).
- Store the size of the meta-node and the tree drawing in the meta-node (Lines 21-23).

• *computeAreaSizeDAGMetaNode* (Algorithm 14.46)

The local data attributes and data structures are:

- *dag*: a DAG
- *nonEmptyGraph*: a graph of the current level
- *decomposition*: a decomposition of the current level
- *dummyRootHierarchy*: a dummy root meta-node of the two-level hierarchy
- *level_index*: number of current level

Please note that *decomposition* and *nonEmptyGraph* provide information for internally use in the algorithm. The algorithm draws a subgraph representing a DAG and finds the height and the width of the smallest rectangle surrounding it.

- Draw DAG component using Algorithm 14.47 (Line 6).
- Call Algorithm 14.48 to find the meta-node representing the DAG in the two-level hierarchy (Line 7).
- If the meta-node is folded, use a fixed value for the size of the meta-node (Lines 8-10).
- Otherwise, get the width and the height of the DAG drawing and use them as size of the meta-node including a border area (Lines 11-16).
- Store the size of the meta-node and the DAG drawing in the meta-node (Lines 17-19).

• *findMetaNode* (Algorithm 14.48)

The local data attributes and data structures are:

- *root*: a meta-node representing a subtree of the meta-node hierarchy
- *id*: the id of a meta-node

The algorithm searches for the meta-node having an id that equals the value of *id*.

- If *root* is the wanted meta-node, return it (Lines 1-2).
- Otherwise for each child of *root*, call *findMetaNode* (Algorithm 14.48) (Lines 3-7).

- *drawOneIsolatedCycle* (Algorithm 14.42)

  The local data attributes and data structures are:

  - *cycle*: a Cycle
  - *startPoint*: a start point for the drawing
  - *nonEmptyGraph*: a graph of the current level
  - *decomposition*: a decomposition of the current level
  - *color*: a color for the drawing

  The algorithm calls Bachmaier's algorithm [22] implemented in the Gravisto Toolkit [23] to draw *cycle*.

- *drawOneIsolatedDAG* (Algorithm 14.47)

  The local data attributes and data structures are:

  - *dag*: a DAG.
  - *startPoint*: a start point for the drawing
  - *nonEmptyGraph*: a graph of the current level
  - *decomposition*: a decomposition of the current level
  - *color*: a color for the drawing

  The algorithm uses the implementation of the Sugiyama algorithm [87] in the NetBeans Visual Library [79] to draw *dag*.

- *drawOneIsolatedDTTree* (Algorithm 14.44)

  The local data attributes and data structures are:

  - *tree*: a Down Tree
  - *startPoint*: a start point for the drawing
  - *nonEmptyGraph*: a graph of the current level
  - *decomposition*: a decomposition of the current level
  - *color*: a color for the drawing

  The algorithm uses the implementation of the improved Walker's algorithm [37] in the abego library TreeLayout [6] to draw down *tree*.

- *drawOneIsolatedUPTree* (Algorithm 14.45)

  The local data attributes and data structures are:

  - *tree*: an Up Tree
  - *startPoint*: a start point for the drawing
  - *nonEmptyGraph*: a graph of the current level
  - *decomposition*: a decomposition of the current level
  - *color*: a color for the drawing

  The algorithm uses the implementation of the improved Walker's algorithm [37] in abego TreeLayout [6] to draw up tree.

Algorithm 14.39 *computeSizeMetaNodesAllLevels*

Description: Compute the sizes of meta-nodes in all levels of the hierarchy
Input: *twoLevelDecompositions*, *wccNumber*, *twoLevelGraph*, *dummyRootHierarchy*
Output: All sizes of the meta-nodes in the hierarchy and the individual drawings of the subgraphs

```
1: for $level\_index \leftarrow 0$, $level\_index <$ size of $twoLevelDecompositions[wccNumber]$ do
2:     Call $startComputeSizeMetaNodesOneLevel(twoLevelDecompositions[wccNumber].get(level\_index), twoLevelGraph[wccNumber].get(level\_index), level\_index, dummyRootHierarchy)$ (Algorithm 14.40)
3: end for
```

Algorithm 14.40 *startComputeSizeMetaNodesOneLevel*

Description: Compute size of meta-nodes in a level and drawing of the subgraphs of the original graph
Input: *decomposition*, *nonEmptyGraph*, *level_index*, *dummyRootHierarchy*
Output: The sizes of the meta-nodes in a level

```
1: $allCycles \leftarrow$ get from $decomposition$ a hashtable of all cycles
2: $allTrees \leftarrow$ get from $decomposition$ a hashtable of all trees
3: $allDAGs \leftarrow$ get from $decomposition$ a hashtable of all DAGs
4: for all $cycle \in allCycles$ do
5:     Call $computeAreaSizeCycleMetaNode(cycle, nonEmptyGraph, decomposition, dummyRootHierarchy, level\_index)$ (Algorithm 14.41)
6: end for
7: for all $dag \in allDAGs$ do
8:     Call $computeAreaSizeDAGMetaNode(dag, nonEmptyGraph, decomposition, dummyRootHierarchy, level\_index)$ (Algorithm 14.46)
9: end for
10: for all $tree \in allTrees$ do
11:     Call $computeAreaSizeTreeMetaNode(tree, nonEmptyGraph, decomposition, dummyRootHierarchy, level\_index)$ (Algorithm 14.43)
12: end for
```

Algorithm 14.41 *computeAreaSizeCycleMetaNode*

Description: Compute a drawing of a cycle represented by a meta-node and the cycle's size

Input: *cycle*, *nonEmptyGraph*, *decomposition*, *dummyRootHierarchy*, *level_index*

Output: drawing of a cycle and the cycle's size

```
1: $startPoint \leftarrow$ a point with (0,0) coordinates
2: $color \leftarrow$ BLACK
3: $cyclicSugiyamaPlot \leftarrow$ call $drawOneIsolatedCycle$ ($cycle$, $startPoint$, $nonEmptyGraph$,
   $decomposition$, $color$) (Algorithm 14.42)
4: $metaNode \leftarrow$ call $findMetaNode$($dummyRootHierarchy$, id of $cycle$) (Algorithm 14.48)
   to find the meta-node representing the cycle in the hierarchy
5: if $metaNode$ is folded then
6:    $radiusBoundingCircle \leftarrow 5$
7: else
8:    $radiusBoundingCircle \leftarrow$ radius of $cyclicSugiyamaPlot$
9: end if
10: Store $radiusBoundingCircle * 2$ as height of $metaNode$ in hierarchy and the graph of
    $decomposition$
11: Store $radiusBoundingCircle * 2$ as width of $metaNode$ in hierarchy and the graph of
    $decomposition$
12: Store $cyclicSugiyamaPlot$ in $metaNode$
```

Algorithm 14.42 *drawOneIsolatedCycle*

Description: Draws a cycle using Bachmaier's algorithm [22]

Input: *cycle*, *startPoint*, *nonEmptyGraph*, *decomposition*, *color*

Output: a cyclic drawing

```
1: call Bachmaier's algorithm [22] implemented in the Gravisto Toolkit [23] to draw $cycle$
```

Algorithm 14.43 *computeAreaSizeTreeMetaNode*

Description: Compute a drawing of a tree represented by a meta-node and the tree's size.
Input: *tree*, *nonEmptyGraph*, *decomposition*, *dummyRootHierarchy*, *level_index*
Output: drawing of a tree and the tree's size

```
 1: startPoint ← a point with (0,0) coordinate
 2: color ← blue
 3: if level_index = 0 then
 4:     color ← black
 5: end if
 6: if tree is down-tree then
 7:     treeLayout ← call drawOneIsolatedDTTree (tree, startPoint, nonEmptyGraph,
        decomposition, color) (Algorithm 14.44)
 8: else if tree is up-tree then
 9:     treeLayout ← call drawOneIsolatedUPTree (tree, startPoint, nonEmptyGraph,
        decomposition, color) (Algorithm 14.45)
10: end if
11: metaNode ← call findMetaNode(dummyRootHierarchy, id of tree) (Algorithm 14.48)
    to find the meta-node representing the tree in the hierarchy
12: if metaNode is folded then
13:     heightBoundingTree ← 5
14:     widthBoundingTree ← 5
15: else
16:     heightBoundingTree ← height of treeLayout
17:     widthBoundingTree ← width of treeLayout
18:     Add 25 to heightBoundingTree
19:     Add 40 to widthBoundingTree
20: end if
21: Store heightBoundingTree as height of metaNode in hierarchy and the graph of
    decomposition
22: Store widthBoundingTree as width of metaNode in hierarchy and the graph of
    decomposition
23: Store treeLayout in metaNode
```

Algorithm 14.44 *drawOneIsolatedDTTree*

Description: Draw a down-tree using the improved Walker's algorithm [37]
Input: *tree*, *startPoint*, *nonEmptyGraph*, *decomposition*, *color*
Output: a Down-tree drawing

```
1: call the improved Walker's algorithm [37] implemented in the abego library TreeLayout [6]
   to draw tree
```

Algorithm 14.45 *drawOneIsolatedUPTree*

Description: Draw an up-tree using the improved Walker's algorithm [37]
Input: *tree*, *startPoint*, *nonEmptyGraph*, *decomposition*, *color*
Output: an Up-tree drawing

```
1: call the improved Walker's algorithm [37] implemented in the abego library TreeLayout [6]
   to draw tree
```

Algorithm 14.46 *computeAreaSizeDAGMetaNode*

Description: Compute a drawing of a DAG represented by a meta-node and the DAG's size
Input: *dag*, *nonEmptyGraph*, *decomposition*, *dummyRootHierarchy*, *level_index*
Output: a drawing of a DAG and the DAG's size

```
1: startPoint ← a point with (0,0) coordinate
2: color ← blue
3: if level_index = 0 then
4:    color ← black
5: end if
6: dagLayout ← call drawOneIsolatedDAG (dag, startPoint, nonEmptyGraph,
   decomposition, color) (Algorithm 14.47)
7: metaNode ← call findMetaNode(dummyRootHierarchy, id of dag) (Algorithm 14.48) to
   find the meta-node representing the DAG in the hierarchy
8: if metaNode is folded then
9:    heightBoundingDAG ← 5
10:   widthBoundingDAG ← 5
11: else
12:   heightBoundingDAG ← height of dagLayout
13:   widthBoundingDAG ← width of dagLayout
14:   Add 25 to heightBoundingDAG
15:   Add 40 to widthBoundingDAG
16: end if
17: Set heightBoundingDAG as height of metaNode in hierarchy and the graph of
    decomposition
18: Set widthBoundingDAG as width of metaNode in hierarchy and the graph of
    decomposition
19: Store dagLayout in metaNode
```

Algorithm 14.47 *drawOneIsolatedDAG*

Description: Draw a DAG using the Sugiyama algorithm [87]
Input: *dag*, *startPoint*, *nonEmptyGraph*, *decomposition*, *color*
Output: a DAG drawing

```
1: Call Sugiyama algorithm [87] implemented in the NetBeans Visual Library [79] to draw dag
```

Algorithm 14.48 *findMetaNode*

Description: Retrieve (Find) meta-node by its id
Input: *root*, *id*
Output: the (meta-)node having id *id*

```
1: if $id$ = id of $root$ then
2:     Return $root$
3: else
4:     for all $child \in$ all children of $root$ do
5:         Call $findMetaNode(child, id)$ (Algorithm 14.48)
6:     end for
7: end if
```

14.2.2 Algorithms of Final Drawing

The class diagram in Figure 14.12 shows an overview of the implementation part for the final drawing. All global data attributes (variables) and data structures that are related to drawing the final layout are listed in these diagrams. The call graph of the algorithms used for the final drawing is shown in Figure 14.13.
The Algorithm consists of several parts to perform the final drawing:

- *drawTwoLevelsAreaAwareHierarchy* (Algorithm 14.49)

  The local data attributes and data structures are:

  - *twoLevelsDecompositions*: a LinkedList (Deque, Table 14.3) containing decompositions of the original graph and the meta-graph
  - *wccNumber*: number of wCC
  - *startPoint*: a drawing start point
  - *twoLevelsGraph*: a LinkedList (Deque, Table 14.3) containing the original graph and the meta-graph
  - *tmpWCCsDrawElements*: an object from Class ComponentsWCCsDrawElements Panel that contains a temporary list of drawing elements for the final drawing

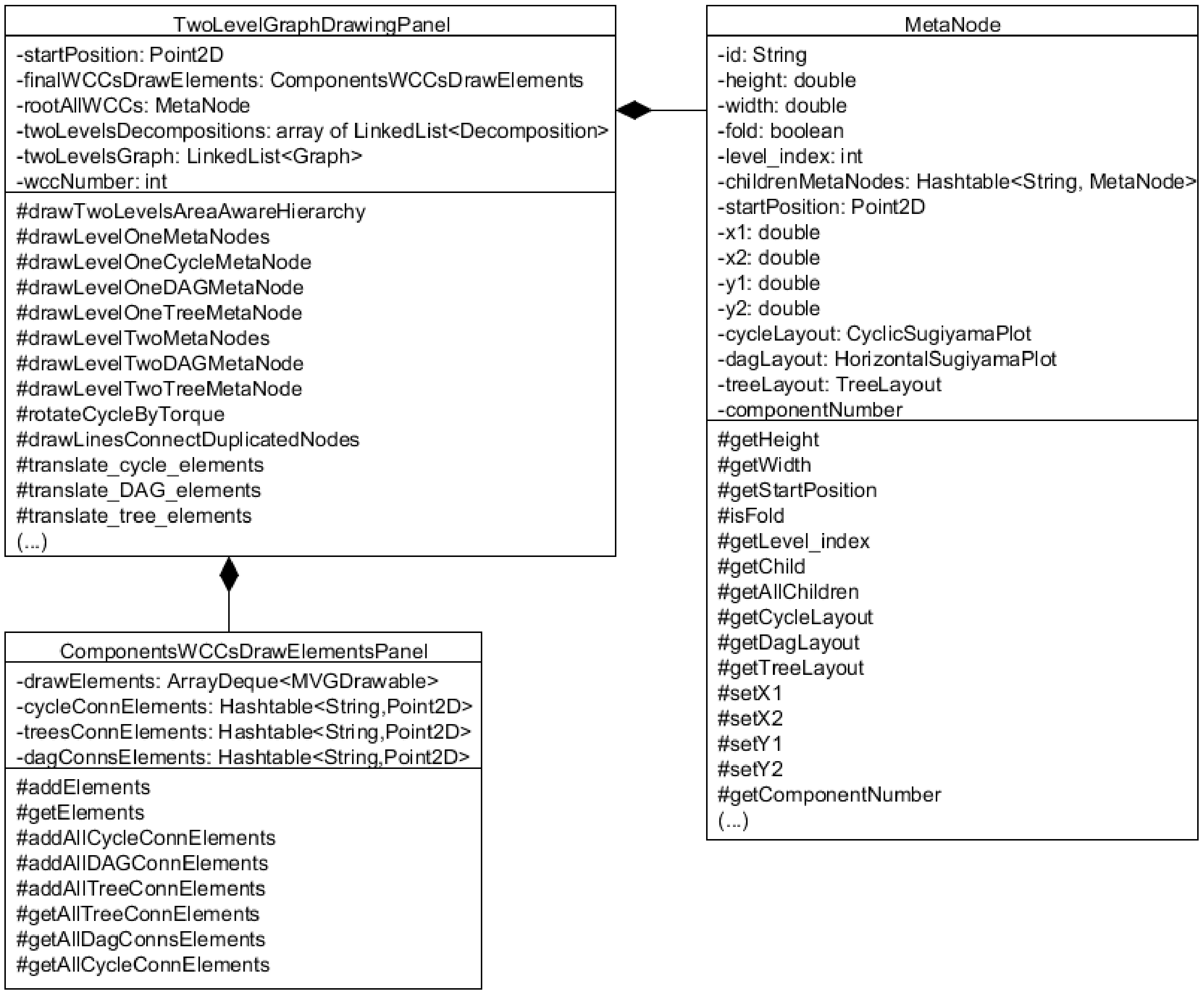


Figure 14.12: Class diagram for final drawing

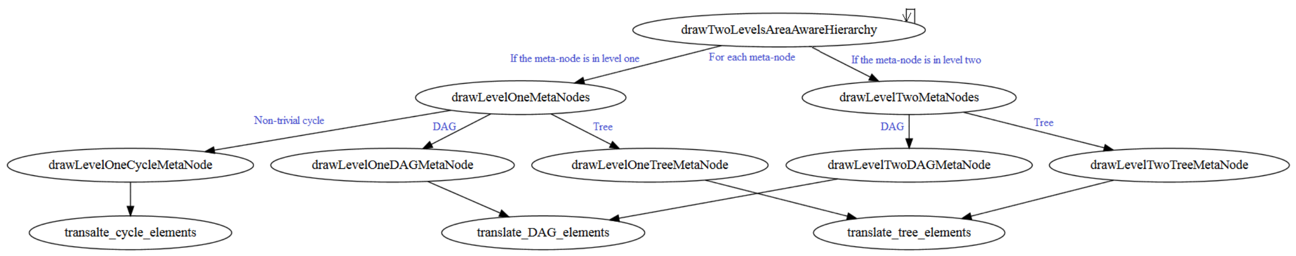


Figure 14.13: Call graph for final drawing algorithms provided by TwoLevelGraphDrawingPanel class in Figure 14.12.

- *parentMetaNode*: parent meta-node in two-levels hierarchy
- *showCycleLeavesLevel*: boolean variable to show cycle drawing elements iff true

Starting from the meta-node belonging to level two in the hierarchy, the algorithm places the area-aware layout representing the meta-graph horizontally. Then, it translates each layout representing a subgraph of the original graph associated with the child meta-node in level one to its dedicated area in the area-aware layout.

Let us assume, that the hierarchy has additionally a virtual root with a predetermined start position. Moreover, the virtual root has virtual children representing one WCC each considered as a parent of the meta-node in level two. If there are more than one WCC, the width of the current area-aware layout will be added to the start position of the next area-aware layout.

- For each child meta-node of parent meta-node.
  - If the meta-node is in level one, it calls *drawLevelOneMetaNodes* (Algorithm 14.50) (Lines 4-5).
  - If the meta-node is in level two, it calls *drawLevelTwoMetaNodes* (Algorithm 14.54) and calls *drawTwoLevelsAreaAwareHierarchy* (Algorithm 14.49), recursively (Lines 7-10).

- *drawLevelOneMetaNodes* (Algorithm 14.50)

  The local data attributes and data structures are:

  - *key*: id of child meta-node
  - *childMetaNode*: child meta-node
  - *showCycleLeavesLevel*: boolean variable to show cycle drawing elements iff true
  - *startPoint*: a drawing start point
  - *tmpWCCsDrawElements*: an object from Class ComponentsWCCsDrawElements Panel that contains a temporary list of drawing elements (arrows, curves, and points) for the final drawing
  - *currentIndex*: level number of child meta-node

  The algorithm works as follows:

  - If *key* represents the id of cycle, it calls Algorithm 14.51 (Lines 1-2).
  - If *key* represents the id of DAG, it calls Algorithm 14.52 (Lines 3-4).
  - If *key* represents the id of Tree, it calls Algorithm 14.53 (Lines 5-7).

- *drawLevelOneCycleMetaNode* (Algorithm 14.51)

  The local data attributes and data structures are:

  - *childMetaNode*: child meta-node
  - *showCycleLeavesLevel*: boolean variable to show cycle drawing elements iff true
  - *startPoint*: a drawing start point
  - *tmpWCCsDrawElements*: an object from Class ComponentsWCCsDrawElements Panel that contains a temporary list of drawing elements for the final drawing
  - *currentIndex*: level number of child meta-node

  The algorithm works as follows:

  - It translates its associated cyclic layout to the dedicated area inside the area-aware layout of the parent meta-node (Line 7)
  - and adds its drawing elements to the final drawing (Lines 8-13).

- *drawLevelOneDAGMetaNode* (Algorithm 14.52)

  The local data attributes and data structures are:

  - *childMetaNode*: child meta-node
  - *showCycleLeavesLevel*: boolean variable to show cycle drawing elements iff true
  - *startPoint*: a drawing start point
  - *tmpWCCsDrawElements*: an object from Class ComponentsWCCsDrawElements Panel that contains a temporary list of drawing elements for the final drawing
  - *currentIndex*: level number of child meta-node

  The algorithm works as follows:

  - It translates its associated DAG layout to the dedicated area inside the area-aware layout of the parent meta-node (Line 7)
  - and adds its drawing elements to the final drawing (Lines 8-10).

- *drawLevelOneTreeMetaNode* (Algorithm 14.53)

  The local data attributes and data structures are:

  - *childMetaNode*: child meta-node
  - *showCycleLeavesLevel*: boolean variable to show cycle drawing elements iff true
  - *startPoint*: a drawing start point
  - *tmpWCCsDrawElements*: an object from Class ComponentsWCCsDrawElements that contains a temporary list of drawing elements for the final drawing
  - *currentIndex*: level number of child meta-node

  The algorithm works as follows:

  - It translates its associated tree layout to the dedicated area inside the area-aware layout of the parent meta-node (Line 7)
  - and adds its drawing elements to the final drawing (Lines 8-10).

- *drawLevelTwoMetaNodes* (Algorithm 14.54)

  The local data attributes and data structures are:

  - *currentIndex*: level number of child meta-node
  - *key*: id of child meta-node
  - *childMetaNode*: child meta-node
  - *startPoint*: a drawing start point
  - *tmpWCCsDrawElements*: an object from Class ComponentsWCCsDrawElements Panel that contains a temporary list of drawing elements for the final drawing
  - *twoLevelsDecompositions*: a LinkedList (Deque, Table 14.3) containing decompositions of the original graph and the meta-graph
  - *wccNumber*: number of wCC
  - *twoLevelsGraph*: a LinkedList (Deque, Table 14.3) containing the original graph and the meta-graph
  - *showCycleLeavesLevel*: boolean variable to show cycle drawing elements iff true

  The algorithm works as follows:

  - If *key* represents the id of DAG, it calls *drawLevelTwoDAGMetaNode* (Algorithm 14.55) (Lines 1-2).
  - If *key* represents the id of tree, it calls *drawLevelTwoTreeMetaNode* (Algorithm 14.56) (Lines 3-5).

- *drawLevelTwoDAGMetaNode* (Algorithm 14.55)

  The local data attributes and data structures are:

  - *childMetaNode*: child meta-node
  - *startPoint*: a drawing start point
  - *currentIndex*: level number of the child meta-node
  - *tmpWCCsDrawElements*: an object from Class ComponentsWCCsDrawElements Panel that contains a temporary list of drawing elements for the final drawing

  The algorithm works as follows:

  - It translates its associated DAG area-aware layout to the predetermined total area (Line 7),
  - then it associates the position for each node in the layout with the respective meta-node in level one (Lines 8-11)
  - and adds the drawing elements of the area-aware layout to the final drawing (Line 12).

- *drawLevelTwoTreeMetaNode* (Algorithm 14.56)

  The local data attributes and data structures are:

  - *childMetaNode*: child meta-node
  - *startPoint*: a drawing start point
  - *currentIndex*: level number of child meta-node

– *tmpWCCsDrawElements*: an object from Class ComponentsWCCsDrawElements Panel that contains a temporary list of drawing elements for the final drawing

The algorithm works as follows:

– It translates its associated tree area-aware layout to the predetermined total area (Line 7),

– then it associates the position for each node in the layout with the respective meta-node in level one (Lines 8-11)

– and adds the drawing elements of the area-aware layout to the final drawing (Line 12).

Algorithm 14.49 *drawTwoLevelsAreaAwareHierarchy*

Description: Draw two-levels area-aware hierarchy
Input: *twoLevelsDecompositions*, *wccNumber*, *startPoint*, *twoLevelsGraph*, *tmpWCCsDrawElements*, *parentMetaNode*, *showCycleLeavesLevel*
Output: drawing for a meta-graph that contains edges

```
1: for all childMetaNode ∈ all children of parentMetaNode do
2:     key = id of childMetaNode
3:     currentIndex ← level of childMetaNode
4:     if currentIndex = 0 then
5:         Call drawLevelOneMetaNodes (key, childMetaNode, showCycleLeavesLevel,
6: startPoint, tmpWCCsDrawElements, currentIndex)
7:     else
8:         Call drawLevelTwoMetaNodes (currentIndex, key, childMetaNode, startPoint,
   tmpWCCsDrawElements, twoLevelsDecompositions, wccNumber, twoLevelsGraph,
   showCycleLeavesLevel)
9:         Call drawTwoLevelsAreaAwareHierarchy (twoLevelsDecompositions,
   wccNumber, startPoint, twoLevelsGraph, tmpWCCsDrawElements, childMetaNode,
   showCycleLeavesLevel)
10:    end if
11: end for
```

Algorithm 14.50 *drawLevelOneMetaNodes*

Description: Draw level one meta-nodes

Input: *key*, *childMetaNode*, *showCycleLeavesLevel*, *startPoint*, *tmpWCCsDrawElements*, *currentIndex*

Output: drawing for a meta-graph that contains no edges

```
1: if key is an id of cycle then
2:   Call drawLevelOneCycleMetaNode (childMetaNode, showCycleLeavesLevel,
     startPoint, tmpWCCsDrawElements, currentIndex)
3: else if key is an id of DAG then
4:   Call drawLevelOneDAGMetaNode (childMetaNode, showCycleLeavesLevel,
     startPoint, tmpWCCsDrawElements, currentIndex)
5: else if key is an id of tree then
6:   Call drawLevelOneTreeMetaNode (childMetaNode, showCycleLeavesLevel,
     startPoint, tmpWCCsDrawElements, currentIndex)
7: end if
```

Algorithm 14.51 *drawLevelOneCycleMetaNode*

Description: Draw level one cycle meta-node

Input: *childMetaNode*, *showCycleLeavesLevel*, *startPoint*, *tmpWCCsDrawElements*, *currentIndex*

Output: drawing of cyclic sub-graph associated with meta-node

```
1: cyclicSugiyamaPlot $\leftarrow$ layout of cycle
2: if showCycleLeavesLevel then
3:   coordinates of metaPoint $\leftarrow$ summation of childMetaNode and startPoint coordinates
     increased by half of width and height childMetaNode
4: else
5:   coordinates of metaPoint $\leftarrow$ coordinates of childMetaNode increased by half of width
     and height childMetaNode
6: end if
7: translate drawing elements of cyclicSugiyamaPlot toward metaPoint
8: if showCycleLeavesLevel $\wedge$ childMetaNode is not folded then
9:   Store drawing elements of childMetaNode in tmpWCCsDrawElements
10:  Store final coordinates of upper left point and bottom right point in childMetaNode
     adding startPoint coordinates to them
11: else
12:  Store final coordinates of upper left point and bottom right point in childMetaNode
13: end if
```

Algorithm 14.52 *drawLevelOneDAGMetaNode*

Description: Draw level one DAG meta-node

Input: *childMetaNode*, *showCycleLeavesLevel*, *startPoint*, *tmpWCCsDrawElements*, *currentIndex*

Output: drawing DAG sub-graph associated with meta-node

```
 1: dagLayout ← layout of DAG
 2: if showCycleLeavesLevel then
 3:    coordinates of metaPoint ← summation of childMetaNode and startPoint coordinates
       increased by half of width and height childMetaNode
 4: else
 5:    coordinates of metaPoint ← coordinates of childMetaNode increased by half of width
       and height childMetaNode
 6: end if
 7: translate drawing elements of dagLayout toward metaPoint
 8: if childMetaNode is not folded then
 9:    Store drawing elements of childMetaNode in tmpWCCsDrawElements
10: end if
11: if showCycleLeavesLevel $\wedge$ childMetaNode is not folded then
12:    Store final coordinates of upper left point and bottom right point in childMetaNode
       adding startPoint coordinates to them
13: else
14:    Store final coordinates of upper left point and bottom right point in childMetaNode
15: end if
```

Algorithm 14.53 *drawLevelOneTreeMetaNode*

Description: Draw level one tree meta-node

Input: *childMetaNode*, *showCycleLeavesLevel*, *startPoint*, *tmpWCCsDrawElements*, *currentIndex*

Output: drawing tree sub-graph associated with meta-node

```
 1: treeLayout ← layout of tree
 2: if showCycleLeavesLevel then
 3:    coordinates of metaPoint ← summation of childMetaNode and startPoint coordinates
       increased by half of width and height childMetaNode
 4: else
 5:    coordinates of metaPoint ← coordinates of childMetaNode increased by half of width
       and height childMetaNode
 6: end if
 7: translate drawing elements of treeLayout towards metaPoint
 8: if childMetaNode is not folded then
 9:    Store drawing elements of childMetaNode in tmpWCCsDrawElements
10: end if
11: if showCycleLeavesLevel $\wedge$ childMetaNode is not folded then
12:    Store final coordinates of upper left point and bottom right in childMetaNode adding
       startPoint coordinates to them
13: else
14:    Store final coordinates of upper left point and bottom right in childMetaNode
15: end if
```

Algorithm 14.54 *drawLevelTwoMetaNodes*

Description: Draw level two meta-nodes

Input: *currentIndex*, *key*, *childMetaNode*, *startPoint*, *tmpWCCsDrawElements*, *twoLevelsDecompositions*, *wccNumber*, *twoLevelsGraph*, *showCycleLeavesLevel*

Output: drawing a meta-graph that contains edges

```
1: if key is an id of DAG then
2:     Call drawLevelTwoDAGMetaNode (childMetaNode, startPoint, currentIndex, tmpWCCsDrawElements)
3: else if key is an id of tree then
4:     Call drawLevelTwoTreeMetaNode (childMetaNode, startPoint, currentIndex, tmpWCCsDrawElements)
5: end if
```

Algorithm 14.55 *drawLevelTwoDAGMetaNode*

Description: Draw level two DAG meta-node

Input: *childMetaNode*, *startPoint*, *currentIndex*, *tmpWCCsDrawElements*

Output: drawing a DAG meta-graph

```
1: dagLayout ← area-aware layout of DAG
2: x coordinate of metaPoint ← x coordinates summation of childMetaNode and startPoint increased by half of width childMetaNode
3: y coordinate of metaPoint: analogue to x
4: distance ← a distance between metaPoint and center of dagLayout
5: dirx ← difference between x coordinates of metaPoint and center of dagLayout divided by distance
6: diry ← difference between y coordinates of metaPoint and center of dagLayout divided by distance
7: translate drawing elements of dagLayout in dirx and diry directions and distance value
8: for all nodeCurrentLevel ∈ dagLayout do
9:     metaNodeLevelOne ← get child of childMetaNode having id of nodeCurrentLevel
10:    The position of metaNodeLevelOne ← the position of nodeCurrentLevel in dagLayout
11: end for
12: Store drawing elements of childMetaNode in tmpWCCsDrawElements
13: Store final x and y coordinates of upper left point and bottom right point in childMetaNode adding startPoint coordinates to them
```

Algorithm 14.56 *drawLevelTwoTreeMetaNode*

Description: Draw level two tree meta-node

Input: *childMetaNode*, *startPoint*, *currentIndex*, *tmpWCCsDrawElements*

Output: drawing a tree meta-graph

```
1: $treeLayout$ ← area-aware layout of tree
2: $x$ coordinate of $metaPoint$ ← $x$ coordinates summation of $childMetaNode$ and $startPoint$
   increased by half of width $childMetaNode$
3: $y$ coordinate of $metaPoint$: analogue to $x$
4: $distance$ ← a distance between $metaPoint$ and center of $treeLayout$
5: $dirx$ ← difference between $x$ coordinates of $metaPoint$ and center of $treeLayout$ divided
   by $distance$
6: $diry$ ← difference between $y$ coordinates of $metaPoint$ and center of $treeLayout$ divided
   by $distance$
7: translate drawing elements of $treeLayout$ in $dirx$ and $diry$ directions and $distance$ value
8: for all $nodeCurrentLevel \in treeLayout$ do
9:     $metaNodeLevelOne$ ← gets child of $childMetaNode$ having id of $nodeCurrentLevel$
10:    The position of $metaNodeLevelOne$ ← the position of $nodeCurrentLevel$ in $treeLayout$
11: end for
12: Store drawing elements of $childMetaNode$ in $tmpWCCsDrawElements$
13: Store final $x$ and $y$ coordinates of upper left point and bottom right point in $childMetaNode$
    adding $startPoint$ coordinates to them
```

14.2.2.1 Rotating the cyclic subgraphs

For rotating ntCSs and minimizing the length of cyan lines connecting shared nodes, the final graph layout is enhanced as in Algorithm 14.57. The algorithm rotates the cyclic subgraphs similar to the approach of Archambault et al. [17] reducing the overall distance between duplicated nodes. The Algorithm consists of one part to perform the rotation of ntCSs:

- *rotateCycleByTorque* (Algorithm 14.57)

The local data attributes and data structures are:

- *twoLevelsDecompositions*: a LinkedList (Deque, Table 14.3) containing decompositions of the original graph and the meta-graph
- *wccNumber*: number of wCC
- *startPoint*: a drawing start point
- *twoLevelsGraph*: a LinkedList (Deque, Table 14.3) containing the original graph and the meta-graph
- *tmpWCCsDrawElements*: an object of the Class ComponentsWCCsDrawElementsPanel that contains a temporary list of drawing elements for the final drawing
- *parentMetaNode*: parent meta-node in two-levels hierarchy

Algorithm 14.57 *rotateCycleByTorque*

Description: Rotate cyclic subgraphs similar to the approach of Archambault et al. [17]
Input: *twoLevelsDecompositions*, *wccNumber*, *startPoint*, *twoLevelsGraph*, *tmpWCCsDrawElements*, *parentMetaNode*, *color*
Output: *tmpWCCsDrawElements* **with rotated cyclic drawing**

```
1: The algorithm rotates the cyclic subgraphs similar to the approach of Archambault et
   al. [17] reducing the overall distance between duplicated nodes.
```

14.2.2.2 Drawing lines between duplicated nodes

Having nodes that belong to more than one topological component, each topological component has its own copy of such a node. The copies are linked by a cyan line between two topological components. Algorithm 14.58 draws lines between duplicated nodes in two topological components by traversing the hierarchy. These duplicated nodes of each topological component were found previously in the decomposition process. The algorithm draws lines between duplicated nodes in two topological components by traversing the hierarchy. The Algorithm consists of one part to perform the lines drawing of duplicated nodes:

- *drawLinesConnectDuplicatedNodes* (Algorithm 14.58)

The local data attributes and data structures are:

- *twoLevelsDecompositions*: a LinkedList (Deque, Table 14.3) containing decompositions of the original graph and of the meta-graph

- *wccNumber*: number of current wCC
- *startPoint*: a drawing start point
- *twoLevelsGraph*: a LinkedList (Deque, Table 14.3) containing the original graph and the meta-graph
- *tmpWCCsDrawElements*: an object of the Class ComponentsWCCsDrawElements that contains a temporary list of drawing elements for the final drawing
- *parentMetaNode*: parent meta-node in two-levels hierarchy
- *color*: an intended color of all lines
- *key*: id of a child meta-node
- *currentIndex*: level of a child meta-node
- *connLineElements*: an empty ArrayDeque (Table 14.5) to store cyan lines

The algorithm works as follows:

- For each child meta-node of parent meta-node.
  - If the meta-node is in level one and is not folded, based on the type of the subgraph associated with the meta-node, go over each shared node that is duplicated in the other subgraph and draw a line connecting them (Lines 4-22).
  - Otherwise, it calls *drawLinesConnectDuplicatedNodes* Algorithm recursively over all child meta-nodes (Line 24).

Algorithm 14.58 *drawLinesConnectDuplicatedNodes*

Description: Draw cyan lines between duplicated nodes in components
Input: *twoLevelsDecompositions*, *wccNumber*, *startPoint*, *twoLevelsGraph*, *tmpWCCsDrawElements*, *parentMetaNode*, *color*
Output: *tmpWCCsDrawElements* **with cyan lines between duplicated nodes**

```
1: for all $childMetaNode \in$ all children of $parentMetaNode$ do
2:    $key \leftarrow$ id of $childMetaNode$
3:    $currentIndex \leftarrow$ level of $childMetaNode$
4:    if $currentIndex = 0 \wedge childMetaNode$ is not folded then
5:       if $key$ is an id of cycle then
6:          $cyclicSugiyamaPlot \leftarrow$ layout of cycle associated with $childMetaNode$
7:          $connLineElements$ an empty ArrayDeque to store cycan lines
8:          for all $cycleConnPointID \in$ all shared nodes ids of $cyclicSugiyamaPlot$ with other components do
9:             $treeConnPoint \leftarrow$ get a point from a hashtable in $tmpWCCsDrawElements$ with $cycleConnPointID$ as id {a node of a tree layout shared by the tree and $cyclicSugiyamaPlot$}
10:            if $treeConnPoint$ is not null then
11:               $cycleConnPoint \leftarrow$ get a point from a HashMap in $cyclicSugiyamaPlot$ with $cycleConnPointID$ as id
12:               $line \leftarrow$ create a cyan line from $cycleConnPoint$ to $treeConnPoint$
13:               Add $line$ to $connLineElements$
14:            end if
15:            $dagConnPoint \leftarrow$ get a point from a hashtable in $tmpWCCsDrawElements$ with $cycleConnPointID$ as id {a node of a DAG layout shared by the DAG and $cyclicSugiyamaPlot$)}
16:            if $dagConnPoint$ is not null then
17:               $cycleConnPoint \leftarrow$ get a point from a HashMap in $cyclicSugiyamaPlot$ with $cycleConnPointID$ as id
18:               $line \leftarrow$ create a cyan line from $cycleConnPoint$ to $dagConnPoint$
19:               Add $line$ to $connLineElements$
20:            end if
21:         end for
22:         Add $connLineElements$ to $tmpWCCsDrawElements$
23:      else
24:         Call $drawLinesConnectDuplicatedNodes$ ($twoLevelsDecompositions$,$wccNumber$, $startPoint$,$twoLevelsGraph$,$tmpWCCsDrawElements$,$childMetaNode$,$color$)
25:      end if
26:   end if
27: end for
```

14.3 Data Structures and Their Time Complexity

The algorithms in Chapter 14 use different data structures provided by Java programing language. Tables 14.1, 14.2, 14.3, 14.4, and 14.5 show these data structures and their time complexity.

Operation	Time Complexity
add	amortized constant time
remove	$O(n)$
clear	$O(n)$
contains	$O(n)$

Table 14.1: The time complexity of the ArrayList operations

Operation	Time Complexity
put	$O(1)$
get	$O(1)$
remove	$O(1)$
clear	$O(n)$

Table 14.2: The time complexity of the Hashtable and HashMap operations

Operation	Time Complexity
Insert/delete at beginning/end	$O(1)$
Insert/delete in middle	$O(n)$
Searching	$O(n)$

Table 14.3: The time complexity of the LinkedList operations (an implementation of list and Deque)

Operation	Time Complexity
add	$O(1)$
remove	$O(1)$
contains	$O(1)$
containsAll	$O(n)$
clear	$O(n)$

Table 14.4: The time complexity of the HashSet and LinkedHashSet operations

Most operations	amortized constant time
Bulk operations	$O(n)$
Remove	$O(n)$
RemoveFirstOccurrence	$O(n)$
RemoveLastOccurrence	$O(n)$
Contains	$O(n)$

Table 14.5: The time complexity of the ArrayDeque operations (Deque and Array implementation)

14.4 Complexity Analysis of Decomposition and Hierarchy Construction

14.4.1 Complexity Analysis of Cycle Detection Algorithms

The time complexity of the *FindingAllCycle* (Algorithm 14.1) is $O(c^3 \cdot n^2 \cdot (n+e))$ and the space complexity is $O(c^3 \cdot n^2 \cdot (n+e))$ (Table 14.6).

Line	Time complexity	Space complexity	Comments
1	$O(e)$		ArrayList iterator
4	$O(n)$		ArrayList iterator
7	$O(n)$	$O(n)$	ArrayList iterator
9	$O(c^3 \cdot n \cdot (n+e))$	$O(c^3 \cdot n \cdot (n+e))$	Algorithm 14.2
7-11	$\boldsymbol{O(c^3 \cdot n^2 \cdot (n+e))}$	$\boldsymbol{O(c^3 \cdot n^2 \cdot (n+e))}$	ArrayList iterator
Overall	$O(c^3 \cdot n^2 \cdot (n+e))$	$O(c^3 \cdot n^2 \cdot (n+e))$	

Table 14.6: Complexity analysis of *FindingAllCycle* (Algorithm 14.1)

The time complexity of the *findCyclesStart* (Algorithm 14.2) is $O(c^3 \cdot n \cdot (n+e))$ and the space complexity is $O(c^3 \cdot n \cdot (n+e))$ (Table 14.7).

Line	Time complexity	Space complexity	Comments
1	$O(1)$	$O(1)$	ArrayList create
2	$O(1)$	$O(1)$	ArrayList create
3	$O(1)$	$O(1)$	Hashtable create
4	$O(1)$	$O(1)$	ArrayList add
6	$O(c^3 \cdot n \cdot (n+e))$	$O(c^3 \cdot n \cdot (n+e))$	Algorithm 14.3
7	$O(n)$		ArrayList clear
8	$O(n)$		Hashtable clear
Overall	$O(c^3 \cdot n \cdot (n+e))$	$O(c^3 \cdot n \cdot (n+e))$	

Table 14.7: Complexity analysis of *findCyclesStart* (Algorithm 14.2)

The time complexity of the *find_cycle* (Algorithm 14.3) is $O(c^3 \cdot n \cdot (n+e))$ and the space complexity is $O(c^3 \cdot n \cdot (n+e))$ (Table 14.8).

Line	Time complexity	Space complexity	Comments
3	$O(c^3 \cdot n \cdot (n+e))$	$O(c^3 \cdot n \cdot (n+e))$	Algorithm 14.5
6	$O(1)$		Hashtable get
7	$O(e)$	$O(e)$	Hashtable iterator
9	$O(1)$		Algorithm 14.6
10	$O(1)$	$O(1)$	HashSet create
11	$O(1)$	$O(1)$	HashSet add
12	$O(1)$	$O(1)$	Hashtable add
15	$O(1)$		Algorithm 14.4
7-18	$\boldsymbol{O(e)}$	$\boldsymbol{O(e)}$	Hashtable iterator
21	$O(e)$		Hashtable iterator
25	$O(1)$		Algorithm 14.4
21-27	$\boldsymbol{O(e)}$		Hashtable iterator
Overall	$O(c^3 \cdot n \cdot (n+e))$	$O(c^3 \cdot n \cdot (n+e))$	

Table 14.8: Complexity analysis of *find_cycle* (Algorithm 14.3)

The time complexity of the *recursive_find_cycle_call* is $O(e)$ and the space complexity is $O(1)$ (Table 14.9).

Line	Time complexity	Space complexity	Comments
2	$O(1)$	$O(1)$	ArrayList add
3	$O(1)$	$O(1)$	Hashtable add
4	$O(1)$		Algorithm 14.3
5	$O(1)$		ArrayList remove
6	$O(1)$		Hashtable remove
Overall	$O(1)$	$O(1)$	

Table 14.9: Complexity analysis of *recursive_find_cycle_call* (Algorithm 14.4)

The time complexity of the *Check_Cycles* is $O(c^3 \cdot n \cdot (n+e))$ and the space complexity is $O(c^3 \cdot n \cdot (n+e))$ (Table 14.10).

Line	Time complexity	Space complexity	Comments
5	$O(1)$		Hashmap get
6	$O(n)$		ArrayList iterate
15	$O(1)$		Hashmap get
19	$O(n)$	$O(n)$	Algorithm 14.7 get
22	$O(n+e)$	$O(n+e)$	HashSet add nodes and edges of one cycle
23	$O(n)$	$O(1)$	Algorithm 14.8 get
24	$O(c^3 \cdot (n+e))$	$O(c^3 \cdot (n+e))$	Algorithm 14.11 get
28	$O(c^3 \cdot n \cdot (n+e))$	$O(c^3 \cdot n \cdot (n+e))$	Algorithm 14.10 get
Overall	$O(c^3 \cdot n \cdot (n+e))$	$O(c^3 \cdot n \cdot (n+e))$	

Table 14.10: Complexity analysis of *Check_Cycles* (Algorithm 14.5)

The time complexity of the *isBackEdge* is $O(1)$ (Table 14.11).

Line	Time complexity	Space complexity	Comments
2	$O(1)$		Hashtable get
Overall	$O(1)$		

Table 14.11: Complexity analysis of *isBackEdge* (Algorithm 14.6)

The time complexity of the *computeReducedPathComplexity* is $O(n)$ and the space complexity is $O(n)$ (Table 14.12).

Line	Time complexity	Space complexity	Comments
2	$O(1)$	$O(1)$	ArrayList create
3	$O(n)$	$O(n)$	ArrayList iterate
4	$O(1)$	$O(1)$	ArrayList add
3-5	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	ArrayList iterate, add
6	$O(1)$	$O(1)$	LinkedHashSet create
10	$O(n)$	$O(n)$	ArrayList iterate
11	$O(1)$		ArrayList get using index
12	$O(1)$	$O(1)$	ArrayList add
13	$O(1)$		Hashtable get
10-20	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	ArrayList iterate, get, add
22	$O(n)$	$O(n)$	ArrayList create initialized by LinkedHashSet
Overall	$O(n)$	$O(n)$	

Table 14.12: Complexity analysis of *compute_ReducedPath* (Algorithm 14.7)

The time complexity of the *subCycle* (Algorithm 14.8) is $O(n)$ and the space complexity is $O(1)$ (Table 14.13).

Line	Time complexity	Space complexity	Comments
1	$O(c)$		ArrayList iterate
2	$O(\frac{n}{c})$		HashSet containAll
1-5	$\boldsymbol{O(c \cdot \frac{n}{c})}$		iterate, containAll
6	$O(1)$	$O(1)$	ArrayList add
Overall	$O(c \cdot \frac{n}{c}) = O(n)$	$O(1)$	

Table 14.13: Complexity analysis of *subCycle* (Algorithm 14.8)

The time complexity of the *addToCycle* (Algorithm 14.9) is $O(n)$ and the space complexity is $O(n)$ (Table 14.14).

Line	Time complexity	Space complexity	Comments
1	$O(n)$	$O(n)$	ArrayList iterate
2	$O(1)$		ArrayList get using index
3	$O(1)$		Hashset contains
4	$O(1)$	$O(1)$	Hashset add
5	$O(1)$		Hashmap add
1-5	$\boldsymbol{O(n)}$	$\boldsymbol{O(n)}$	ArrayList iterate, get, contains, add, add
Overall	$O(n)$	$O(n)$	

Table 14.14: Complexity analysis of *addToCycle* (Algorithm 14.9)

The time complexity of the *partialCycle* (Algorithm 14.10) is $O(c^3 \cdot n \cdot (n + e))$ and the space complexity is $O(c^3 \cdot n \cdot (n + e))$ (Table 14.15).

Line	Time complexity	Space complexity	Comments
1	$O(1)$		Hashmap get
3	$O(n)$	$O(n)$	ArrayList iterate
4	$O(1)$		ArrayList get using index
6	$O(1)$		Hashmap get
8	$O(n)$	$O(n)$	Algorithm 14.9
9	$O(c^3 \cdot (n + e))$	$O(c^3 \cdot (n + e))$	Algorithm 14.11
3-13	$\boldsymbol{O(c^3 \cdot n \cdot (n + e))}$	$\boldsymbol{O(c^3 \cdot n \cdot (n + e))}$	ArrayList iterate, get, get, Algorithm 14.9, Algorithm 14.11
Overall	$O(c^3 \cdot n \cdot (n + e))$	$O(c^3 \cdot n \cdot (n + e))$	

Table 14.15: Complexity analysis of *partialCycle* (Algorithm 14.10)

The time complexity of the *merge_Cycles* (Algorithm 14.11) is $O(c^3 \cdot (n+e))$ and the space complexity is $O(c^3 \cdot (n+e))$ (Table 14.16).

Line	Time complexity	Space complexity	Comments
1	$O(c)$		Hashtable iterate
2	$O(c)$		ArrayList iterate
3	$O(n)$		HashSet containsAll
4	$O(1)$		Hashtable remove
1-7	$\boldsymbol{O(c^2 \cdot n)}$		Hashtable iterate, iterate, remove
8	$O(c)$	$O(c)$	LinkedList create
10	$O(c)$		LinkedList iterate
11	$O(1)$		LinkedList remove first
13	$O(n+e)$	$O(n+e)$	HashSet add nodes and edges of one cycle
15	$O(c)$	$O(c)$	while loop
16	$O(1)$	$O(1)$	LinkedList create
18	$O(c)$		LinkedList iterate
19	$O(1)$		LinkedList remove first
20	$O(n)$	$O(n)$	HashSet add nodes
21	$O(n+e)$	$O(n+e)$	
22	$O(n)$	$O(n)$	
25	$O(n+e)$	$O(n+e)$	HashSet add nodes and edges of one cycle
27	$O(1)$	$O(1)$	LinkedList add
30	$O(1)$		LinkedList assign
32	$O(1)$	$O(1)$	Hashtable add
33	$O(n)$		HashSet iterate
10-36	$\boldsymbol{O(c^3 \cdot (n+e))}$	$\boldsymbol{O(c^3 \cdot (n+e))}$	
37	$O(c)$		LinkedList clear
38	$O(c)$	$O(c)$	LinkedList add all
39	$O(c)$		Hashtable iterate
40	$O(n)$	$O(n)$	HashSet add nodes
39-41	$\boldsymbol{O(c \cdot n)}$	$\boldsymbol{O(c \cdot n)}$	
Overall	$O(c^3 \cdot (n+e))$	$O(c^3 \cdot (n+e))$	

Table 14.16: Complexity analysis of *merge_Cycles* (Algorithm 14.11)

14.4.2 Complexity Analysis of Split Algorithms

The actual time complexity of the *SplitConnectedComponentAtCyclePoint* (Algorithm 14.12) is $O(n+e)$ and the space complexity is $O(n+e)$, in contrast with the quadratic time complexity and the quadratic space complexity shown in Table 14.17. The reason for the difference in the time complexity and space complexity is that lines 4-8 consider all wCCs as separated subgraphs, where $wCCi = (Ni, Ei)$, $ni = |Ni|$, and $ei = |Ei|$. The time and the space needed for each $wCCi$ is $O(ni + ei)$. As $\sum ni \leqslant n$ and $\sum ei \leqslant e$, the time complexity of all $wCCi$s is given by the following summation.

$$\sum O(ni, ei) \leqslant O(n+e) \tag{14.1}$$

Line	Time complexity	Space complexity	Comments
1	$O(n)$		ArrayList iterator
2	$O(1)$	$O(1)$	Hashtable Add
4	$O(n)$	$O(n)$	HashMap iterator
6	$O(n \cdot md^2)$	$O(e \cdot n \cdot md)$	Algorithm 14.13
4-8	$\boldsymbol{O(n^2 \cdot md^2)}$		HashMap iterator, Algorithm 14.13
Overall	$O(n^2 \cdot md^2)$	$O(e \cdot n^2 \cdot md)$	

Table 14.17: Complexity analysis of *SplitConnectedComponentAtCyclePoint* (Algorithm 14.12), with md =max degree (node)

The time complexity of the *check_Start* (Algorithm 14.13) is $O(n \cdot md^2)$ and the space complexity is $O(e \cdot n \cdot md)$ (Table 14.18). The time complexity is dominated by calling Algorithm 14.14.

Line	Time complexity	Space complexity	Comments
1	$O(1)$		HashSet Create
2	$O(1)$	$O(1)$	HashSet Add
3	$O(md)$	$O(e)$	HashSet Create
4	$O(md)$	$O(md)$	HashSet Iterator
12	$O(1)$		Hashtable get
16	$O(1)$	$O(1)$	HashSet Create
17	$O(1)$	$O(1)$	HashSet Add
18	$O(1)$	$O(1)$	HashSet Add
19	$O(1)$		HashSet containsKey (get)
20	$O(1)$	$O(1)$	HashSet add
22	$O(n \cdot md)$	$O(e \cdot n)$	Algorithm 14.14
4-25	$\boldsymbol{O(n \cdot md^2)}$	$\boldsymbol{O(e \cdot n \cdot md)}$	HashSet Iterator, get create, add, add, containsKey, add, Algorithm 14.14
26	$O(n \cdot md)$	$O(n \cdot md)$	Algorithm 14.15
28	$O(1)$		ArrayList Add
Overall	$O(n \cdot md^2)$	$O(e \cdot n \cdot md)$	

Table 14.18: Complexity analysis of *check_Start* (Algorithm 14.13), with md =max degree (node)

The time complexity of the *check* (Algorithm 14.14) is $O(n \cdot md)$ and the space complexity is $O(n \cdot e)$ (Table 14.19).

Line	Time complexity	Space complexity	Comments
3	$O(md)$	$O(e)$	HashSet Create
4	$O(md)$		HashSet iterator
12	$O(1)$		Hashtable get
16	$O(1)$	$O(1)$	HashSet Add
17	$O(1)$		HashSet containsKey (get)
18	$O(1)$	$O(1)$	HashSet Add
20	$O(md)$		Algorithm 14.14
4-23	$\boldsymbol{O(md \cdot n)}$	$\boldsymbol{O(n \cdot e)}$	HashSet iterator, get, add, containsKey, add, Alg 14.14
Overall	$O(n \cdot md)$	$O(n \cdot e)$	

Table 14.19: Complexity analysis of *check* (Algorithm 14.14), with *md* =max degree (node)

The time complexity of the *createGraph* (Algorithm 14.15) is $O(n \cdot md)$ and the space complexity is $O(n \cdot md)$ (Table 14.20).

Line	Time complexity	Space complexity	Comments
1	$O(1)$	$O(1)$	Graph create
2	$O(n)$	$O(n)$	HashSet Iterate
3	$O(1)$		Hashtable get
5	$O(1)$	$O(1)$	Hashtable add
8	$O(n)$	$O(n)$	HashSet Iterate
9	$O(1)$		Hashtable get
11	$O(md)$	$O(md)$	Hashtable Iterator
12	$O(1)$		Hashtable get
13	$O(1)$		Hashtable get
16	$O(1)$	$O(1)$	Hashtable add
8-20	$\boldsymbol{O(n \cdot md)}$	$\boldsymbol{O(n \cdot md)}$	HashSet Iterate, get, Iterator, get, get, add
Overall	$O(n \cdot md)$	$O(n \cdot md)$	

Table 14.20: Complexity analysis of *createGraph* (Algorithm 14.15), with *md* =max degree (node)

14.4.3 Complexity Analysis of Detect DAGs and Trees Algorithms

The time complexity of the *detectDAGsTrees* (Algorithm 14.16) is $O(n^2 \cdot md + e)$ and the space complexity is $O(n^2 \cdot md + e)$ (Table 14.21).

Line	Time complexity	Space complexity	Comments
1	$O(n)$	$O(n)$	ArrayList Iterator
3	$O(1)$	$O(1)$	ArrayList add
5	$O(n)$	$O(n)$	ArrayList Iterator
7	$O(1)$	$O(1)$	ArrayList add
9	$O(n + e)$	$O(n + e)$	ArrayList Iterator
10	$O(1)$	$O(1)$	ArrayList add
12	$O(n)$	$O(n)$	ArrayList Iterator
13	$O(1)$	$O(1)$	ArrayList add
15	$O(n)$	$O(n)$	ArrayList Iterator
16	$O(1)$	$O(1)$	ArrayList add
21	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	Algorithm 14.17
23	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	Algorithm 14.20
25	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	Algorithm 14.23
28	$O(n + e)$	$O(n + e)$	Algorithm 14.26
Overall	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	

Table 14.21: Complexity analysis of *detectDAGsTrees* (Algorithm 14.16)

The time complexity of the *check_DownTreeStart* (Algorithm 14.17) is $O(n^2 \cdot md + e)$ and the space complexity is $O(n^2 \cdot md + e)$ (Table 14.22).

Line	Time complexity	Space complexity	Comments
2	$O(1)$	$O(1)$	ArrayList Create
4	$O(n)$		ArrayList Iterator
7	$O(1)$		ArrayList get
9	$O(1)$	$O(1)$	ArrayList add
10	$O(n \cdot md)$	$O(n \cdot md)$	Algorithm 14.18
12	$O(n + e)$	$O(n + e)$	Algorithm 14.24
18	$O(n)$	$O(n)$	ArrayList add
19	$O(n)$	$O(n)$	ArrayList add at end
20	$O(n)$	$O(n)$	ArrayList add at end
21	$O(n)$	$O(n)$	ArrayList Iterator
22	$O(n)$		ArrayList Iterator
25	$O(n)$		ArrayList clear
27	$O(1)$	$O(1)$	ArrayList add
28	$O(n \cdot md)$	$O(n \cdot md)$	Algorithm 14.18
30	$O(n + e)$	$O(n + e)$	Algorithm 14.24
21-33	$\boldsymbol{O(n^2 \cdot md + e)}$	$\boldsymbol{O(n^2 \cdot md + e)}$	iterate, clear, add, Algorithm 14.18, Algorithm 14.24
Overall	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	

Table 14.22: Complexity analysis of *check_DownTreeStart* (Algorithm 14.17)

The time complexity of the *check_DownTree* (Algorithm 14.18) is $O(n \cdot md)$ and the space complexity is $O(n \cdot md)$ (Table 14.23).

Line	Time complexity	Space complexity	Comments
4	$O(md)$		Algorithm 14.19
9	$O(md)$	$O(md)$	HashSet Iterator
11	$O(1)$	$O(1)$	ArrayList Add
12	$O(n)$		ArrayList indexOf
13	$O(n)$		ArrayList lastIndexOf
17	$O(n)$	$O(n)$	Algorithm 14.18
18	$O(n)$		ArrayList lastIndexOf
9-22	$\boldsymbol{O(n \cdot md)}$	$\boldsymbol{O(n \cdot md)}$	iterate, add, indexOf, lastIndexOf, Algorithm 14.18
Overall	$O(n \cdot md)$	$O(n \cdot md)$	

Table 14.23: Complexity analysis of *check_DownTree* (Algorithm 14.18)

The time complexity of the *countOneDirectionIncomingEdges* (Algorithm 14.19) is $O(md)$ (Table 14.24).

Line	Time complexity	Space complexity	Comments
2	$O(md)$		HashSet Iterator
3	$O(1)$		HashSet contains
Overall	$O(md)$		

Table 14.24: Complexity analysis of *countOneDirectionIncomingEdges* (Algorithm 14.19)

The time complexity of the *check_UpTreeStart* (Algorithm 14.20) is $O(n^2 \cdot md + e)$ and the space complexity is $O(n^2 \cdot md + e)$ (Table 14.25).

Line	Time complexity	Space complexity	Comments
2	$O(1)$		ArrayList Create
4	$O(n)$	$O(n)$	ArrayList Iterator
7	$O(1)$		ArrayList get
9	$O(1)$	$O(1)$	ArrayList add
10	$O(n \cdot md)$	$O(n \cdot md)$	Algorithm 14.21
12	$O(n + e)$	$O(n + e)$	Algorithm 14.25
18	$O(n)$	$O(n)$	ArrayList add
19	$O(n)$	$O(n)$	ArrayList add at end
20	$O(n)$	$O(n)$	ArrayList add at end
21	$O(n)$	$O(n)$	ArrayList Iterator
22	$O(n)$		ArrayList Iterator
25	$O(n)$		ArrayList clear
27	$O(1)$	$O(1)$	ArrayList add
28	$O(n \cdot md)$	$O(n \cdot md)$	Algorithm 14.21
30	$O(n + e)$	$O(n + e)$	Algorithm 14.25
21-33	$\boldsymbol{O(n^2 \cdot md + e)}$	$\boldsymbol{O(n^2 \cdot md + e)}$	iterate, clear, add, Algorithm 14.21, Algorithm 14.25
Overall	$O(n^2 \cdot md + e)$	$O(n^2 \cdot md + e)$	

Table 14.25: Complexity analysis of *check_UpTreeStart* (Algorithm 14.20)

The time complexity of the *check_UpTree* (Algorithm 14.21) is $O(n \cdot md)$ and the space complexity is $O(n \cdot md)$ (Table 14.26).

Line	Time complexity	Space complexity	Comments
4	$O(md)$		Algorithm 14.22
9	$O(md)$	$O(md)$	HashSet Iterator
11	$O(1)$	$O(1)$	ArrayList add
12	$O(n)$		ArrayList indexOf
13	$O(n)$		ArrayList lastIndexOf
17	$O(n)$	$O(n)$	Algorithm 14.21
18	$O(n)$		ArrayList lastIndexOf
9-22	$\boldsymbol{O(n \cdot md)}$	$\boldsymbol{O(n \cdot md)}$	iterate, add, indexOf, lastIndexOf, Algorithm 14.21
Overall	$O(n \cdot md)$	$O(n \cdot md)$	

Table 14.26: Complexity analysis of *check_UpTree* (Algorithm 14.21)

The time complexity of the *countOneDirectionOutgoingEdges* (Algorithm 14.22) is $O(md)$ (Table 14.27).

Line	Time complexity	Space complexity	Comments
2	$O(md)$		HashSet Iterator
3	$O(1)$		HashSet contains
Overall	$O(md)$		

Table 14.27: Complexity analysis of *countOneDirectionOutgoingEdges* (Algorithm 14.22)

The time complexity of the *check_DoubleEdges* (Algorithm 14.23) is $O(n \cdot md + e)$ and the space complexity is $O(n \cdot md + e)$ (Table 14.28).

Line	Time complexity	Space complexity	Comments
1	$O(n \cdot md + e)$	$O(n \cdot md + e)$	Algorithm 14.17
3	$O(n \cdot md + e)$	$O(n \cdot md + e)$	Algorithm 14.20
Overall	$O(n \cdot md + e)$	$O(n \cdot md + e)$	

Table 14.28: Complexity analysis of *check_DoubleEdges* (Algorithm 14.23)

The time complexity of the *createDownTree* (Algorithm 14.24) is $O(n + e)$ and the space complexity is $O(n + e)$ (Table 14.29).

Line	Time complexity	Space complexity	Comments
1	$O(1)$	$O(1)$	Hashtable Create
2	$O(n)$	$O(n)$	Hashtable Add
4	$O(1)$	$O(1)$	Hashtable Add
Overall	$O(n + e)$	$O(n + e)$	

Table 14.29: Complexity analysis of *createDownTree* (Algorithm 14.24)

The time complexity of the *createUpTree* (Algorithm 14.25) is $O(n+e)$ and the space complexity is $O(n+e)$ (Table 14.30).

Line	Time complexity	Space complexity	Comments
1	$O(1)$	$O(1)$	Hashtable Create
2	$O(n)$	$O(n)$	Hashtable Add
4	$O(1)$	$O(1)$	Hashtable Add
Overall	$O(n+e)$	$O(n+e)$	

Table 14.30: Complexity analysis of *createUpTree* (Algorithm 14.25)

The time complexity of the *createDAG* (Algorithm 14.26) is $O(n+e)$ and the space complexity is $O(n+e)$ (Table 14.31).

Line	Time complexity	Space complexity	Comments
1	$O(1)$	$O(1)$	Hashtable Create
2	$O(n)$	$O(n)$	Hashtable Add
4	$O(1)$	$O(1)$	Hashtable Add
Overall	$O(n+e)$	$O(n+e)$	

Table 14.31: Complexity analysis of *createDAG* (Algorithm 14.26)

14.4.4 Complexity Analysis of Hierarchy Construction Algorithms

For one weakly connected component, the time complexity of the $startCreateMetaNodes$ (Algorithm 14.27) is $O(n' \cdot n + n')$ and the space complexity is $O(n' \cdot n + n')$ (Table 14.32). Because there are many weakly connected components, the time complexity is $O(w \cdot n' \cdot n + w \cdot n')$ and the space complexity is $O(w \cdot n' \cdot n + w \cdot n')$.

Line	Time complexity	Space complexity	Comments
1	$O(1)$	$O(1)$	Hashtable create
2	$O(1)$	$O(1)$	Hashtable create
3	$O(1)$	$O(1)$	Hashtable create
7	$O(n' \cdot n + n')$	$O(n' \cdot n + n')$	Algorithm 14.28
12	$O(n')$	$O(n')$	ArrayList add
13	$O(1)$	$O(1)$	Hashset add
16	$O(n')$	$O(n')$	ArrayList add
17	$O(1)$	$O(1)$	Hashset add
Overall	$O(n' \cdot n + n')$	$O(n' \cdot n + n')$	

Table 14.32: Complexity analysis of $startCreateMetaNodes$ (Algorithm 14.27),with $n' = \#$ meta-nodes $\in$ Two-level hierarchy

The time complexity of the $createMetaNodesByBFS$ algorithm (Algorithm 14.28) is $O(n' \cdot n + n')$ and the space complexity is $O(n' \cdot n + n')$ (Table 14.33).

Line	Time complexity	Space complexity	Comments
2	$O(1)$	$O(1)$	Hashtable create
3	$O(1)$	$O(1)$	LinkedList create
4	$O(1)$	$O(1)$	HashSet create
6	$O(1)$	$O(1)$	LinkedList add
7	$O(1)$	$O(1)$	HashSet add
8	$O(n')$	$O(n')$	LinkedList Iterator
9	$O(1)$		LinkedList remove first
10	$O(1)$		
11	$O(1)$		Hashtable get
13	$O(n)$	$O(n)$	Hashset Iterator
14	$O(1)$	$O(1)$	ArrayList add
16	$O(1)$		
17	$O(1)$		Hashtable get
19	$O(n)$	$O(n)$	Hashset Iterator
20	$O(1)$	$O(1)$	ArrayList add
22	$O(1)$		
23	$O(1)$		Hashtable get
25	$O(n)$	$O(n)$	Hashset Iterator
26	$O(1)$	$O(1)$	ArrayList add
29	$O(1)$	$O(1)$	Hashset add
31	$O(n')$	$O(n')$	Hashtable Iterator
32	$O(1)$		Hashset contains
33	$O(1)$	$O(1)$	Hashset add
34	$O(1)$	$O(1)$	LinkedList add
35	$O(1)$	$O(1)$	Hashtable add
37	$O(1)$	$O(1)$	Hashtable add
31-39	$\boldsymbol{O(n')}$	$\boldsymbol{O(n')}$	iterate, contains, add, add, add, add
8-41	$\boldsymbol{O(n' \cdot n + n')}$	$\boldsymbol{O(n' \cdot n + n')}$	
Overall	$O(n' \cdot n + n')$	$O(n' \cdot n + n')$	

Table 14.33: Complexity analysis of $createMetaNodesByBFS$ algorithm (Algorithm 14.28), with $n' = \#$ meta-nodes $\in$ Two-level hierarchy

The time complexity of the $startCreateMetaEdges$ (Algorithm 14.29) is $O(n' \cdot n^3 \cdot md')$ and the space complexity is $O(n' \cdot n^3 \cdot md')$ (Table 14.34).

Line	Time complexity	Space complexity	Comments
2	$O(n')$	$O(n')$	Hashtable get
3	$O(1)$	$O(1)$	
4	$O(n' \cdot n^3 \cdot md')$	$O(n' \cdot n^3 \cdot md')$	Algorithm 14.30
Overall	$O(n' \cdot n^3 \cdot md')$	$O(n' \cdot n^3 \cdot md')$	

Table 14.34: Complexity analysis of $startCreateMetaEdges$ (Algorithm 14.29), with $n' = \#$ meta-nodes $\in$ Two-level hierarchy

The time complexity of the *createMetaEdges* (Algorithm 14.30) is $O(n^3 \cdot md')$ and the space complexity is $O(n^3 \cdot md')$ (Table 14.35).

Line	Time complexity	Space complexity	Comments
1	$O(1)$		
2	$O(1)$		Hashtable get
3	$O(1)$		Hashtable get
4	$O(1)$		
5	$O(1)$		Hashtable get
6	$O(1)$		Hashtable get
9	$O(n)$	$O(n)$	LinkedList add all
10	$O(n)$	$O(n)$	LinkedList Iterator
11	$O(1)$	$O(1)$	Hashtable create
12	$O(1)$	$O(1)$	Hashtable create
13	$O(1)$	$O(1)$	Hashtable create
14	$O(1)$	$O(1)$	Hashtable create
15	$O(1)$		LinkedList remove first
16	$O(1)$		Hashtable get
17	$O(1)$		Hashtable get
18	$O(1)$	$O(1)$	Hashtable create
19	$O(1)$	$O(1)$	Hashtable create
20	$O(n)$	$O(n)$	Hashtable iterate
21	$O(1)$		Hashtable get
22	$O(1)$		Hashtable get
23	$O(1)$		
24	$O(1)$		Hashtable get
26	$O(n \cdot md')$	$O(n \cdot md')$	Algorithm 14.31
20-27	$\boldsymbol{O(n^2 \cdot md')}$	$\boldsymbol{O(n^2 \cdot md')}$	iterate, get, get, get, Algorithm 14.31
28	$O(1)$	$O(1)$	Algorithm 14.32
10-29	$\boldsymbol{O(n^3 \cdot md')}$	$\boldsymbol{O(n^3 \cdot md')}$	
Overall	$O(n^3 \cdot md')$	$O(n^3 \cdot md')$	

Table 14.35: Complexity analysis of *createMetaEdges* (Algorithm 14.30)

The time complexity of the *startCheckCycleNodeEdges* (Algorithm 14.31) is $O(n \cdot md')$ and the space complexity is $O(n \cdot md')$ (Table 14.36).

Line	Time complexity	Space complexity	Comments
2	$O(md')$	$O(md')$	Hashtable Iterator
3	$O(1)$	$O(1)$	Algorithm 14.38
6	$O(1)$	$O(1)$	Hashtable add
9	$O(n)$	$O(1)$	Algorithm 14.33
2-11	$\boldsymbol{O(n \cdot md')}$	$\boldsymbol{O(n \cdot md')}$	iterate, Algorithm 14.38, add, Algorithm 14.33
14	$O(md')$	$O(md')$	Hashtable Iterator
15	$O(1)$	$O(1)$	Algorithm 14.38
18	$O(1)$	$O(1)$	Hashtable add
21	$O(n)$	$O(1)$	Algorithm 14.34
14-23	$\boldsymbol{O(n \cdot md')}$	$\boldsymbol{O(n \cdot md')}$	iterate, Algorithm 14.38, add, Algorithm 14.34
Overall	$O(n \cdot md')$	$O(n \cdot md')$	

Table 14.36: Complexity analysis of *startCheckCycleNodeEdges* (Algorithm 14.31), with md' = max degree (node) for *wCC*

The time complexity of the *combineDiffEdgesType* (Algorithm 14.32) is $O(1)$ and the space complexity is $O(1)$ (Table 14.37).

Line	Time complexity	Space complexity	Comments
6	$O(1)$	$O(1)$	Hashtable add
10	$O(1)$	$O(1)$	Hashtable add
13	$O(1)$	$O(1)$	Hashtable add
17	$O(1)$	$O(1)$	Hashtable add
20	$O(1)$	$O(1)$	Hashtable add
Overall	$O(1)$	$O(1)$	

Table 14.37: Complexity analysis of *combineDiffEdgesType* (Algorithm 14.32)

The time complexity of the *check_CycleNodeOutgoingEdgesInDAGorTree* (Algorithm 14.33) is $O(n)$ and the space complexity is $O(1)$ (Table 14.38).

Line	Time complexity	Space complexity	Comments
1	$O(1)$		Hashmap get
3	$O(1)$	$O(1)$	Hashtable add
9	$O(1)$	$O(1)$	Hashtable add
13	$O(1)$	$O(1)$	Hashtable add
20	$O(n)$		Algorithm 14.33
Overall	$O(n)$	$O(1)$	

Table 14.38: Complexity analysis of *check_CycleNodeOutgoingEdgesInDAGorTree* (Algorithm 14.33)

The time complexity of the $check\_CycleNodeIncomingEdgesInDAGorTree$ (Algorithm 14.34) is $O(n)$ and the space complexity is $O(1)$ (Table 14.39).

Line	Time complexity	Space complexity	Comments
Overall	$O(n)$	$O(1)$	Algorithm 14.34 is analogous to Algorithm 14.33

Table 14.39: Complexity analysis of $check\_CycleNodeIncomingEdgesInDAGorTree$ (Algorithm 14.34)

The time complexity of the $check\_DownTreeByBFSLevels$ (Algorithm 14.35) is $O(n^2 \cdot md' + n \cdot e)$ and the space complexity is $O(n^2)$ (Table 14.40).

Line	Time complexity	Space complexity	Comments
2	$O(1)$	$O(1)$	Hashset Create
5	$O(n)$		Hashtable clear
6	$O(1)$	$O(1)$	Hashtable add
7	$O(1)$	$O(1)$	Hashtable add
8	$O(e')$		Algorithm 14.37
9	$O(n \cdot md)$		Algorithm 14.36
14	$O(n)$	$O(n)$	ArrayList add all
15	$O(n)$	$O(n)$	ArrayList add all at end
16	$O(n)$	$O(n)$	ArrayList add all at end
17	$O(n)$		ArrayList Iterator
18	$O(n)$		Hashtable clear
19	$O(n)$		Hashset clear
20	$O(n)$		Hashtable clear
21	$O(1)$	$O(1)$	Hashset add
22	$O(e')$		Algorithm 14.37
23	$O(n)$		Hashset add
24	$O(n \cdot md')$	$O(n)$	Algorithm 14.36
17-28	$\boldsymbol{O(n^2 \cdot md' + n \cdot e)}$	$\boldsymbol{O(n^2)}$	ArrayList Iterator
Overall	$O(n^2 \cdot md' + n \cdot e)$	$O(n^2)$	

Table 14.40: Complexity analysis of $check\_DownTreeByBFSLevels$ (Algorithm 14.35)

The time complexity of the $check\_DownTreePotentialRootNodeByBFS$ (Algorithm 14.36) is $O(n \cdot md')$ and the space complexity is $O(n)$ (Table 14.41).

Line	Time complexity	Space complexity	Comments
1	$O(1)$	$O(1)$	Hashtable Create
2	$O(1)$	$O(1)$	Hashtable Add
4	$O(n)$		Hashtable iterate
5	$O(1)$	$O(1)$	Hashtable Create
6	$O(n)$		Hashtable iterate
8	$O(1)$		Hashtable remove
9	$O(md')$		Hashtable iterate
15	$O(1)$	$O(1)$	Algorithm 14.38
16	$O(1)$		Hashset contains
19	$O(1)$	$O(1)$	Hashset add
20	$O(1)$	$O(1)$	Hashtable add
21	$O(1)$	$O(1)$	Hashtable add
27	$O(1)$		
4-28	$\boldsymbol{O(n)}$		iterate, Create, iterate, remove, iterate, Algorithm 14.38, contains, add, add, add
Overall	$O(n \cdot md')$	$O(n)$	

Table 14.41: Complexity analysis of $check\_DownTreePotentialRootNodeByBFS$ (Algorithm 14.36)

The time complexity of the $setAllEdgesAsNotVisited$ (Algorithm 14.37) is $O(e')$ and the space complexity is $O(e')$ (Table 14.42).

Line	Time complexity	Space complexity	Comments
2	$O(e')$	$O(e')$	HashSet Create
3	$O(e')$		HashSet Iterator
Overall	$O(e')$	$O(e')$	

Table 14.42: Complexity analysis of $setAllEdgesAsNotVisited$ (Algorithm 14.37), with $e' = \#$ edges of meta-graph

The time complexity of the $isBackEdge$ (Algorithm 14.38) is $O(1)$ and the space complexity is $O(1)$ (Table 14.43).

Line	Time complexity	Space complexity	Comments
1	$O(1)$	$O(1)$	Hashtable get
Overall	$O(1)$	$O(1)$	

Table 14.43: Complexity analysis of $isBackEdge$ (Algorithm 14.38)

14.5 Complexity Analysis of Drawing

14.5.1 Complexity Analysis of Computing Size of all Meta Nodes

The time complexity and the space complexity of the *drawOneIsolatedCycle* (Algorithm 14.42) and the *drawOneIsolatedDAG* (Algorithm 14.47) depend on approximation algorithms used by Bachmaier's algorithm [22] and Sugiyama framework [87]. As these two algorithms spread to other algorithms in this part, we keep the complexity analysis of this part opened.

14.5.2 Complexity Analysis of Final Drawing

The time complexity of the *drawTwoLevelsAreaAwareHierarchy* (Algorithm 14.49) is $O(n' \cdot (n+e))$ (Table 14.44).

Line	Time complexity	Space complexity	Comments
1	$O(n')$		Two-level Hierarchy (tree) iterate
5	$O(n+e)$		Algorithm 14.50
8	$O(n+e)$		Algorithm 14.54
10	$O(1)$		Algorithm 14.49
1-11	$\boldsymbol{O(n')}$		Two-level Hierarchy (tree) iterate, Algorithm 14.50, Algorithm 14.54
Overall	$O(n' \cdot (n+e))$		

Table 14.44: Complexity analysis of *drawTwoLevelsAreaAwareHierarchy* (Algorithm 14.49), with $n' = \#$ meta-nodes $\in$ Two-level hierarchy

The time complexity of the *drawLevelOneMetaNodes* (Algorithm 14.50) is $O(n+e)$ (Table 14.45).

Line	Time complexity	Space complexity	Comments
2	$O(n+e)$		Algorithm 14.51
4	$O(n+e)$		Algorithm 14.52
6	$O(n+e)$		Algorithm 14.53
Overall	$O(n+e)$		

Table 14.45: Complexity analysis of *drawLevelOneMetaNodes* (Algorithm 14.50)

The time complexity of the *drawLevelOneCycleMetaNode* (Algorithm 14.51) is $O(n+e)$ (Table 14.46).

Line	Time complexity	Space complexity	Comments
7	$O(n+e)$		translate all elements of the tree
9	$O(n+e)$		ArrayDeque add all elements of the tree
Overall	$O(n+e)$		

Table 14.46: Complexity analysis of *drawLevelOneCycleMetaNode* (Algorithm 14.51)

The time complexity of the $drawLevelOneDAGMetaNode$ (Algorithm 14.52) is $O(n+e)$ (Table 14.47).

Line	Time complexity	Space complexity	Comments
7	$O(n+e)$		translate all elements of the tree
9	$O(n+e)$		ArrayDeque add all elements of the tree
Overall	$O(n+e)$		

Table 14.47: Complexity analysis of $drawLevelOneDAGMetaNode$ (Algorithm 14.52)

The time complexity of the $drawLevelOneTreeMetaNode$ (Algorithm 14.53) is $O(n+e)$ (Table 14.48).

Line	Time complexity	Space complexity	Comments
7	$O(n+e)$		translate all elements of the tree
9	$O(n+e)$		ArrayDeque add all elements of the tree
Overall	$O(n+e)$		

Table 14.48: Complexity analysis of $drawLevelOneTreeMetaNode$ (Algorithm 14.53)

The time complexity of the $drawLevelTwoMetaNodes$ (Algorithm 14.54) is $O(n+e)$ (Table 14.49).

Line	Time complexity	Space complexity	Comments
2	$O(n+e)$		Algorithm 14.55
4	$O(n+e)$		Algorithm 14.56
Overall	$O(n+e)$		

Table 14.49: Complexity analysis of $drawLevelTwoMetaNodes$ (Algorithm 14.54)

The time complexity of the $drawLevelTwoDAGMetaNode$ (Algorithm 14.55) is $O(n + e)$ (Table 14.50).

Line	Time complexity	Space complexity	Comments
7	$O(n + e)$		translate all elements of the DAG
8	$O(n)$		ArrayList iterate
9	$O(1)$		Hashtable get
12	$O(n + e)$		ArrayDeque add all elements of the DAG
Overall	$O(n + e)$		

Table 14.50: Complexity analysis of $drawLevelTwoDAGMetaNode$ (Algorithm 14.55)

The time complexity of the $drawLevelTwoTreeMetaNode$ (Algorithm 14.56) is $O(n + e)$ (Table 14.51).

Line	Time complexity	Space complexity	Comments
7	$O(n + e)$		translate all elements of the tree
8	$O(n)$		ArrayList iterate
9	$O(1)$		Hashtable get
12	$O(n + e)$		ArrayDeque add all elements of the tree
Overall	$O(n + e)$		

Table 14.51: Complexity analysis of $drawLevelTwoTreeMetaNode$ (Algorithm 14.56)

The time complexity of the $rotateCycleByTorque$ (Algorithm 14.57) is $O(n' \cdot e)$ (Table 14.52).

Line	Time complexity	Space complexity	Comments
Overall	$O(n' \cdot e)$		rotate the cyclic subgraphs similar to the approach of Archambault et al. [17]

Table 14.52: Complexity analysis of $rotateCycleByTorque$ (Algorithm 14.57), with $n' = \#$ cyclic subgraphs $\in$ Two-level hierarchy

The time complexity of the *drawLinesConnectDuplicatedNodes* (Algorithm 14.58) is $O(n \cdot n')$ (Table 14.53).

Line	Time complexity	Space complexity	Comments
1	$O(n')$		Hashtable iterate
7	$O(1)$	$O(1)$	ArrayDeque create
8	$O(n)$		Hashmap iterate
9	$O(1)$		Hashtable get
11	$O(1)$		Hashmap get
13	$O(1)$		ArrayDeque add
15	$O(1)$		Hashtable get
17	$O(1)$		Hashmap get
19	$O(1)$		ArrayDeque add
8-21	$\boldsymbol{O(n \cdot 1)}$		create, iterate, get, get, add, get, get, add, add all, Algorithm 14.58
22	$O(n)$		ArrayDeque add all
24	$O(n)$		Algorithm 14.58
Overall	$O(n \cdot n')$		

Table 14.53: Complexity analysis of *drawLinesConnectDuplicatedNodes* (Algorithm 14.58), with $n' = \#$ meta-nodes $\in$ Two-level hierarchy

Chapter 15

Results

We tested our approach on many small, medium, large, and very large data sets. Table 15.1 shows the properties of the two examples presented. We describe the results of the synthetic data set used to explain split and the **usage** dependencies of the JFtp software data set [29] in detail. We will describe more examples in next part (Chapter 22).

Synthetic data set

The graph of the synthetic data set (Figure 15.1(a)) consists of 26 nodes and 34 edges in one wCC that is decomposed into six subgraphs: four ntCSs, one tree, and one DAG. Representing each of these subgraphs by a meta-node and computing the meta-edges between these meta-nodes yields the meta-graph shown in Figure 15.1(b). This meta-graph is classified as down-tree. The resulting graph hierarchy is then used as input of the drawing stage leading to the layout shown in Figure 15.1(c). Both the overall down-tree structure of the meta-graph and the structure of the lowest level can be seen. The blue shapes surrounding the topological components of the initial graph represent the meta-nodes of the down-tree. Additionally, the meta-edges are displayed in blue, too.

Usage dependencies of the JFtp software data set

The graph of the **usage** dependencies of the JFtp software data set [29] consists of 78 nodes and 38 edges in 45 single nodes and four wCCs: two trees, one DAG, and one mixed wCCs. The mixed wCC is decomposed into four subgraphs: one ntCS, two trees, and one DAG. Representing each of these subgraphs by a meta-node, and computing the meta-edges between these meta-nodes yields the meta-graph shown in Figure 15.2(a). The meta-graph is classified as trivial up-tree. The resulting graph hierarchy is then used as input of the drawing stage leading to the layout shown in Figure 15.2(b). On the lowest level, the cyclic drawing of the ntCS can be seen, which shares some nodes with the other subgraphs (cyan lines).

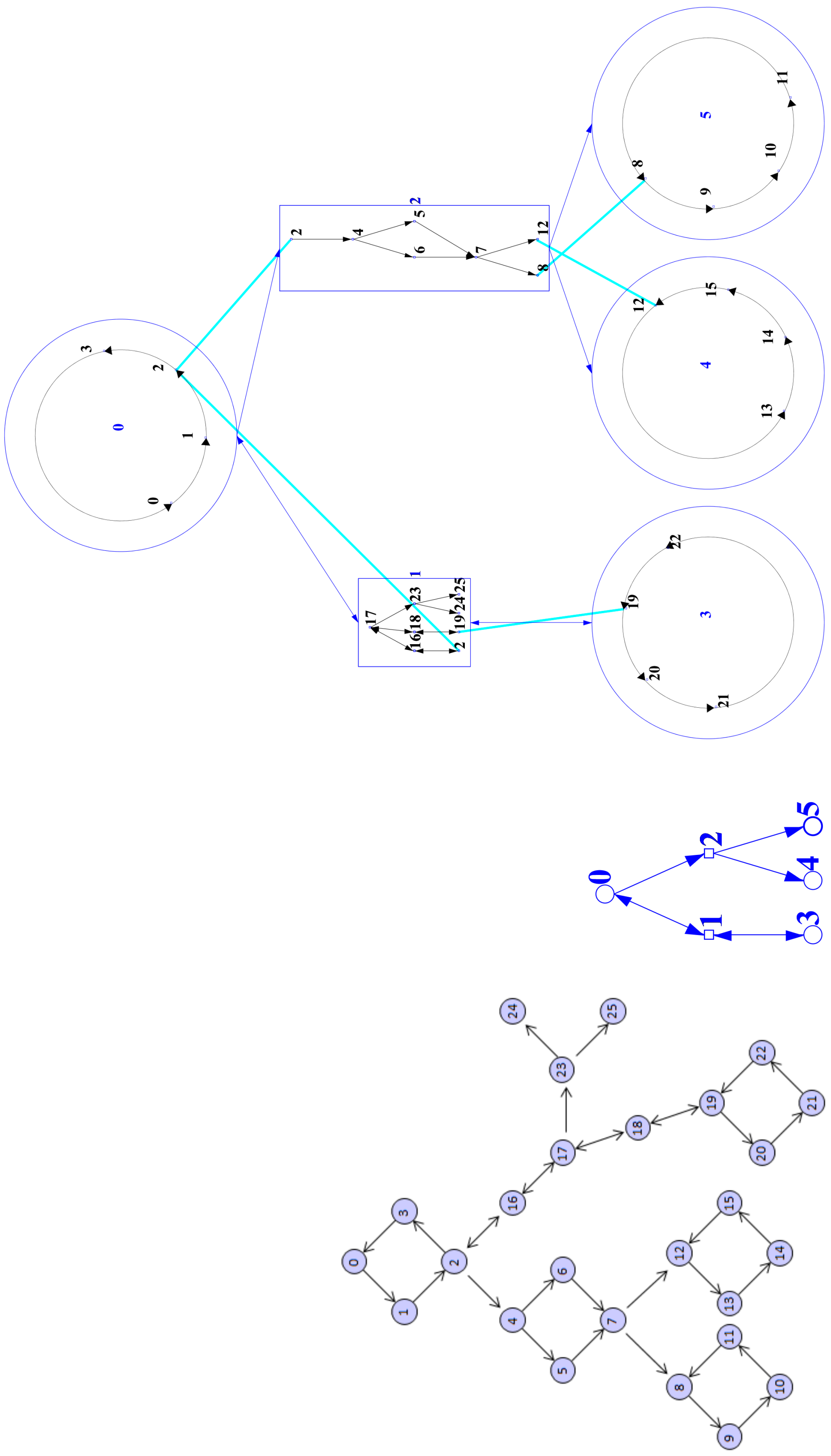
From a software engineering point of view, the design of JFtp is clear. Three components—the two trees and the DAG—are separated from the remaining code. In general, cyclic dependencies in software systems are unwanted [75]. Although the central component contains a

#nodes	*#edges*	Figure	Example
26	34	15.1	synthetic example (split)
78	38	15.1	**usage** dependencies of the JFtp software

Table 15.1: Examples.

cycle, the cycle is rather small. The layout shows the cyclic dependency between four classes that can be investigated by a software engineer who should decide, whether this design is tolerable according to software engineering guidelines and soft requirements like runtime.

As can be seen from the figures, neither the cyclic layout of Bachmaier et al. [22] (Figure 15.2(c)) nor the hierachical Sugiyama layout [87] (Figure 15.2(d)) show the structure of the wCCs that is revealed by our approach (Figure 15.2(b)).

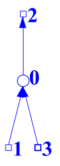


(a) Schematic drawing

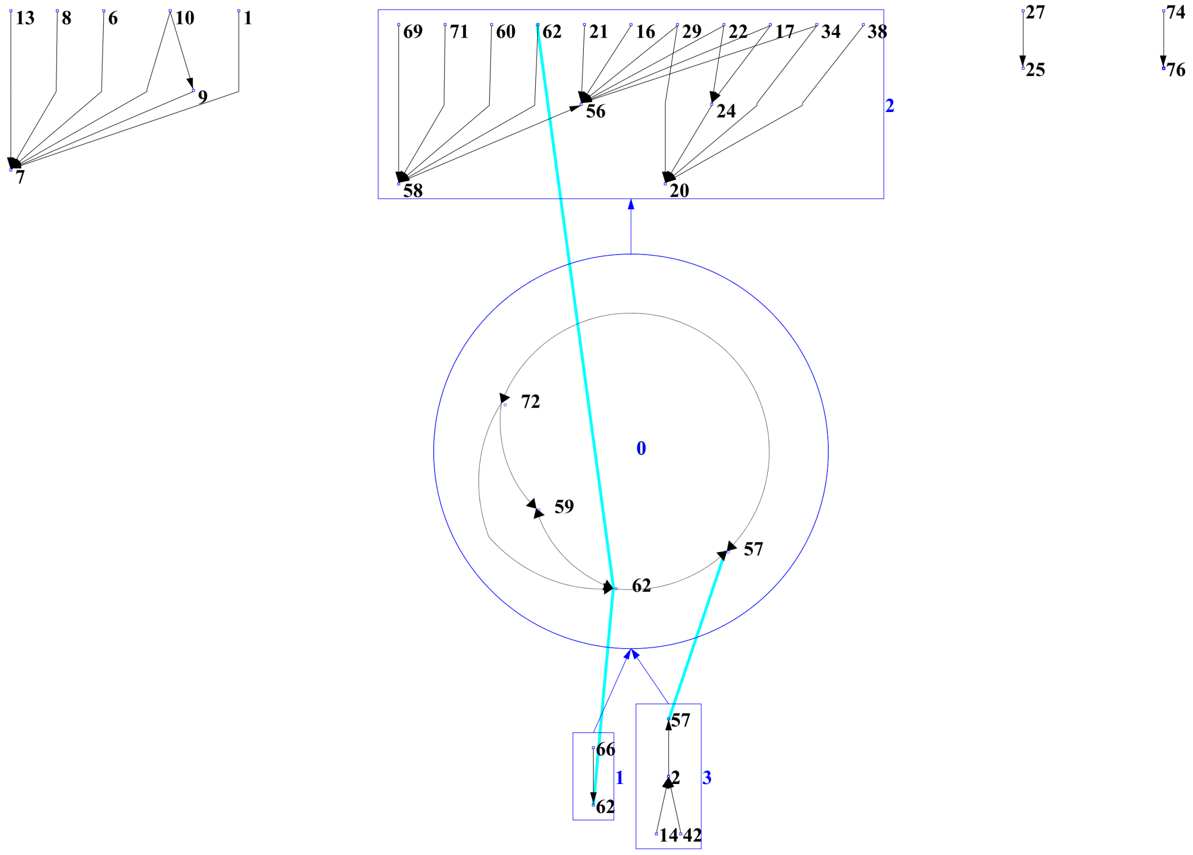
(b) Meta-Graph

(c) New topological layout

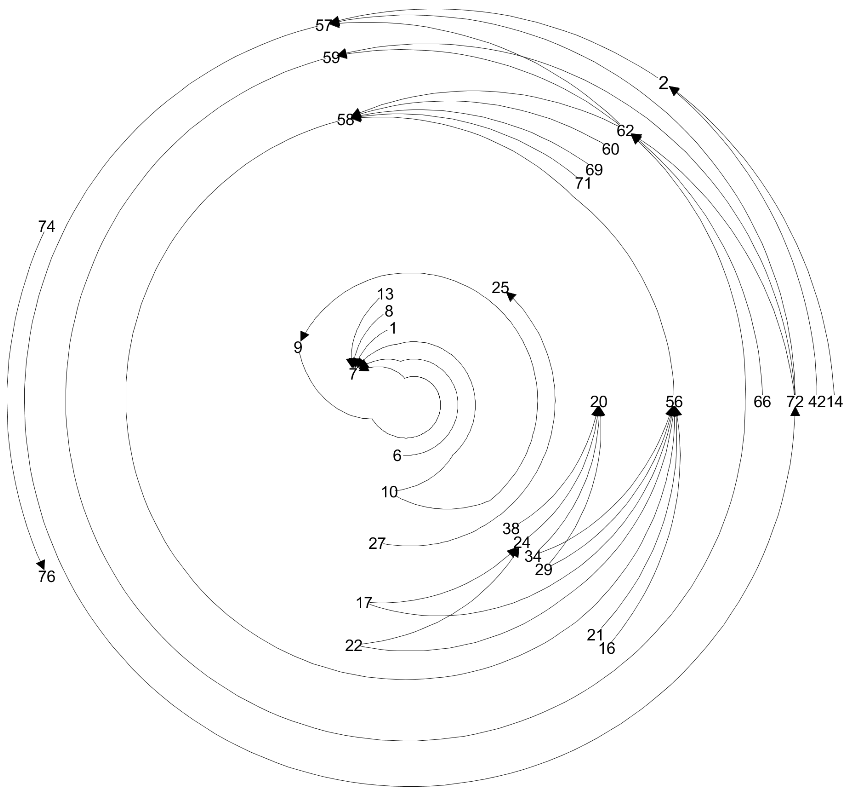
Figure 15.1: Synthetic example showing the layout of directed graphs.



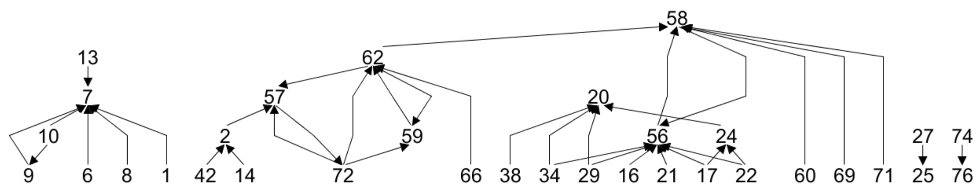
(a)



(b) New topological layout



(c) Cyclic layout which is produced by Gravisto Toolkit [23]



(d) Sugiyama layout which is produced by Gravisto Toolkit [23]

Figure 15.1: Layout of the directed graph encoding the **usage** dependencies of JFtp software (wCCs only, without single nodes). 15.2(a) Meta-Graph of the mixed wCC

Chapter 16

Conclusion

We proposed a new approach for drawing directed graphs based on a hierarchical, topological decomposition into cyclic and acyclic subgraphs. This allows to show cyclic and acyclic structures more clearly than previous approaches, thus enhancing its usefulness for applications in Bio-Informatics, Biology, and Software Engineering. The approach is based on a rigorous definition of the extracted subgraphs and enhanced algorithms for their detection. It overcomes many problems inherent in using the Sugiyama layout or the cyclic layout only, and improves on using circular layouts for non-trivial cyclic subgraphs.

Applying methods for computing hierarchies of arbitrary graphs can help to manage topological components that are still very large after decomposition. Further, adding techniques from Information Visualization to interact with the hierarchy producing arbitrary cuts will enhance the support for analyzing very large graphs.

Part III

The Small-Multiples Node-Link Visualization

Chapter 17

Preamble

Part I targeted developing IMMV visualizing software structures containing multiple types of relations. Further, in a user study, it was determined which visual patterns can be observed using IMMV and PNLV. Based on class views, these two approaches support finding similarities and differences between different edge types. Primarily, local structures such as nodes having many incoming or outgoing edges can be identified using these approaches. However, global structures involving paths from one node to other nodes like non-trivial cyclic subgraphs or hierarchical structures are not readily identified using these approaches. Therefore, Part II provided the topological approach for drawing directed graphs separating it into non-trivial cyclic subgraphs, trees, and DAGs and using drawing algorithms that fit each of them.

This part describes the extended version of the small-multiples node-link visualization (SMNLV) that was introduced by Abuthawabeh and Zeckzer [11]. They propose visualizing multiple edge types of directed graphs using small-multiples, where each tile is dedicated to one edge-type graphs. For each of these graphs, the topological approach of Abuthawabeh and Zeckzer [10] is used to highlight non-trivial cycles, trees, and DAGs showing global structures. They showed one use case, that analyzes the structure of their tool implementation by itself.

The extended version expanded the related work (Chapter 19) with additional related work summarized in Section 19.2 and Section 19.3. Moreover, Chapter 21 was added to describe other approaches to display several relation types in more detail. Chapter 22 was extended by showing four additional use cases.

Chapter 18

Introduction

Software engineering tasks like understanding of (legacy) software, checking guidelines, improving product lines, finding structure, or re-engineering of existing software require the analysis of the static software structure [42]. Often, the actual structure of the implementation is different from the designed one. Therefore, the actual structure is extracted from the code and visualization is used to display the actual structure, sometimes comparing it to the planned one [78].

The optimal visualization of the structure depends on the task at hand. In general, the software structure is mapped to a graph and graph drawing is used for display [42]. The task of drawing these graphs becomes more involved if not only one type of relations, e.g., method calls, but many relation types should be analyzed at the same time. We restrict our discussion to object-oriented programming using Java as example.

Our contributions in this part are:

- Small-multiples node-link visualization (SMNLV): each relation type is shown in a separate node-link view.
- Using drawing algorithms that are optimized for each type of relationship while displaying multiple relationships simultaneously.
- Using highlighting and coordinated views to link the individual views.
- Providing five use cases analyzing the structures of SMNLV itself, JUnit 4.12 [4], JFtp 1.59 [2], Stripes 1.5.7 [1], and Checkstyle 6.5 [3].

Chapter 19

Related Work

In the following, work related to recent visualization approaches for analyzing multiple relations between software artifacts (Section 19.1), to small-multiples visualizations in general (Section 19.2), and to the use of small-multiples visualizations for dynamic data (Section 19.3) is presented.

19.1 Visualization Approaches for Analyzing Multiple Relations Between Software Artifacts

There exist several visualization approaches for analyzing multiple relations between software artifacts based on node-link and matrix representations of graphs. The closest to our approach are the following representations. First of all, Gutwenger et al. [61], Eichelberger [47], and Eiglsperger et al. [49, 48]—among others—presented optimized layouts of UML diagrams showing inheritance and, e.g., aggregation at the same time. However, their approaches are limited to a small number of specific relation types. Our approach allows users to select those relation types that they find most useful for their analysis. Moreover, their constraints on the layout focus on the inheritance relationship leaving less degrees of freedom for other relations. Our approach puts focus on each relation type separately.

The Software Architecture Visualization and Evaluation (SAVE) tool [78] shows different relationships between classes. The relationships can be bundled showing only one edge or unbundled showing all different types of relationships, one per edge. In both cases, the edges are considered as single edge when computing the layout using one of the available algorithms. In contrast, our approach separates the different edge types, which allows to optimize the layout and to discover structures, e.g., cycles, in one relation graph that are not present in others and thus might be hidden, if all relations are combined in one graph.

The parallel node link visualization (PNLV) introduced by Beck et al. [26] uses parallel axes and links between these axes. Each axis represents the classes in the same order. Each pair of axis is connected by links of one specific type. For example, the links between the first two axes could show the call relationships, while the links between the second pair of axes could display code clones. Abuthawabeh and Zeckzer [9] introduced an interactive multi-matrix visualization (IMMV) for showing up to nine different edge types. Each row and column of the matrix represents one class. Each cell in the matrix is subdivided into 9 sub-cells. These sub-cells show one relation each. Additionally, each edge type has its own color. An evaluation was performed to assess which structures can be found using PNLV and IMMV [8]. It showed, that mostly local structures are found, e.g., nodes with high in- or out-degree, that is, classes that are connected to many other classes. Our approach on the other hand reveals global structures that are difficult to find with the previous two approaches due to needing to follow links.

Other node-link based visualizations include the approach of Erdemir et al. [52] that uses a force-directed layout with a focus on metrics. Juxtaposition of nodes connected by links is also used by the approach of Ducasse and Lanza [46]. Both approaches have a different focus compared to ours.

Matrix-based Visualizations were also used by Beck and Diehl [30] who compared package structures, and by van Ham [53] who focused on call graphs. Both approaches focus on one specific relation, only, while we consider several relations simultaneously.

Laval et al. [73, 74] used detailed views in a matrix representation of dependencies between packages. These detailed views are small-multiples arranged inside the matrix. They show summary statistics of the relations between the classes of the two packages whose relation is represented by the cell using a node-link and a bar representation. However, they show all relations between these two packages in one graph, while we split the relations into several graphs.

Other approaches using small-multiples for displaying graphs can be found in other application areas. Tominski et al. [93] provided an interactive system for coordinated graph visualization combining different views. Bremm et al. [36] used small-multiples to compare several trees simultaneously. Their goal was to identify global and local structure similarities focusing on biological applications, while we focus on software engineering tasks.

We describe UML, SAVE, and the approaches of Beck et al. and Abuthawabeh and Zeckzer as representatives of the different methods in more detail in Section 21.

19.2 Small-Multiples Visualizations

In 1983, Tufte [94] described small-multiples and showed some examples. In 1990, he mentioned [95], that *"At the heart of quantitative reasoning is a single question: Compared to what? Small multiple designs, multivariate and data bountiful, answer directly by visually enforcing comparisons of changes, of the differences among objects, of the scope of alternatives. For a wide range of problems in data presentation, small multiples are the best design solution."* In 2006 [96], he states, that *"comparison must take place within the eyespan"*.

Baldonado et al. [99] proposed guidelines when using multiple views over single views is helpful. Gleicher et al. [59] proposed a taxonomy where small-multiples are considered as a type of juxtaposition. Javed and Elmqvist [63] explored the design space of composite visualization and show five visual design patters. Kehrer et al. [68] compared categories of small-multiples using a formal model.

Van den Elzenet et al. [97] proposed analyzing multi-dimensional data using small-multiples for selecting interesting dimensions that were then shown as one large single. Switching between small-multiples and large singles allowed to interactively explore the data.

Stasko et al. [85] described Jigsaw, a system for visualizing and analyzing connections between document entities using multiple coordinated views.

Cottrell et al. [41] introduced Guido, a tool showing the commonalities and differences between different source codes using multiple coordinated views. Reniers et al. [81] described the Solid* Toolset for program comprehension taking advantage of using coordinated multiple views.

Finally, in 2007, Roberts [82] explored the state of art of coordinated multiple views for exploratory visualization.

19.3 Small-Multiples Visualizations for Analyzing Dynamic Data

To visualize dynamic data, several kinds of visualizations such as graph layouts and flow maps were combined using small-multiples. Fuchs et al. [55] conducted a comparative study to assess the performance of different glyphs representing time series which are embedded in small-multiples.

Archambault et al. [15] compared the influence of preserving the mental map of dynamic graphs while using animation and small-multiples. They found, that small-multiples provide faster performance, while animation is more accurate in determining node or edge tasks in a specific snapshot of a graph. Later, Archambault and Purchase [14] extended the previous study investigating the relation between mental map preservation and memorability of evolving graph sequences. They found, that considering mental map preservation aids mental comparisons and visualizing differences in several data views. Moreover, preserving the mental map is not always helpful with respect to performance (response time and errors rate), while qualitatively it facilitates memorability based on user perception. Burch and Weiskopf [39] use small-multiples showing sequences of dynamic graphs combined with flip-book interaction as a kind of animation to increase scalability.

Chapter 20

Small-Multiples Node-Link Visualization for Displaying Several Relation Types

20.1 Visualization

20.1.1 Small-Multiples Node-Link Visualization

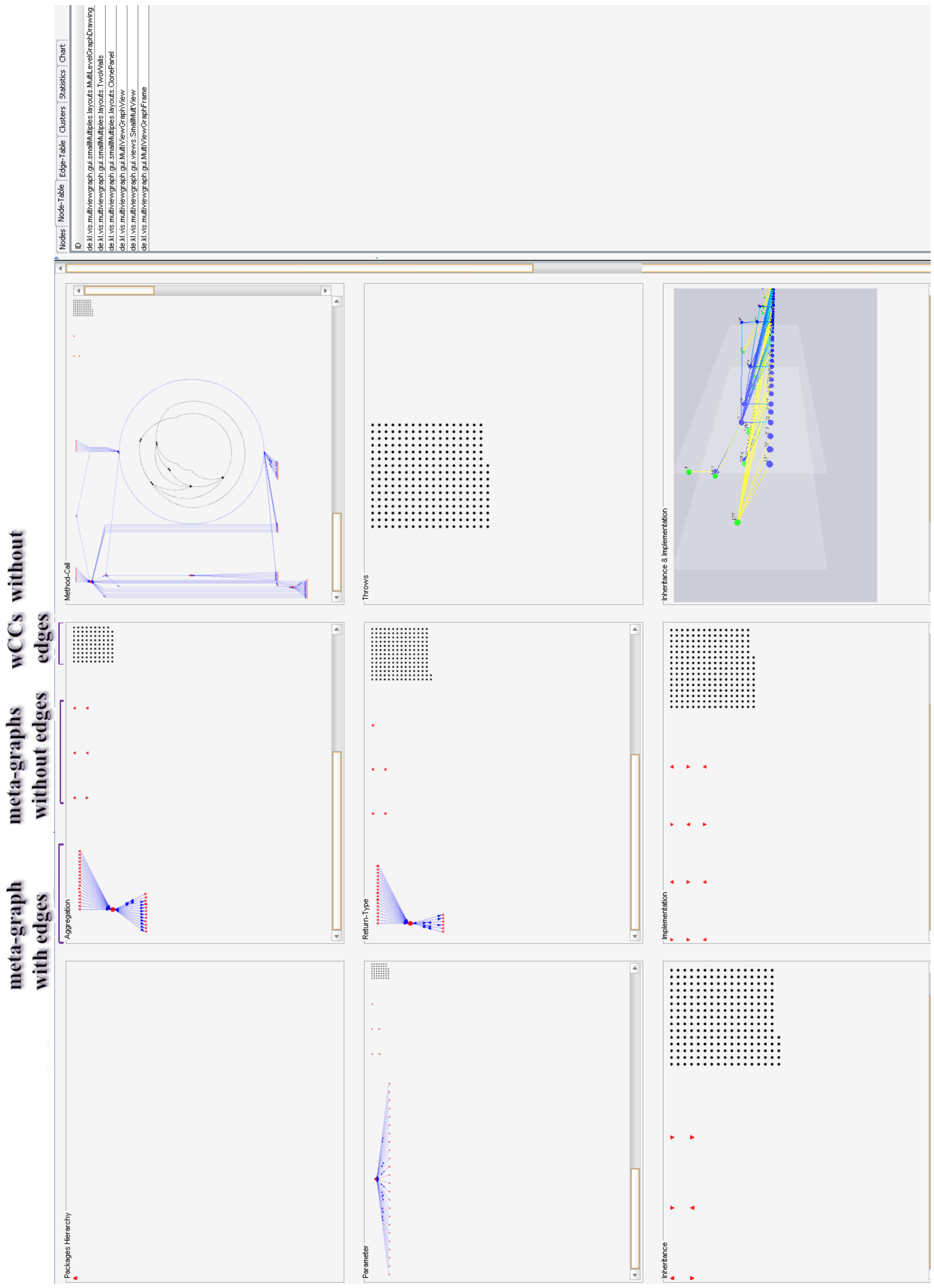
The principal idea of the small-multiples node-link visualization (SMNLV) approach for displaying several relation types is to subdivide the available screen space (Fig. 20.1). An initial subdivision into 3×3 tiles is chosen displaying up to 9 different types of relations. This is motivated by the paper of Abuthawabeh and Zeckzer [9], where the same subdivision was used. In each tile, a graph representing the structure for one relation type is drawn using the optimized topological layout for directed graphs introduced by Abuthawabeh and Zeckzer [10]. Table 20.1 shows the arrangement of the tiles, while Table 20.2 lists the entities that are mapped to nodes and edges for each of these graphs. An exception is the last tile. Here, a combination of the inheritance and implementation relations is shown using a 2.5D approach. Using two planes in a 3D space, one plane represents the inheritance of the classes while the other represents the inheritance of the interfaces. Links from the classes to the interfaces represent the implementation relationship.

On a standard full HD display, the available space for each of the graphs becomes quite small. However, the approach easily scales to tiled displays, using one complete full HD display for each of the tiles. Moreover, each tile can be displayed on a second screen in an enlarged version (Section 20.2). Please note, that using additional displays is only an optional extension for SMNLV.

An information view can be displayed on demand on the right side of the small-multiples view. Further, the source code of a selected class can be opened in an editor for further inspections.

package structure	aggregation	method-call
parameter	return-type	throws
inheritance	implementation	inheritance & implementation

Table 20.1: Arrangement of the graphs



ID
de.kl.vis.multiviewgraph.gui.smallMultiples.layouts.MultiLevelGraphDrawing
de.kl.vis.multiviewgraph.gui.smallMultiples.layouts.TwoWalls
de.kl.vis.multiviewgraph.gui.smallMultiples.layouts.ClonePanel
de.kl.vis.multiviewgraph.gui.MultiViewGraphView
de.kl.vis.multiviewgraph.gui.views.SmallMultView
de.kl.vis.multiviewgraph.gui.MultiViewGraphFrame

Figure 20.1: Small-Multiples Node-Link Visualization of SMNLV

relationship type	graph type	nodes	edges
package structure	tree	packages & classes & interfaces	package A belongs to package B & class/interface C belongs to a package D
aggregation	directed graph	classes	class A uses class B as type for class variable
method-call	directed graph	classes	methods of class A call methods of class B
parameter	directed graph	classes	methods of class A use class B as type of parameter
return-type	directed graph	classes	methods of class A use class B as return type
throws	bipartite directed graph	classes	methods of class A throw exceptions of type class B
inheritance	tree	classes	class A inherits from (extends) class B
implementation	bipartite directed graph	classes & interfaces	class A implements interface B
inheritance & implementation	tree & bipartite directed graph	classes & interfaces	implementation & inheritance

Table 20.2: Mapping of software entities to graph objects; A: source, B: destination of an edge

20.1.2 Graph Layout

The graph of each tile (except the last one) is drawn using the optimized topological layout for directed graphs proposed by Abuthawabeh and Zeckzer [10]. The process is outlined in Fig. 20.2. First, the graph is decomposed into weakly connected components (*wCC*). Each of the wCCs is then decomposed into non-trivial cyclic sub-graphs, trees, and DAGs. Each of the sub-graphs is represented by a meta-node. Computing meta-edges between these meta-nodes from the edges of the original graph yields a meta-graph. This two-level hierarchy is then used for drawing each wCC.

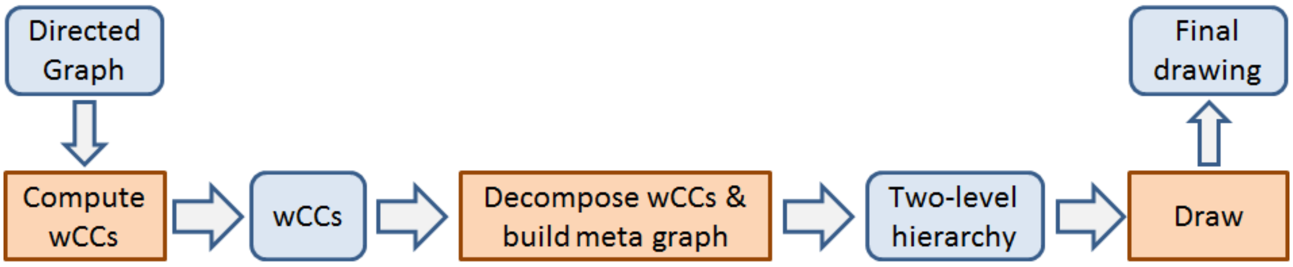


Figure 20.2: Process of the topological visualization of directed graphs [10]

Nodes of the original graph are represented by black filled circles. Edges of the original graph are drawn in black. Edges in trees and DAGs are represented by lines with arrows showing the direction. If two edges with reverse direction exist between two nodes, a line with two arrows is used. The edges of the non-trivial cyclic sub-graphs are represented by curves with arrows showing the direction. If two edges with reverse direction exist between two nodes, a curve with two arrows is used.

Unfolded meta-nodes are represented using blue rectangles for trees and DAGs and blue circles for non-trivial cyclic sub-graphs surrounding the respective sub-graph. Folded meta-

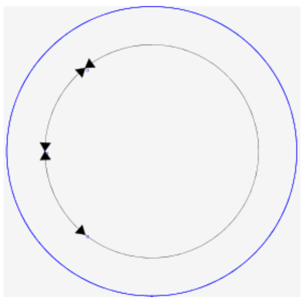
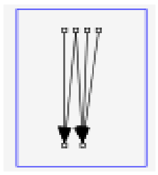
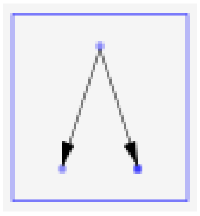
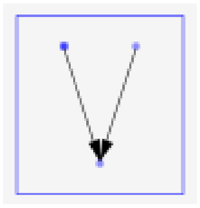
Sub-graph type	Unfolded Meta-Node	Folded Meta-Node
non-trivial cyclic sub-graph		●
DAG		▬
Down Tree		▲
Up Tree		▼

Table 20.3: Mapping of sub-graph types to meta-nodes

nodes are represented by small red circles for folded non-trivial cyclic sub-graphs, by small red triangles for folded non-trivial trees, and by small red rectangles for folded non-trivial DAGs (Table 20.3). For down-trees, the base of the triangle is down, while for up-trees, the base of the triangle is up. All meta-edges are represented by blue color. Besides one-directional and double meta-edges, two more meta-edge types are used to convey the connection between the sub-graphs [10]. Double path edges ($\hookleftarrow\!\!\rightarrow$) show, that there is an outgoing connection to one and an incoming connection from another component. Blocking edges ($\rightarrow\leftarrow$) represent connections, where an outgoing edge is followed by an incoming edge in a path or vice versa.

For the final drawing, it is further distinguished, if the wCC or the meta-graph contain edges or not. All wCCs that do not contain edges (only one node) are arranged in a matrix. Similarly, all meta-graphs that do not contain edges are arranged in a matrix, too. Then, horizontally from left to right, meta-graphs with edges, the matrix of meta-graphs without edges, and the matrix of wCCs without edges are drawn. Drawing all one node wCCs (meta-graphs) in a matrix allows for a compact representation of the graph. The annotated tile (top-middle) of Fig. 20.1 illustrates an example of the final drawing of the aggregation relations.

The drawing of each meta-graph and of each sub-graph uses optimal drawing algorithms depending on the type of the (sub-)graph: trees are drawn using the improved Walker's algorithm [37], directed acyclic graphs (DAGs) are drawn using the Sugiyama algorithm [87], and non-trivial cyclic sub-graphs are drawn using Bachmaier's algorithm [22, 20, 21]. Further, all algorithms for the meta-graph are area-aware.

If the complete graph is drawn, nodes that belong to two or more sub-graphs are cloned and placed in each of the sub-graphs [10]. Cyan lines are used to connect the clones.

To avoid clutter, no information is displayed next to the nodes. Interaction (Section 20.2) is provided to select nodes and to display additional information for each node.

20.1.3 Information View

An information view to the right of the small-multiples node-link visualization shows context information about nodes and meta-nodes. It either shows the information associated to a node of the original graph, or it shows the information about all nodes contained in the sub-graph of a meta-node in a table (Fig. 20.1, right side).

20.2 Interaction

A video supporting the explanations of the interaction capabilities can be provided upon request.

20.2.1 Large view

Each tile of the small-multiples view can be displayed using a separate window, e.g., full screen on an additional display, by clicking the right button of the mouse.

20.2.2 Zooming and Panning

Each view supports standard zooming and panning functionality to adjust the view to the focus of the analysis.

20.2.3 Highlighting

Selecting a node of the original graph in any tile highlights it by drawing a red rectangle around it in all tiles of the small-multiples view. If the node is contained in a folded meta-node (hidden), the meta-node will be highlighted. Additionally, the full name of the node will be shown in the information view. Please note, that cloned nodes (shared nodes between two topological sub-graphs) will also be highlighted. Clicking in the free area, deselects the nodes and removes the highlighting. Selecting a meta-node of the meta-graph highlights it by drawing a red rectangle around it only in its tile. Additionally, the information about all nodes of the sub-graph associated to this meta-node is listed in a table in the information view.

20.2.4 Folding and unfolding

The user can display the lowest level of the two-level hierarchy—the original graph—and the meta-graph. Additionally, each meta-node can be opened or closed individually, allowing to reduce large structures to a single node while examining details in other sub-graphs.

Each meta-node can be folded by double clicking inside the blue boundary surrounding it. Then, the enclosed sub-graph disappears and the meta-node is displayed as small red circle for folded non-trivial cyclic sub-graphs, as small red triangle for folded non-trivial trees, and as small red rectangle for folded non-trivial DAGs (Table 20.3). The meta-node can be unfolded by double clicking the red circle, triangle, or rectangle.

20.2.5 Opening source code

Showing the source code file of any class is possible by <shift>+<left mouse button> click on the node representing the class. Currently, Notepad++ [45] is used to present the source code.

20.3 Implementation

The relationships were extracted from the class files using jclassinfo [64]. The resulting information was post-processed using customized UNIX scripts to obtain the complete graph (classes, interfaces, and their relationships) in graphml format. The package hierarchy is computed from the qualified class and interface names by the tool itself. The tool is implemented using Java 8 using the Gravisto Toolkit [23] for drawing non-trivial cyclic sub-graphs, the abego TreeLayout [6] for drawing trees, and the NetBeans Visual Library [79] (undocumented Hierarchical Layout) for drawing DAGs.

Chapter 21

Other Approaches to Display Several Relation Types

In this section, four alternative approaches for displaying multiple relation types simultaneously are described, representing one category, each.

21.1 UML-Style

An approach for drawing optimized UML diagrams was presented by Gutwenger et al. [61]. Using the so-called GoVisual layout of the diagram, classes are represented by boxes that contain the class description according to the UML specification. The edge types shown are inheritance in class hierarchies (generalizations) and associations (e.g., aggregation and composition). Similar approaches include the ones by Eichelberger [47] and Eiglsperger et al. [49, 48]. However, all approaches are limited to the relations shown in UML diagrams.

21.2 SAVE

In the Software Architecture Visualization and Evaluation (SAVE) tool [78], classes and interfaces are represented by boxes containing annotations, metrics, and additional information. Bundled edges are drawn as straight lines and represent all relations between two classes or interfaces. This allows using available layout algorithms as long as the size of the boxes is taken into account. The edges can be unbundled. Unbundling does not change the overall layout, but splits each bundled edge into the original edges that are then represented by one polyline per edge type. Color is used to distinguish between different edge types. A hierarchical layout is used for drawing the graph. Manual post-processing of the layout is provided to enhance the initial automatic drawing.

21.3 PNLV

In the parallel node-link visualization (PNLV) approach [26], the nodes of the graph represent classes and interfaces of a software system. Fig. 22.23 represents eight relations for Checkstyle 6.5 [3]. The nodes are mapped to boxes that are laid out above each other in the first column. PNLV uses icicle plots to show the package structure of the classes on the left of the classes and interfaces. The nodes are arranged according to the associated packages at the leaf level of the icicle plot. For each edge type except the package hierarchy, one additional copy of the column of node boxes is added to the right. Edges originate from the classes and interfaces in the left column and end at classes and interfaces in the respective right column.

21.4 IMMV

In the interactive multi-matrix view (IMMV) approach [9], an adjacency matrix is used to represent graphs having multiple edges with different types. Fig. 22.21 represents eight relations for Checkstyle 6.5 [3]. Similar to the PNLV approach, icicle plots are used for representing the package structure. A cell represents the relations between the classes and interfaces of the row to the classes and interfaces of the column. Each cell is subdivided into nine tiles. The remaining seven relations are mapped to one tile of each cell and additionally color coded.

Chapter 22

Results

In Section 22.1, we show the structure of SMNLV using IMMV and SMNLV itself. Then, we show the structure of JUnit 4.12 [4], JFtp 1.59 [2], and Stripes 1.5.7 [1] using IMMV and SMNLV in Section 22.2, Section 22.3, and Section 22.4, respectively. The relations provided are package containment, aggregation, method-call, parameter, return-type, throws, inheritance, and implementation. As standard UML layouts do not consider all relations, they were not included in the comparison. The SAVE tool is not publicly available and therefore was also not included.

Additionally, in Section 22.5, we show the structure of Checkstyle 6.5 [3] using three different approaches: IMMV (Section 21.4, Fig. 22.21), PNLV (Section 21.3, Fig. 22.23), and SMNLV (Section 20, Fig. 22.24). The statistics of SMNLV, JUnit, JFtp, Stripes, and Checkstyle are shown in Table 22.1.

relationship type	N	E	# wCC	# single nodes	# cycles	# trees	# DAGs
package structure	314	313	1			1	
aggregation	282	246	96	89	1	35	1
method-call	282	595	98	94	4	59	4
parameter	282	378	60	54	1	27	2
return-type	282	136	190	184	1	26	3
throws	282		282	282			
inheritance	282	31	251	245		6	
implementation	282	35	247	235		12	
total	282	1421	1224	1183	7	165	10
package structure	286	285	1			1	
aggregation	258	109	165	151		11	3
method-call	258	306	64	59	1	5	2
parameter	258	297	89	84		4	1
return-type	258	87	185	180		3	2
throws	258	31	227	220		6	1
inheritance	258	90	168	147		21	
implementation	258	33	227	216		10	1
total	258	953	1125	1057	1	60	10
package structure	181	180	1			1	
aggregation	162	146	64	54	1	9	1
method-call	162	376	32	30	1	10	1
parameter	162	69	95	83		10	2
return-type	162	24	143	138	1	8	1
throws	162	0	162	162			
inheritance	162	37	125	120		5	
implementation	162	20	143	138		3	2
total	162	672	764	725	3	45	7
package structure	325	324	1			1	
aggregation	301	177	154	147		5	2
method-call	301	620	81	76	3	9	1
parameter	301	155	185	173		7	5
return-type	301	128	201	188		9	4
throws	301	34	268	261		6	1
inheritance	301	79	222	206		16	
implementation	301	71	230	204		25	1
total	301	1264	1341	1255	3	77	14
package structure	499	498	1			1	
aggregation	473	116	369	346		16	7
method-call	473	763	76	71	1	5	2
parameter	473	581	91	75		13	3
return-type	473	826	301	289	2	7	5
throws	473	11	462	461		1	
inheritance	473	235	238	229		9	
implementation	473	32	441	422		19	
total	473	2564	1677	1893	3	70	17

Table 22.1: Extracted information from and decomposition of SMNLV, JUnit 4.12 [4], JFtp 1.59 [2], Stripes 1.5.7 [1], and Checkstyle 6.5 [3] shown from top to bottom, respectively with N: number of nodes and E: number of edges

22.1 SMNLV

For SMNLV, the results using IMMV and SMNLV itself are presented.

Fig. 22.1 shows the relations of SMNLV using IMMV revealing distinct visual structures such as the small stripes along the diagonal or the off-diagonal clusters on the right. The upper left corner shows green dots, red dots, and blue dots representing method-call relations, parameter relations, and aggregation relations shaping one vertical stripe. This shows that methods of many different classes call the methods of only one class, use the same class as parameter type, and use the same class as type for class variable. Also, a vertical line is parallel to the previous stripe, that is formed by only green dots. Similar stripes appear in the middle along the diagonal. Further, the purple dots compose a vertical line in the middle representing the inheritance relations. This shows that several classes inherits from (extends) only one class. The brown dots along diagonal represent the return-type relations. This shows that several classes are used as return type by their methods. At right side, relatively many dots having different colors form an off-diagonal cluster representing mixture of relations. This implies that several classes, that belong to different packages use classes associated to two packages (model and modelVis).

To assess the suitability of SMNLV for analyzing static software structure, we analyzed the structure of SMNLV by itself. SMNLV contains 282 classes (nodes) and 1421 relations (edges, without package structure, Table 22.1). The largest graph is the method-call graph containing 595 edges between 188 nodes and 94 single nodes. Starting SMNLV and loading the data set results in a visualization of each relation type showing the respective meta-graph. We focused on cyclic dependencies to assess if there are unwanted ones. We find one cyclic dependency for the aggregation relation, four for the method-call relation, one for the parameter relation, and one for the return-type relation. Being interested in the cyclic dependencies of the method-call relation, we opened it in a single, full-screen window in Fig. 22.2. Opening one of the cyclic dependencies of the method-call relation results in the visualization shown in Fig. 20.1, upper right corner. It is a small cyclic dependency with six classes and ten edges. Consulting the table in the information view revealed, that three classes belong to the layout package and that three more classes belong to the grand-parent package of the layout package. Cyclic structures over package borders are to be avoided. Thus, we had a closer look at the relations using the editing facility. We found, that the class MultiLevelGraphDrawing is calling the class MultiViewGraphView directly instead of the class SmallMultView and fixed this issue. Moreover, we found, that the cycle itself is due to reciprocal method calls of classes and that the structure of this cycle is fine.

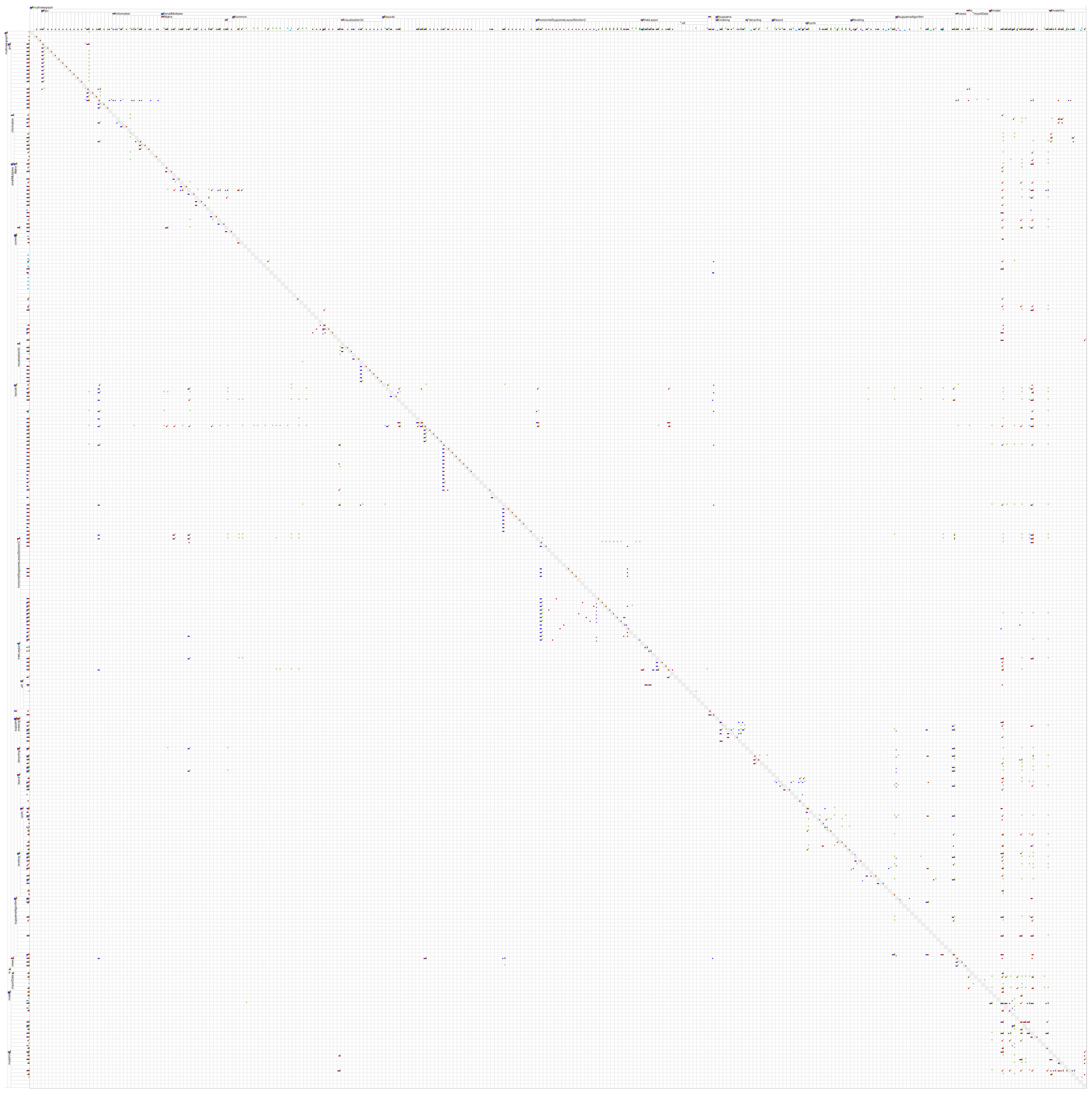


Figure 22.1: Interactive multi-matrix view (IMMV) [9] of SMNLV

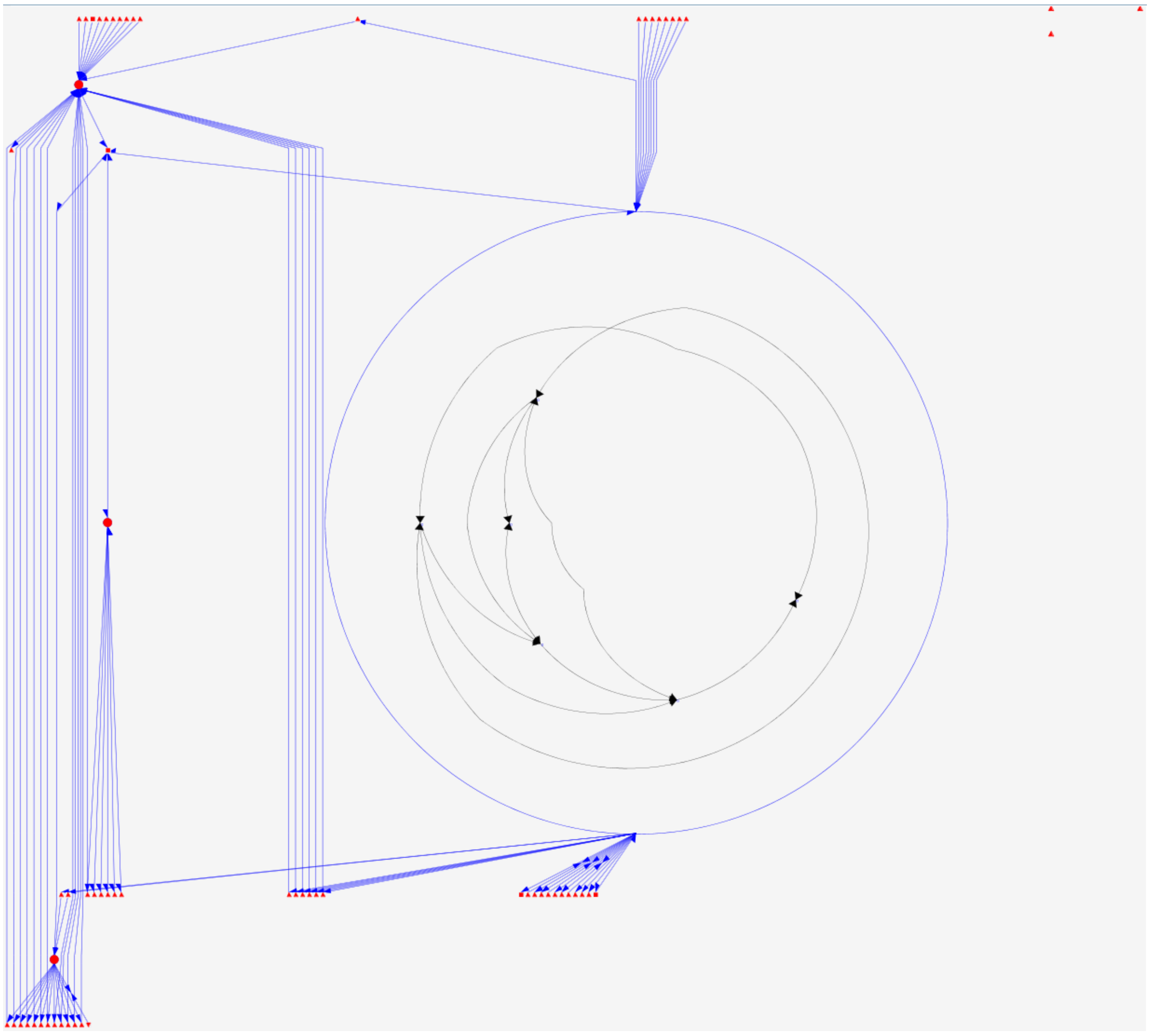


Figure 22.2: Small-Multiples Node-Link Visualization (SMNLV) of SMNLV itself: method-call relation, enlarged view, one meta-node unfolded

22.2 JUnit

For JUnit 4.12 [4], the results for IMMV and SMNLV are presented.

Fig. 22.3 displays the relations of JUnit 4.12 using IMMV showing characteristic structures like the small stripes along the diagonal or the small stripes in the left or top Icicle plots inside their last level cells. Having a close look at the square in the lower right corner reveals, that it consists of several vertical and horizontal lines structures along the diagonal (Fig. 22.4). The green dots in the middle shape a horizontal line representing the method-call relations. This shows that methods of only one class call methods of several different classes. The red dots combined by green dots form a vertical line along diagonal representing the parameter and the method-call relations. This implies that several classes use only one class as parameter type and they call the methods of the same class. Further, the red dots combined by blue dots form a vertical line along diagonal representing the parameter and the aggregation relations. This implies that several classes use only one class as parameter type and they use the same class as type for class variable. The purple dots form a vertical line along diagonal representing the inheritance relations. This shows that several classes inherits from (extends) only one class. The brown dots along diagonal represent the return-type relations. This shows that several classes are used as return type by their methods.

Fig. 22.5 shows the meta-graphs of all relations examined. JUnit 4.12 contains 258 classes (nodes) and 953 relations (edges, without package structure, Table 22.1). The largest graph is the method-call graph containing 306 edges between 199 nodes and 59 single nodes. The package hierarchy is a single down-tree. Overall, only one non-trivial cyclic sub-graph is found (in method-call). Inheritance only forms tree structures (21), five of them being unfolded. Aggregation forms mostly tree structures (11) and 3 DAG structures. The same holds for implementation with 10 trees and 1 DAG, throws with 6 trees and 1 DAG, parameter with 4 trees and 1 DAG, and return-type with 3 trees and 2 DAGs. Expanding the non-trivial cyclic sub-graph of the meta-graph of the method-call relation shows, that it contains four classes and one double edge. Also, one tree and one DAG are unfolded in this weakly connected component, showing simple structures. The software engineer can accept the non-trivial cyclic sub-graph or try to resolve the cyclic dependencies.

Opening the DAG of the second trivial meta-graph of the return-type relation shows, that the DAG has four levels, and that three classes in levels three and four are dominantly used as return-type by the methods of many classes, while one class in level one targets many classes in level two.

Unfolding the DAG of the first trivial meta-graph of the throw relation shows, that the DAG has only two levels and that three classes are used by all other classes.

Unfolding the DAG of the first trivial meta-graph of the parameter relation shows, that the DAG is rather deep ending in only two classes with several cross relations between the big sub-DAGs on the left and on the right. This needs investigations from the software engineer to check why these classes are tightly coupled and if they belong to the same package or few packages, or not.

Unfolding the up-trees of the first five trivial meta-graphs of the inheritance relation shows, that the first four trees are shallow with one or two levels of inheritance while the last tree is relatively deep with up to three levels of inheritance.

Finally, unfolding the only DAG of the weakly connected components of the implementation relation graph (the first trivial meta-graph) shows, that two interfaces on the lower level were implemented by all three classes on upper level. The software engineer might be interested to investigate whether combining both interfaces into one interface enhances the design or not.

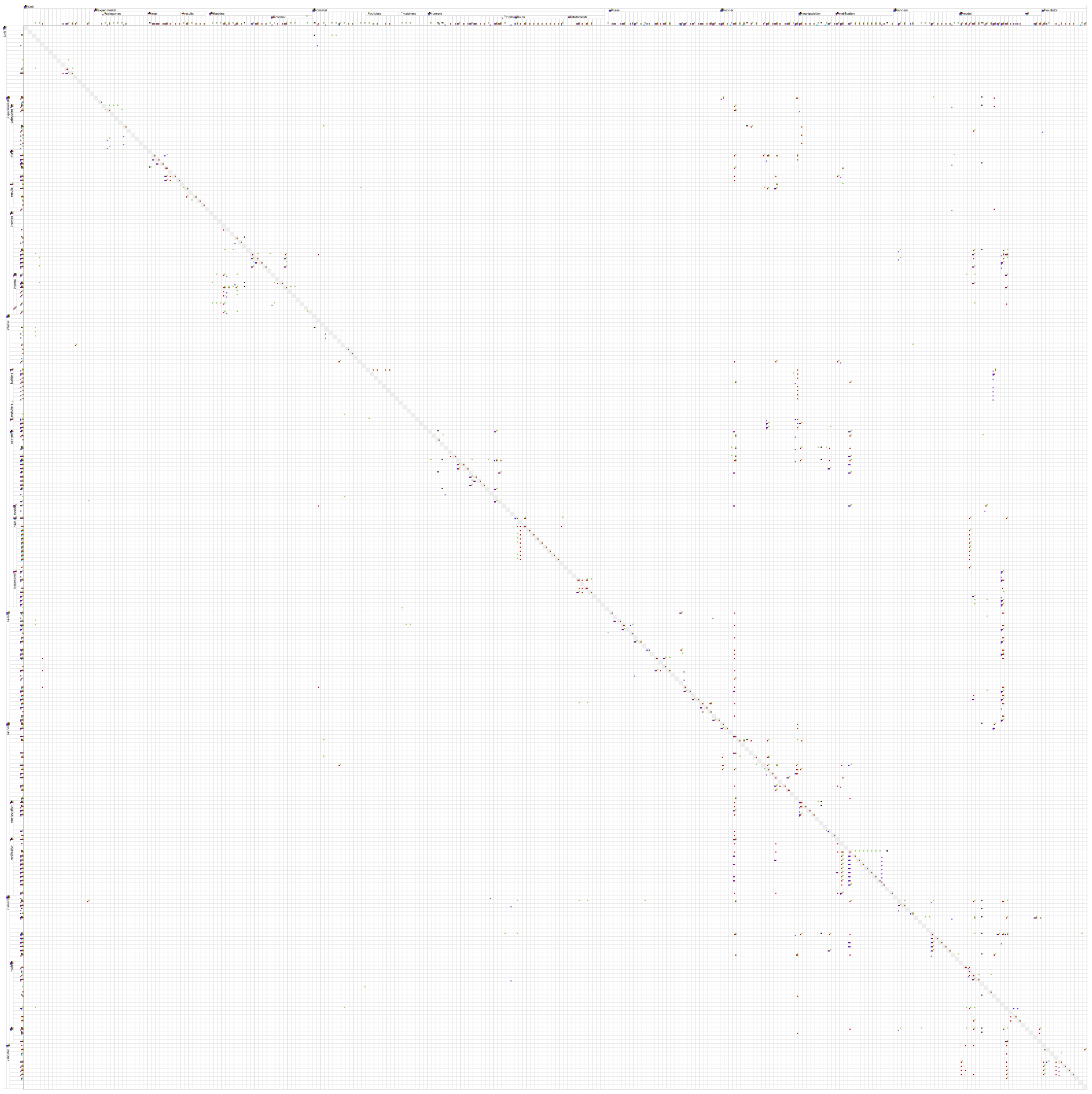


Figure 22.3: Interactive multi-matrix view (IMMV) [9] of JUnit 4.12

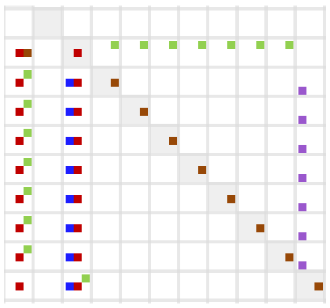


Figure 22.4: IMMV [9] of JUnit 4.12: small square in the lower right corner (close view)

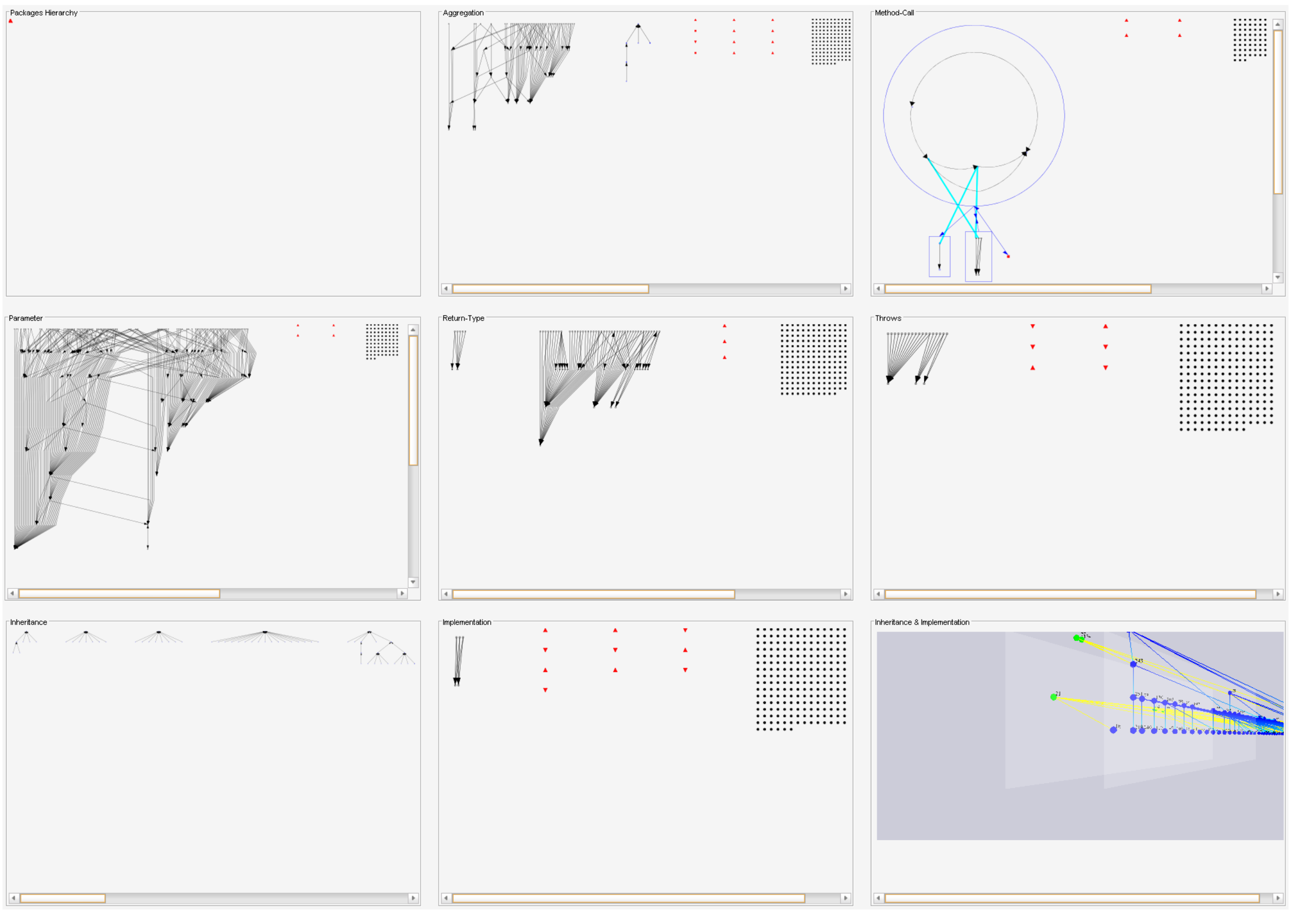


Figure 22.5: Small-multiples node-link visualization (SMNLV) of JUnit 4.12

22.3 Jftp

For JFtp 1.59 [2], the results for IMMV and SMNLV are presented.

Fig. 22.6 shows the relations of JFtp using IMMV distinguishing visual structures such as off-diagonal clusters or vertical stripes. In the middle, the green dots, red dots, and blue dots represent method-call relations, parameter relations, and aggregation relations as one combination forming a single line along diagonal. This shows that methods of many different classes call the methods of only one class, use the same class as parameter type, and use the same class as type for class variable. Further, the green dots and blue dots form an off-diagonal cluster in the middle as one combination. This shows that methods of many different classes call the methods of several classes and several classes use several classes as type for class variable. The purple dots form a vertical line in the middle representing the inheritance relations. This shows that several classes inherits from (extends) only one class.

Fig. 22.7 shows the meta-graphs of all relations examined. JFtp 1.59 contains 162 classes (nodes) and 672 relations (edges, without package structure, Table 22.1). The largest graph is the method-call graph containing 376 edges between 132 nodes and 30 single nodes. The package hierarchy is a single down-tree. None of the weakly connected components of the throws, the parameter, the inheritance, and the implementation relationships contains cycles. However, the aggregation, the method-call, and the return-type relations have one, respectively three weakly connected components that contain cyclic sub-graphs.

Having a close look at the type of the weakly connected components, we find that inheritance relation only forms tree structures (5). The implementation relation forms mostly tree structures (3) and 2 DAG structures. The same holds for the parameter relation forms mostly tree structures (10) and 2 DAG structures.

Opening the non-trivial cyclic sub-graph of the method-call relation reveals, that the cycle is relatively large containing nineteen nodes (Fig. 22.8).

Opening the non-trivial cyclic sub-graph of the aggregation relation reveals, that the cycle is rather small containing three nodes (Fig. 22.9). While unfolding the DAG shows four-level DAG with many classes use several classes as type for class variable in the second and the last levels.

Further, unfolding the non-trivial cyclic sub-graph of the return-type relation reveals, that the cycle is rather small containing three nodes (Fig. 22.10). While unfolding the tree and the DAG meta-nodes shows simple structures.

Unfolding the DAG of the parameter relation (Fig. 22.11) shows four-level DAG with many different methods of several classes use several classes as parameter type.

Unfolding the two trees of the inheritance relation (Fig. 22.12) shows a four-level tree and two-level tree.

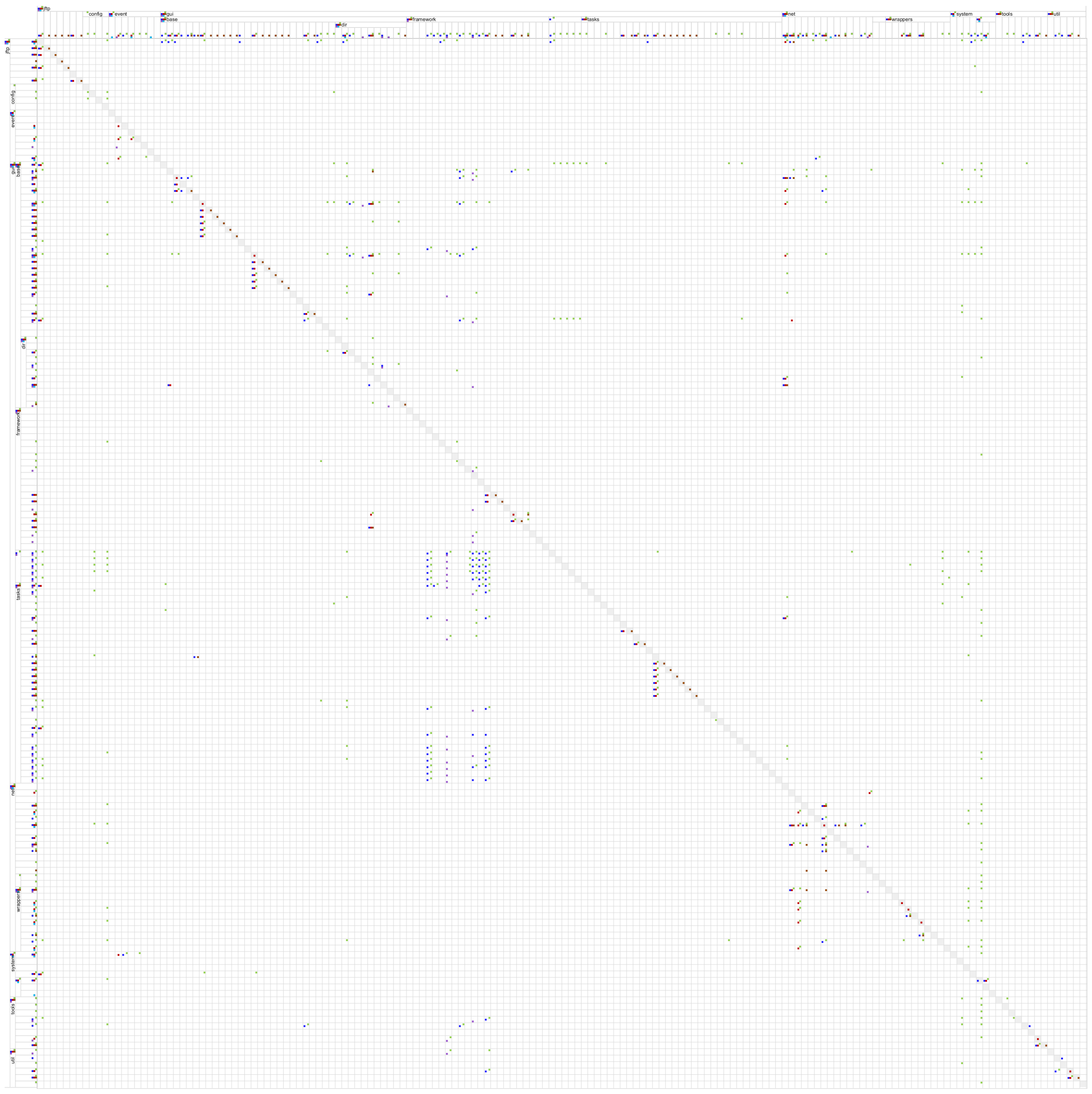


Figure 22.6: Interactive multi-matrix view (IMMV) [9] of JFtp 1.59

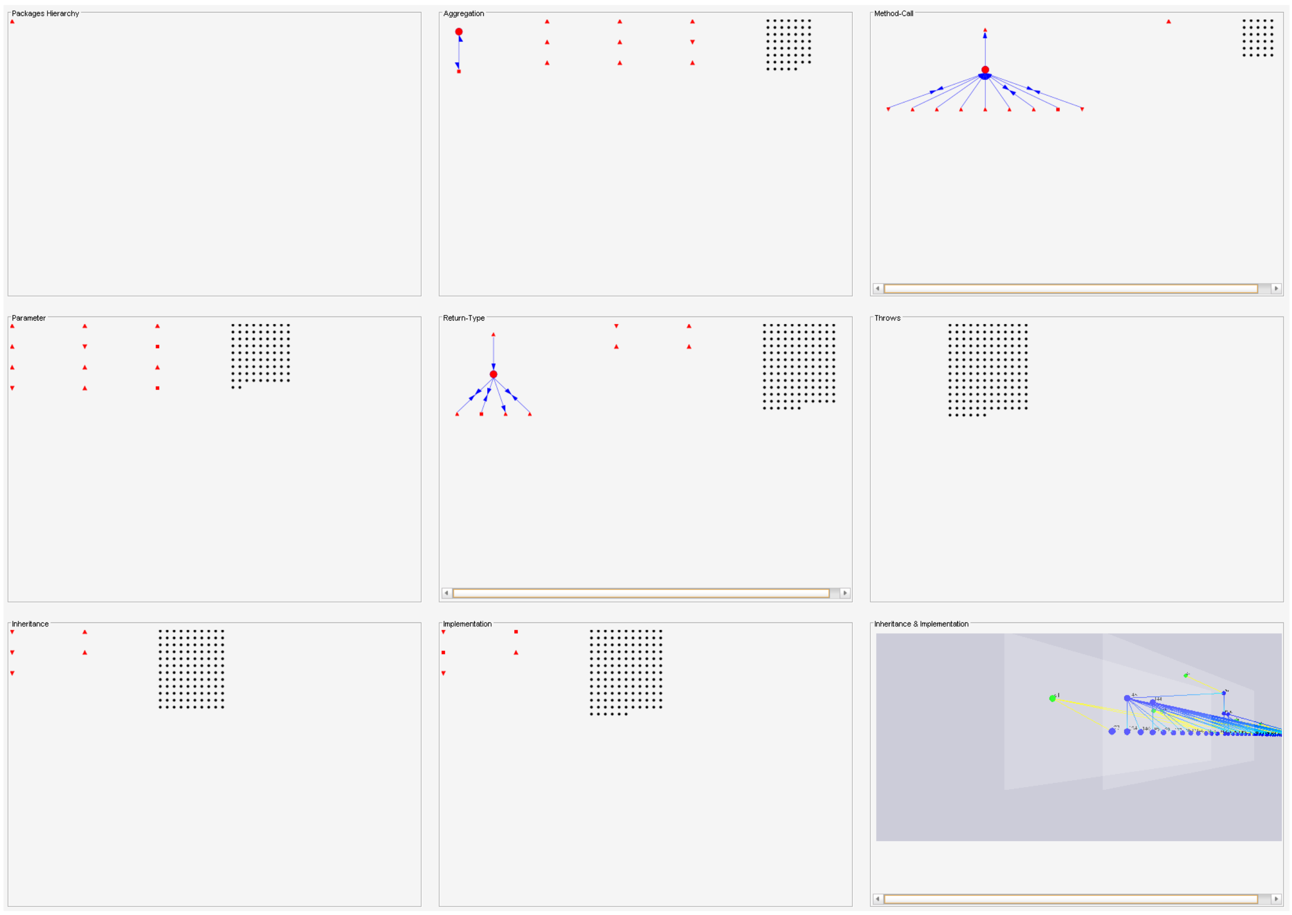


Figure 22.7: Small-multiples node-link visualization (SMNLV) of JFtp 1.59

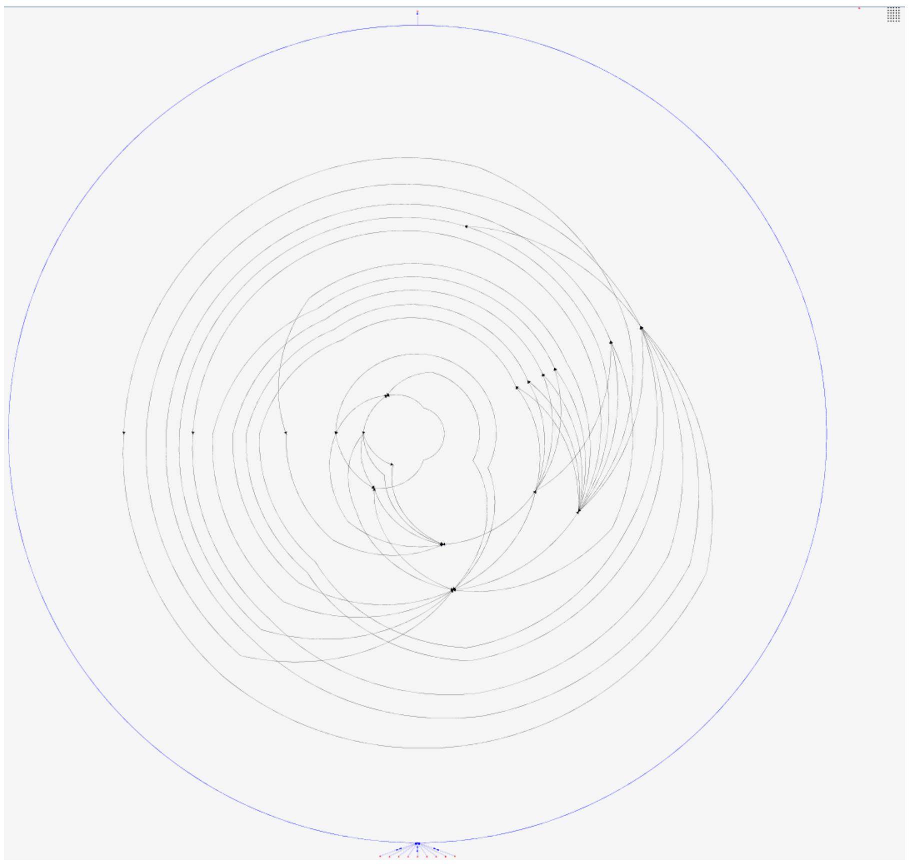


Figure 22.8: Small-multiples node-link visualization (SMNLV) of JFtp 1.59: method-call relation, enlarged view, one meta-node unfolded

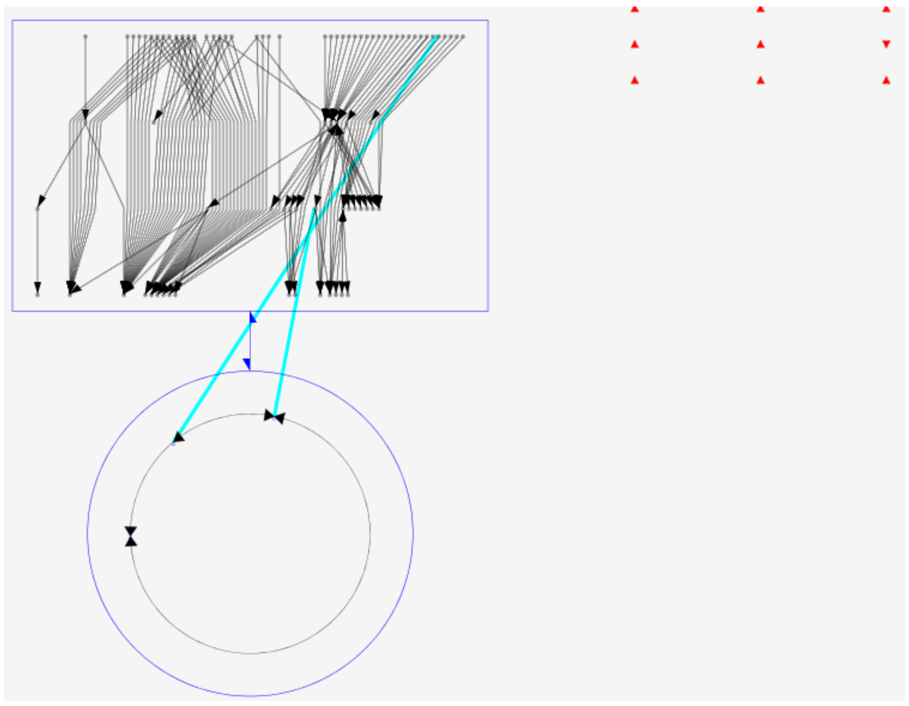


Figure 22.9: Small-multiples node-link visualization (SMNLV) of JFtp 1.59: aggregation relation, enlarged view, two meta-nodes unfolded

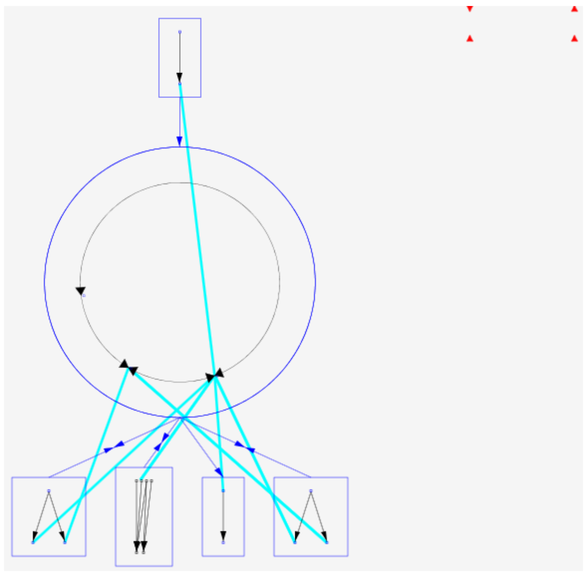


Figure 22.10: Small-multiples node-link visualization (SMNLV) of JFtp 1.59: return-type relation, enlarged view, six meta-nodes unfolded

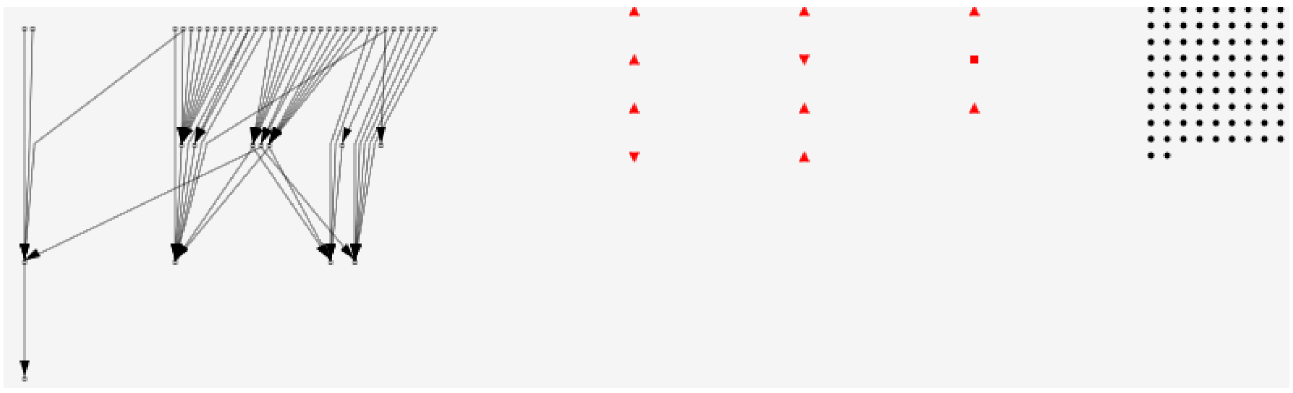


Figure 22.11: Small-multiples node-link visualization (SMNLV) of JFtp 1.59: parameter relation, enlarged view, one meta-nodes unfolded

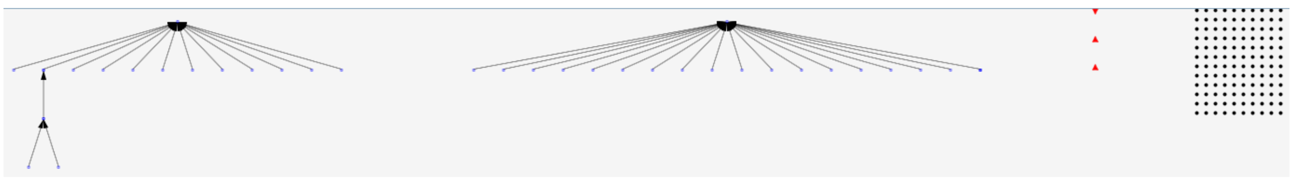


Figure 22.12: Small-multiples node-link visualization (SMNLV) of JFtp 1.59: inheritance relation, enlarged view, two meta-nodes unfolded

22.4 Stripes

For Stripes 1.5.7 [1], the results for IMMV and SMNLV are presented.

Fig. 22.13 shows the relations of Stripes 1.5.7 using IMMV discerning visual structures such as a long stripe and many short stripes. At right, a long lines shaped by a combination of green and blue dots can be easily noticed. This shows that methods of many different classes call the methods of only one class and and several classes use the same class as type for class variable. In the upper left corner (Fig. 22.14), the brown dots representing return-type relations form a short vertical line. This shows that several classes are used as return type by methods of only one class. Also, other brown dots form a line along diagonal. This shows that several classes are used as return type by methods of the same classes. Further, the green dots and red dots representing method-call relations and parameter relations, respectively form a short vertical line. This implies that methods of many different classes call the methods of only one class and the classes use the same class as parameter type.

Fig. 22.15 shows the meta-graphs of all relations examined. Stripes 1.5.7 contains 301 classes (nodes) and 1264 relations (edges, without package structure, Table 22.1). The largest graph is the method-call graph containing 620 edges between 225 nodes and 76 single nodes. The package hierarchy is a single down-tree. Overall, three non-trivial cyclic sub-graphs are found (in method-call). Inheritance only forms tree structures (16). Aggregation forms mostly tree structures (5) and 2 DAG structures. The same holds for implementation with 25 trees and 1 DAG, throws with 6 trees and 1 DAG, parameter with 7 trees and 5 DAG, and return-type with 9 trees and 4 DAGs.

Opening the three non-trivial cyclic sub-graphs of the method-call relation reveals, that the cycles are relatively small (Fig. 22.16).

Unfolding the DAG of the aggregation relation (Fig. 22.17) shows, that the DAG has five levels and that two classes (level five) are used by several classes. Also, one class (level one) use several classes (level two).

Unfolding the DAG of the parameter relation (Fig. 22.18) shows, that the DAG has six levels and that five classes are used by several classes.

Unfolding the first DAG of the return-type relation (Fig. 22.19) shows, that the DAG has four levels and that three classes are used by several classes. While unfolding the second DAG shows, that two classes use all other classes.

Unfolding the DAG of the implementation relation (Fig. 22.20) shows, that the DAG has two levels and that one class is used by several classes. While unfolding the tree shows, that several classes use one class.

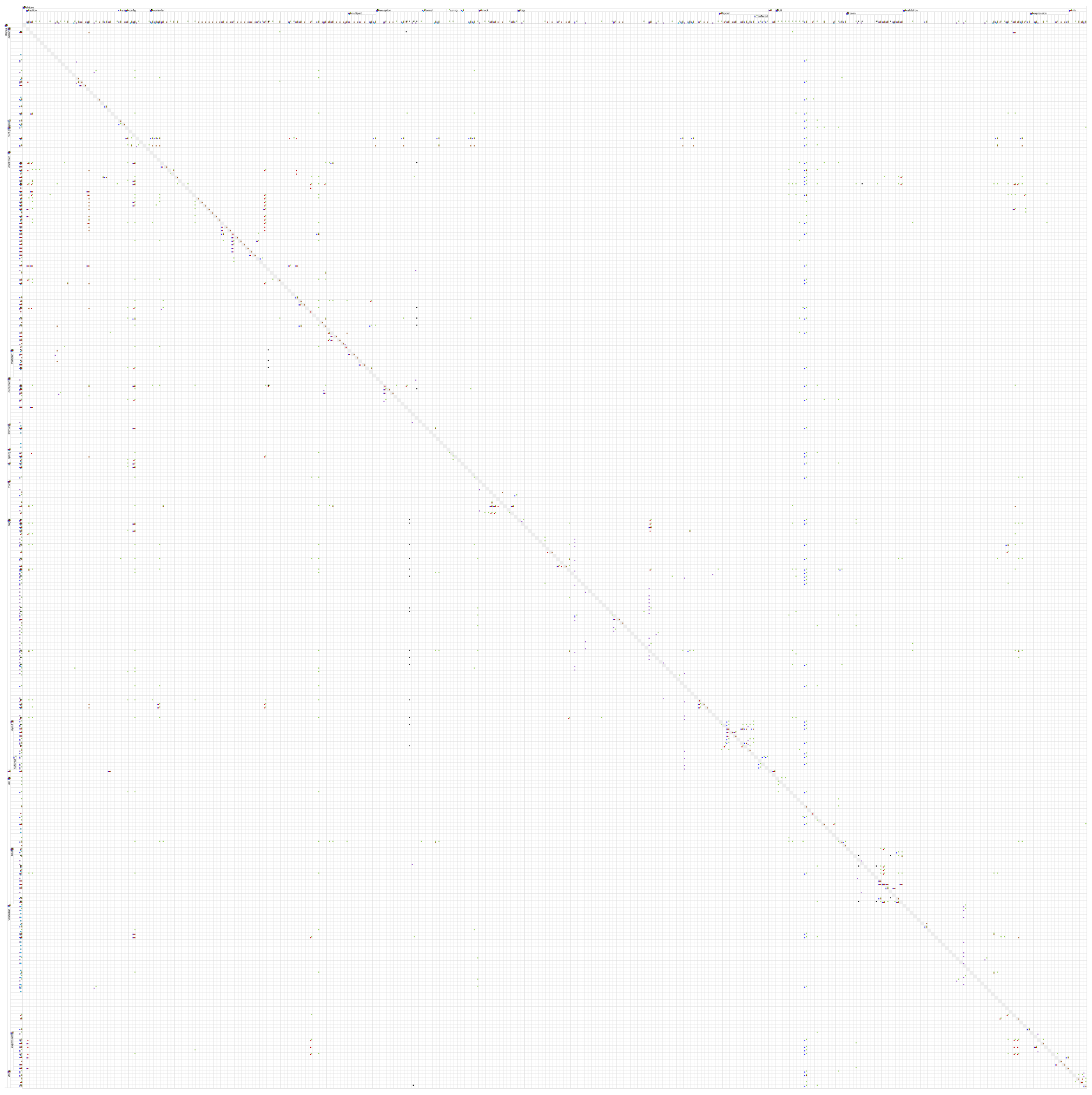


Figure 22.13: Interactive multi-matrix view (IMMV) [9] of Stripes 1.5.7

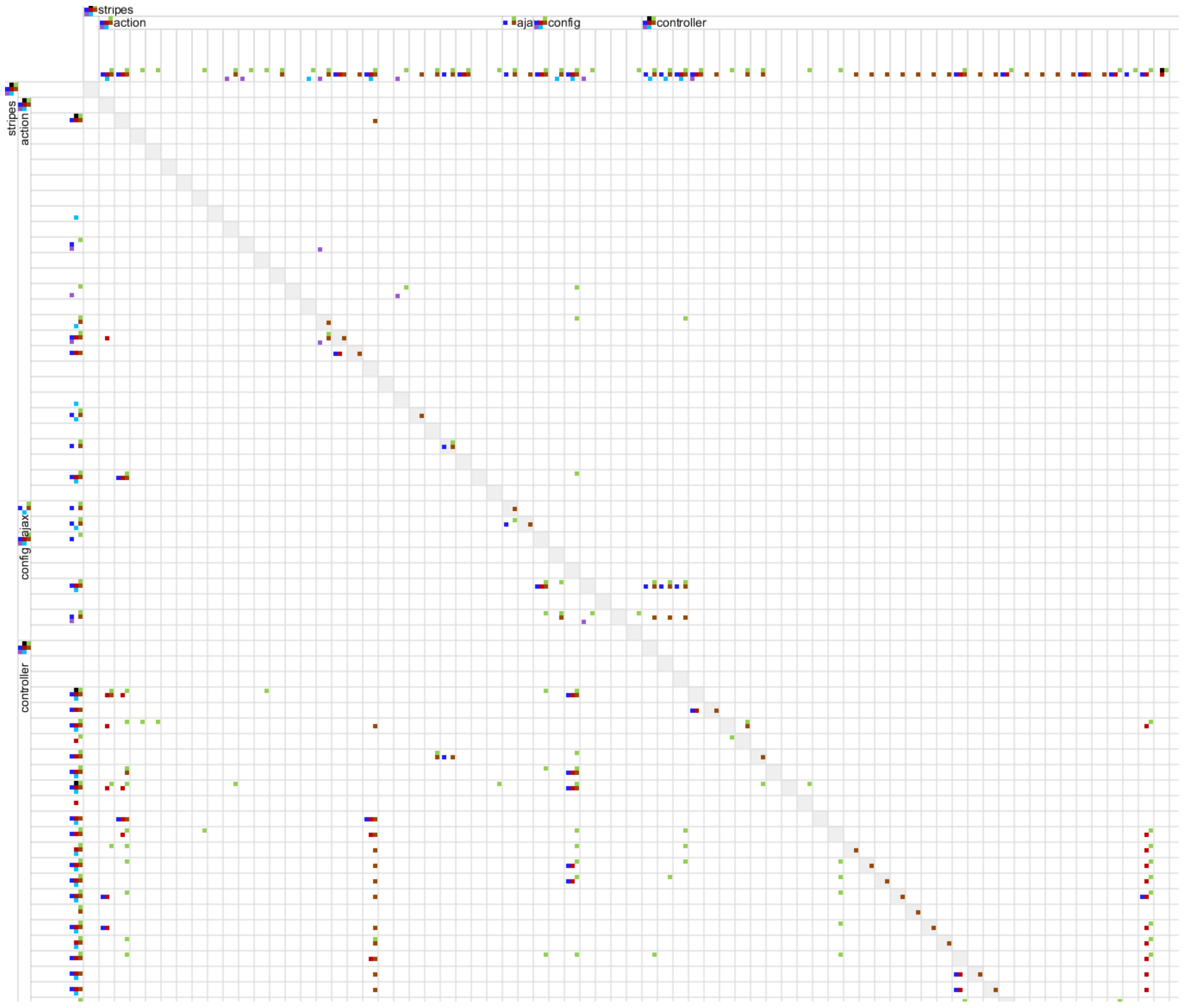


Figure 22.14: IMMV [9] of Stripes 1.5.7: square in the upper left corner (close view)

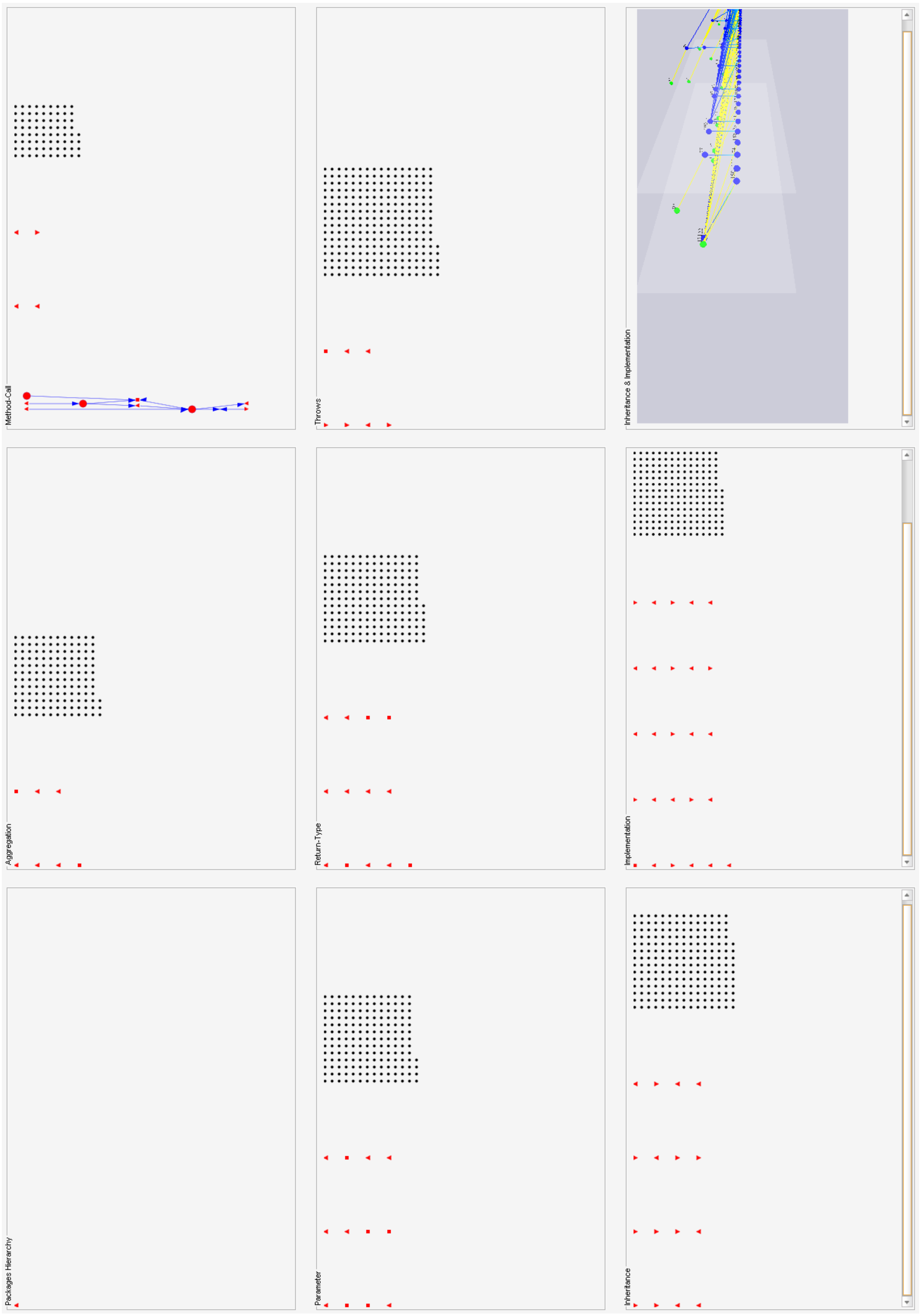


Figure 22.15: Small-multiples node-link visualization (SMNLV) of Stripes 1.5.7

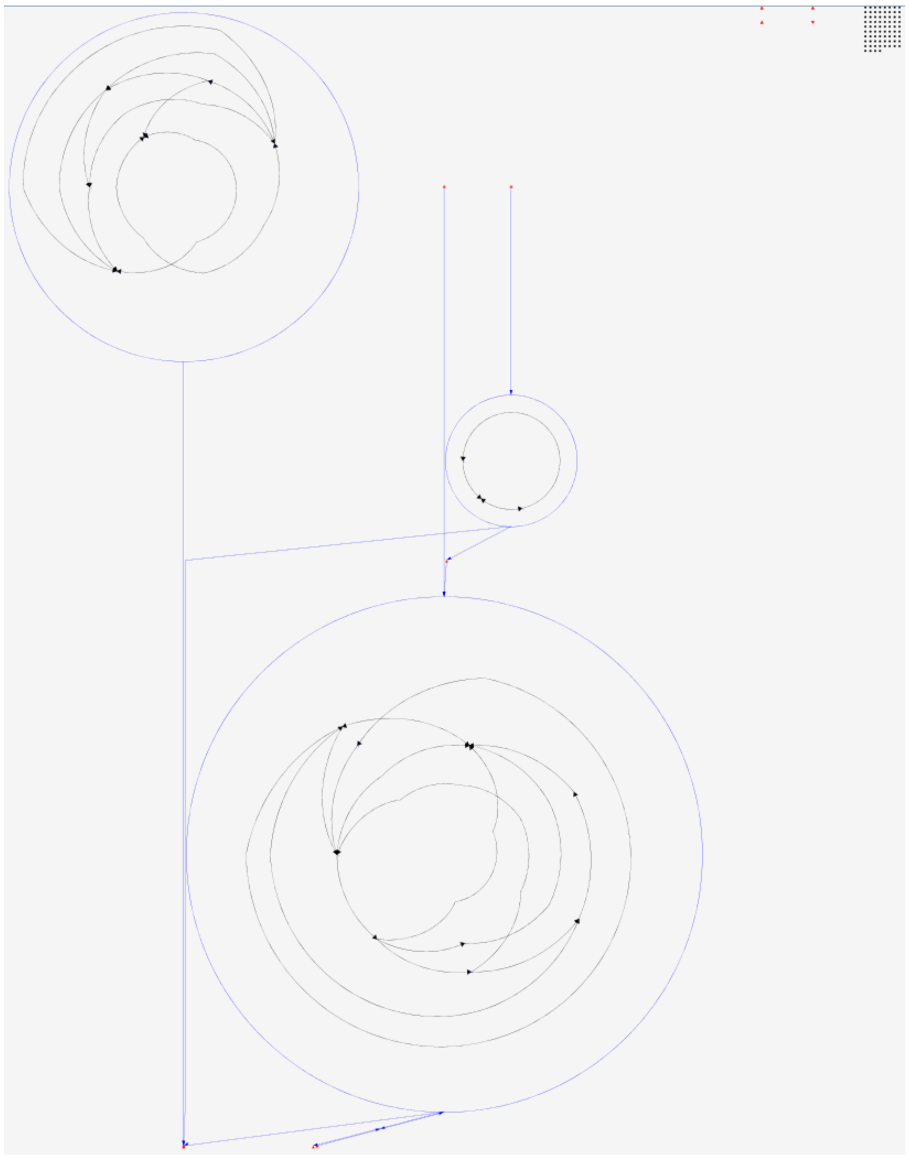


Figure 22.16: Small-multiples node-link visualization (SMNLV) of Stripes 1.5.7: method-call relation, enlarged view, two meta-nodes unfolded

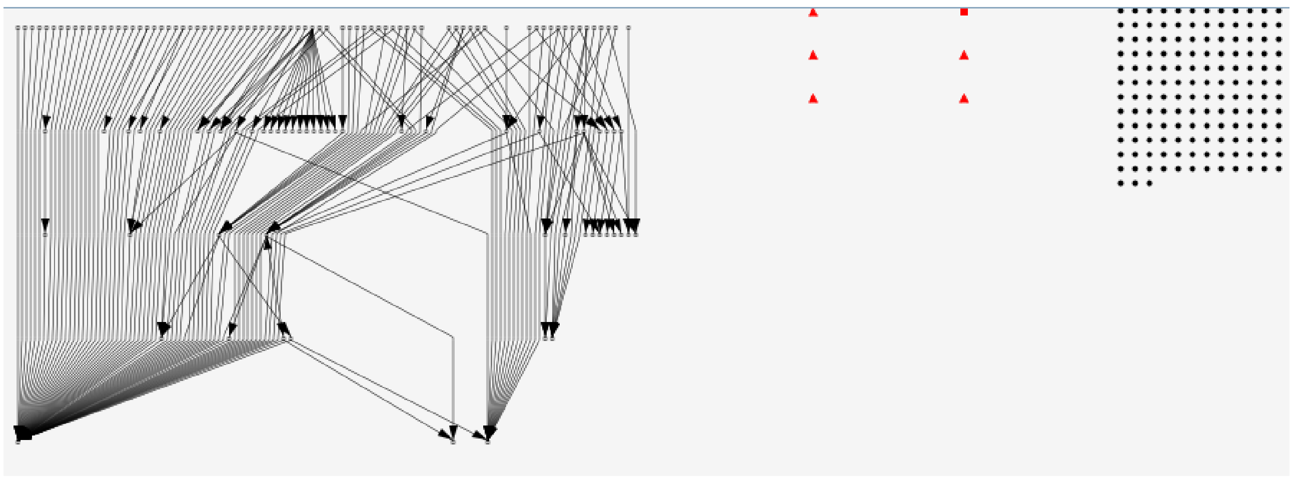


Figure 22.17: Small-multiples node-link visualization (SMNLV) of Stripes 1.5.7: aggregation relation, enlarged view, two meta-nodes unfolded

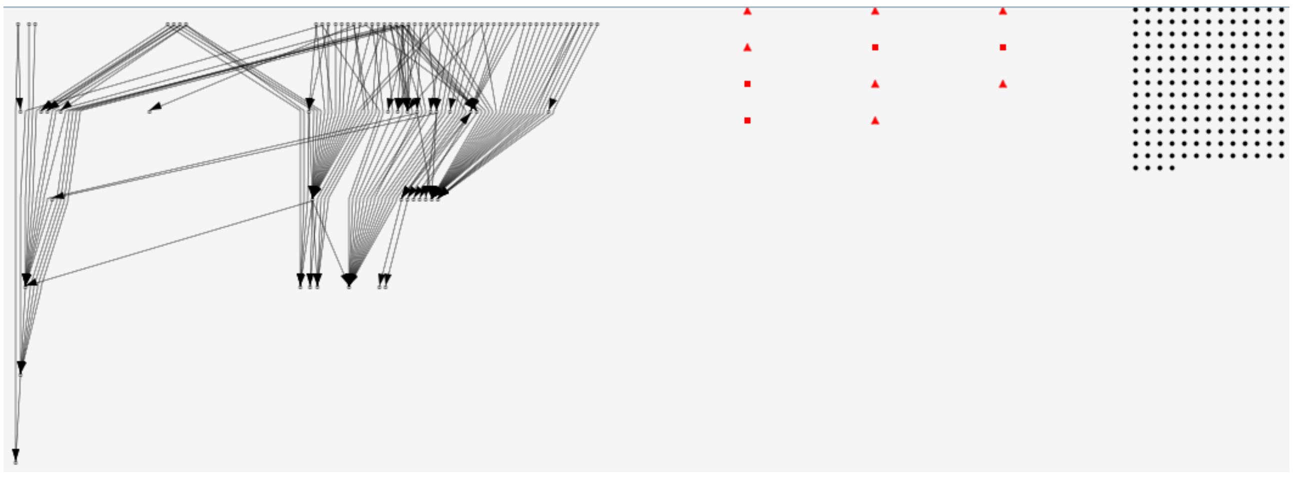


Figure 22.18: Small-multiples node-link visualization (SMNLV) of Stripes 1.5.7: parameter relation, enlarged view, two meta-nodes unfolded

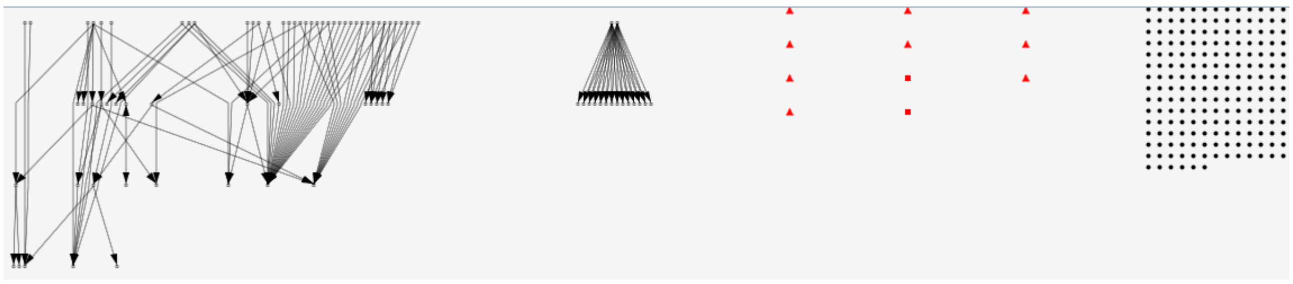


Figure 22.19: Small-multiples node-link visualization (SMNLV) of Stripes 1.5.7: return-type relation, enlarged view, two meta-nodes unfolded

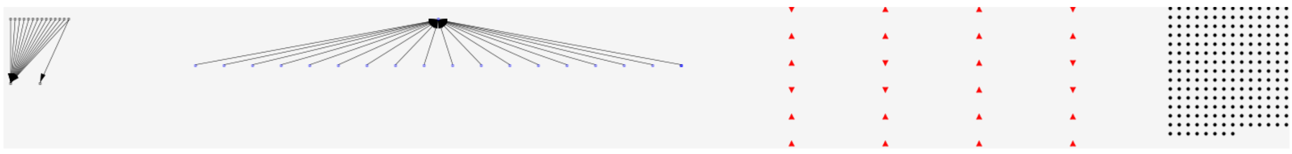


Figure 22.20: Small-multiples node-link visualization (SMNLV) of Stripes 1.5.7: implementation relation, enlarged view, two meta-nodes unfolded

22.5 Checkstyle

Showing the relationships of Checkstyle 6.5 [3] in IMMV creates a rather large view (Fig. 22.21). Interaction can be used to focus the analysis on specific parts of the graph or on higher, more abstract (package) levels. However, even in this view, characteristic structures are visible like the square in the lower right corner, the long stripes on the left, or the small stripes along the diagonal. Having a close look at the square in the lower right corner reveals, that it consists of four sub-structures (Fig. 22.22). Four types of these visual structures were distinguished based on their shape in Abuthawabeh et al. [8]: *line, diagonal cluster, off-diagonal cluster, and empty area*. The green dots to the right form a double line representing the method-call relations. This shows that methods of many different classes call methods of only two classes. The red dots on the bottom form a double line, too, representing the parameter relations. However, here two classes extensively use many other classes as parameter type. The brown dots forming a diagonal cluster represent the return-type relation. This shows that several classes are used as return type by methods of many classes. These relations occur between classes associated to same package as they are displayed along the diagonal.

The PNLV also shows characteristic pattern. Five categories of these visual structures were introduced by Burch et al. [38]: *fan, beam, cross beam, hour glass, and gap*. The inheritance and the throws relation are both limited to a small number of target classes, Classes having a high fan-in and fan-out can also easily be spotted, e.g., in the grammars.javadoc sub-package for the relations method-call, parameter, and return-type (Fig. 22.23). Also the differences can be clearly seen, e.g., in the upper part, where method-call and return-type originate in many different classes while parameter originates only in few, and where parameter and return-type end in many different classes while method-call only in two. The interaction provided by the tool allows to perform additional investigations that are beyond the scope of this paper.

While the previous approaches (PNLV and IMMV) provide class centric views focusing on similarities and differences between edge types wrt. classes and interfaces, the SMNLV approach focuses on the structure of each individual type of relation between entities. In Fig. 22.24, the meta-graphs for all relation types are shown (see also Section 20.1).

Checkstyle 6.5 contains 473 classes (nodes) and 2564 relations (edges, without package structure, Table 22.1). The largest graph is the return-type graph containing 826 edges between 184 nodes and 289 single nodes. As expected the package hierarchy is a single down-tree. Also, none of the weakly connected components of the throws, the inheritance, and the implementation relationships contains cycles. While this seems to be obvious, it is a good quality check for both the extraction process and the decomposition algorithm. In both the aggregation and the parameter relations, no cyclic dependencies are found. However, both the method-call and the return-type relations have one, respectively two weakly connected components that contain cyclic sub-graphs. For software engineers it might be interesting to find out, how large these cycles are (see also below and Fig. 22.25 and Fig. 22.26) and if a refactoring is necessary to remove (parts of) these cycles.

Having a close look at the type of the weakly connected components, we find that the throws, inheritance, and implementation relations only form tree structures. The parameter relation forms mostly tree structures (13) and 3 DAG structures. The same holds for the aggregation relation with 16 tree structures and 7 DAG structures.

The UML and SAVE approaches described in the related work Chapter 19 would not convey such information. The UML drawings are optimized for showing the inheritance relation, while the multi-edge connection of entities in SAVE prevents the differentiation of relation type specific cyclic and non-cyclic sub-graphs.

Opening the non-trivial cyclic sub-graph and the two tree meta-nodes of the method-call relation reveals, that the cycle is rather simple containing three nodes (Fig. 22.25). The trees are even more simple. However, it also shows that two classes from the cyclic structure call each other (double edge on the lower left of the cycle).

Unfolding the DAG of the second meta-graph of the return-type relation (Fig. 22.26) shows, that the DAG is shallow having three levels and that two classes are dominantly used as return-type.

Finally, unfolding two DAGs and one tree meta-node in the parameter relation (Fig. 22.27) shows one two-level DAG with the methods of two classes using a large number of different classes as parameter types, while the other DAG and the tree have a comparatively simple structure.

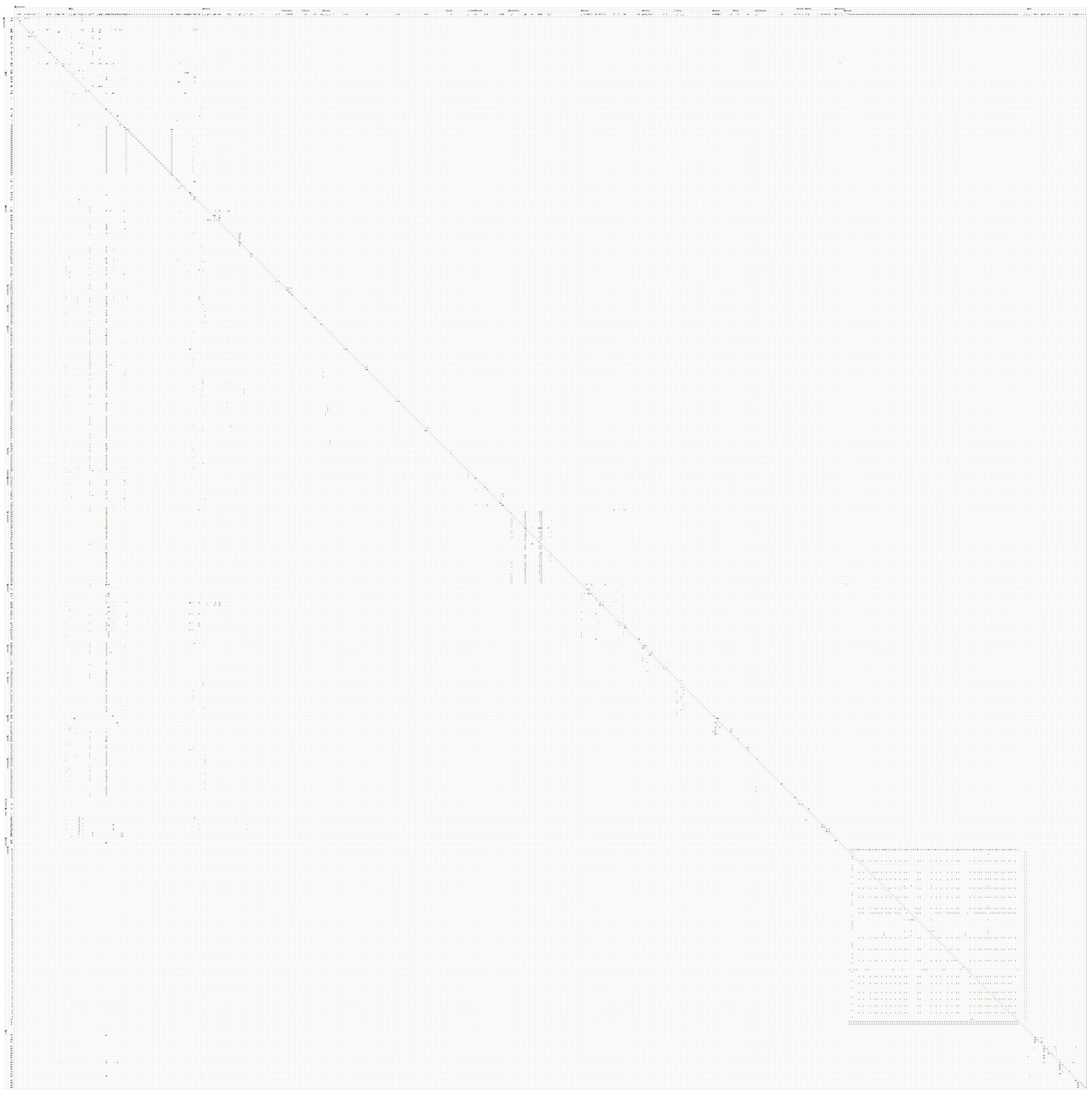


Figure 22.21: Interactive multi-matrix view (IMMV) [9] of Checkstyle 6.5

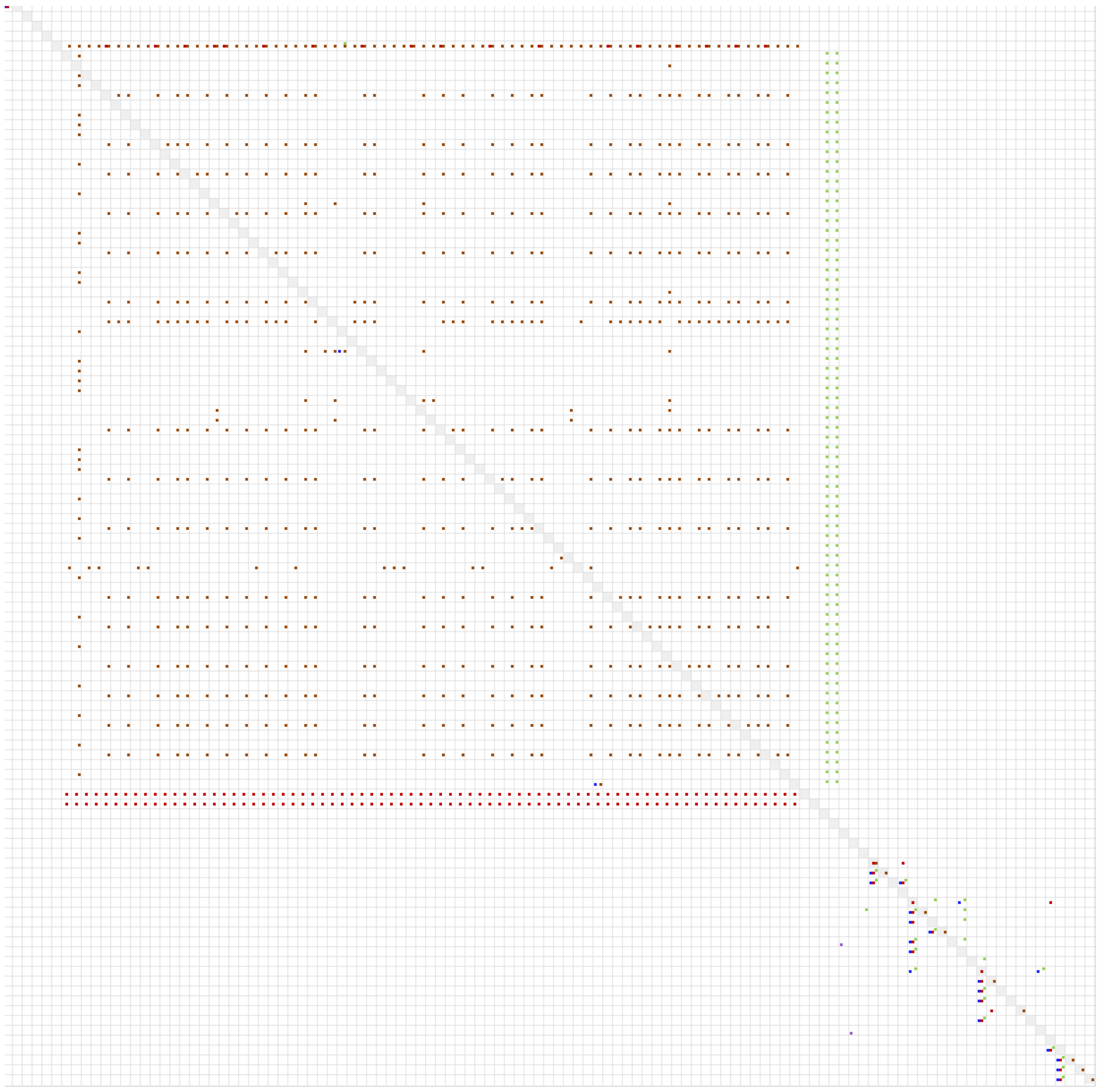


Figure 22.22: IMMV [9] of Checkstyle 6.5: large square in the lower right corner (close view)

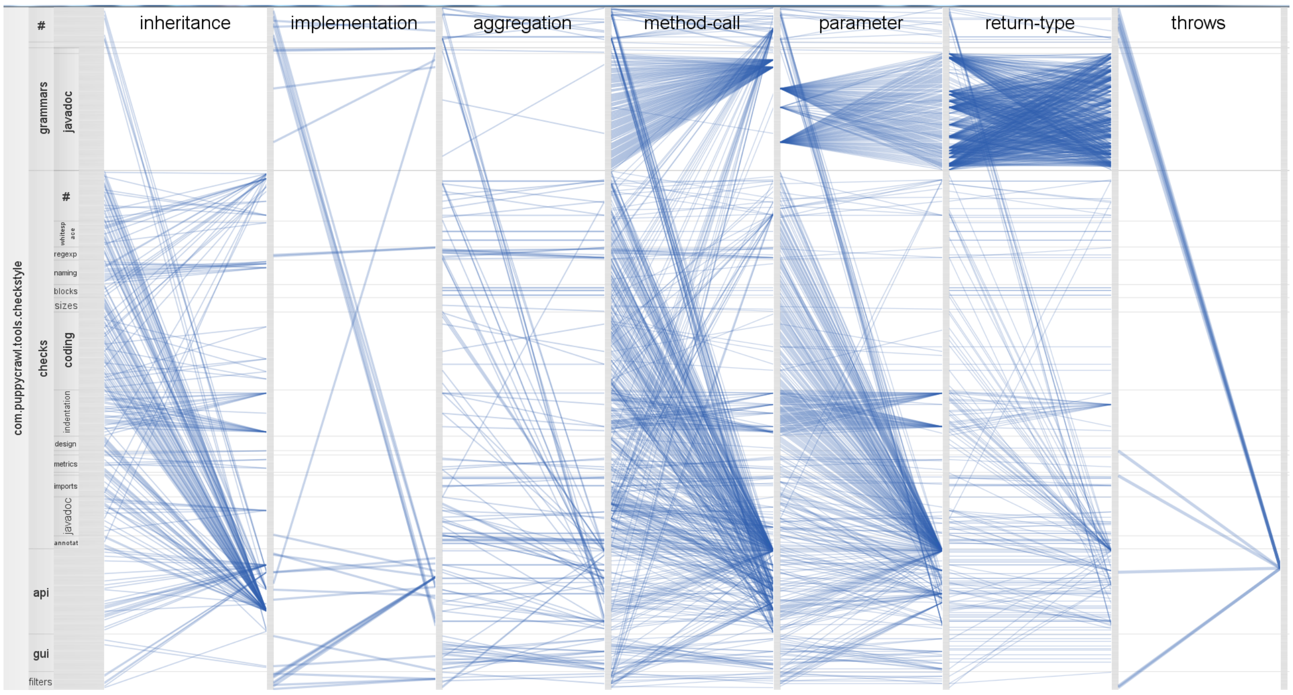


Figure 22.23: Parallel node-link visualization (PNLV) [26] of Checkstyle 6.5

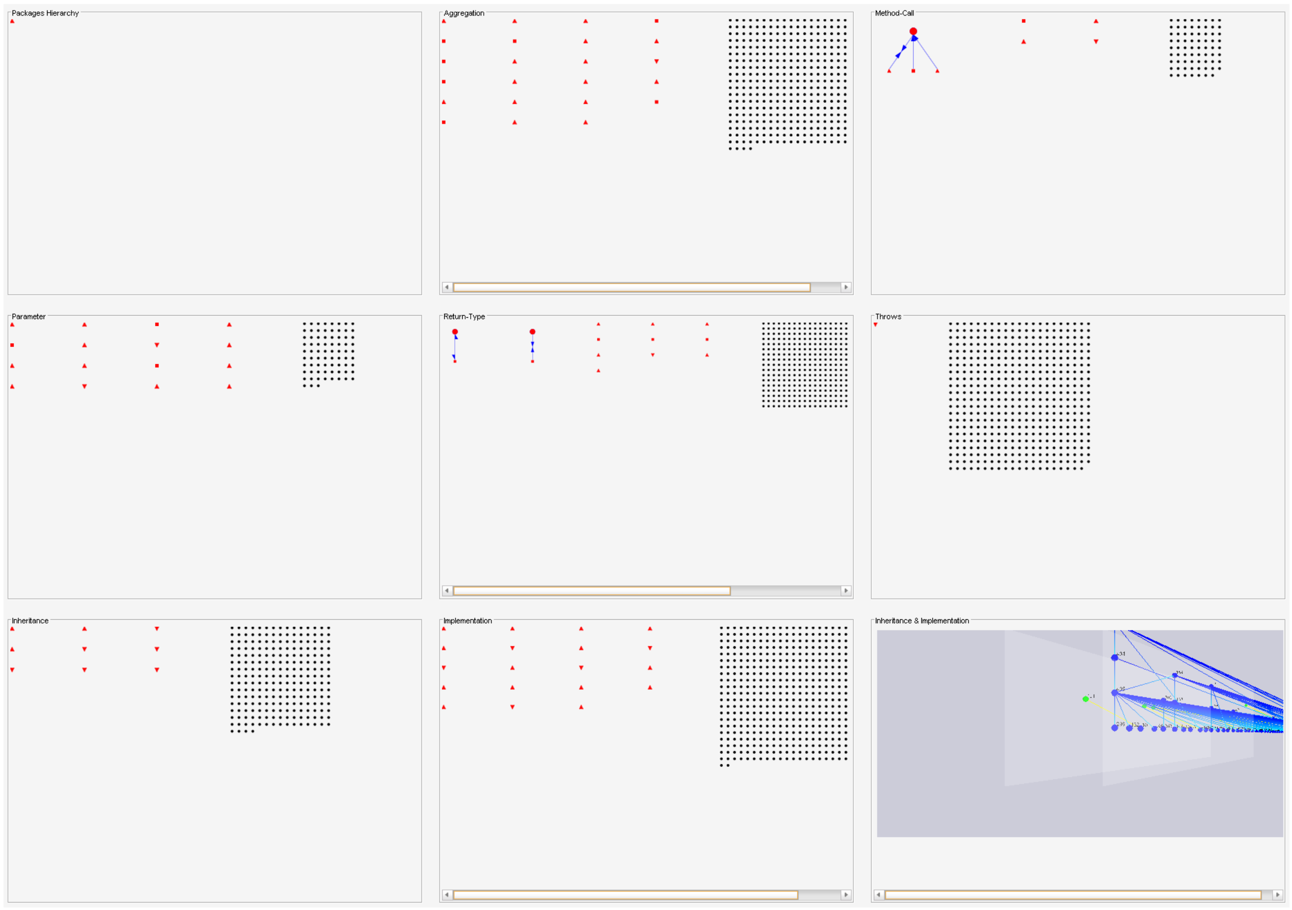


Figure 22.24: Small-multiples node-link visualization (SMNLV) of Checkstyle 6.5

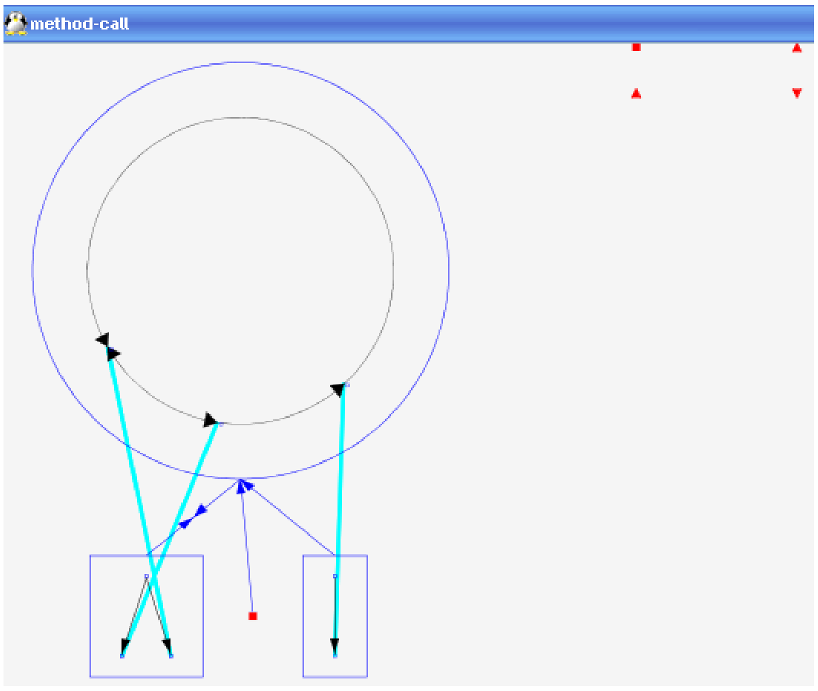


Figure 22.25: Small-multiples node-link visualization (SMNLV) of Checkstyle 6.5: method-call relation, enlarged view, three meta-nodes unfolded

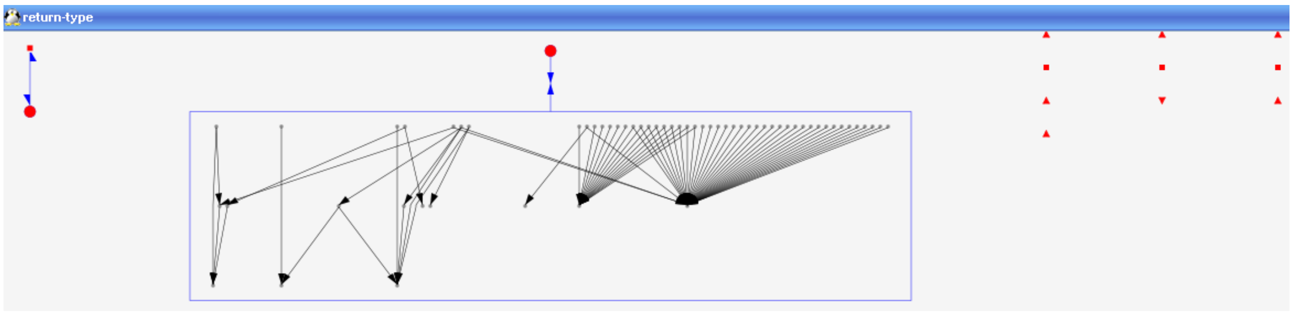


Figure 22.26: Small-multiples node-link visualization (SMNLV) of Checkstyle 6.5: return-type relation, enlarged view, one DAG meta-node unfolded

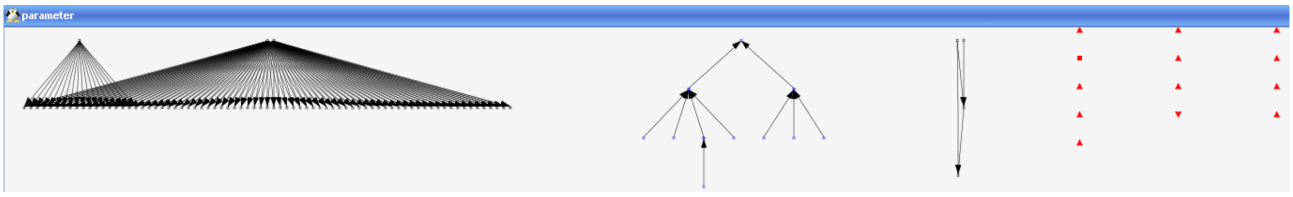


Figure 22.27: Small-multiples node-link visualization (SMNLV) of Checkstyle 6.5: parameter relation, enlarged view, two DAGs and one tree meta-nodes unfolded

Chapter 23

Conclusion

We presented a small-multiples node-link visualization supporting the analysis of the static structure of software systems based on multiple relations between classes and interfaces. Five use cases were explored showing the possibility to enrich the analysis by more insightful information. It complements previous approaches known from literature and benefits especially global analysis tasks. Future work should focus on which approaches fit to which task, and which combinations of the approaches would be beneficial. While evaluations would provide further insight, the space for these evaluations is definitely rather large. Therefore, a series of controlled experiments [76] is needed to assess differences of the approaches wrt. tasks, audience, and other parameters.

Part IV

Conclusion

Chapter 24

Conclusion

I developed and presented two different new approaches for visualizing directed graphs with multiple edge types for the analysis of static software structure. Additionally, I developed and analyzed a new topological decomposition and drawing algorithm that is geared towards showing topological structures for static software analysis.

Therefore, I described the development of IMMV and the determination of pattern in IMMV compared to PNLV using an explorative user study. Four categories of visual patterns in IMMV compared to PNLV were determined in the extended user study. The two tools are likely to support software comprehension.

In addition, I presented and analyzed algorithms that are used for the topological decomposition of the graphs into cyclic and acyclic subgraphs, and for building a two level hierarchy from these subgraphs for the purpose of drawing directed graphs topologically. This graph layout handles the drawbacks of the uniform drawing style when applying the Sugiyama method or the cyclic method only. The time and space complexity of the decomposition and the drawing algorithms are polynomial.

Approaches for extracting hierarchies from very large graphs may moderate the size of the decomposed subgraphs that remain large. Moreover, using interaction techniques together with the extracted hierarchies will facilitate dealing with large graphs.

Finally, I described a small-multiples node-link visualization (SMNLV) aiding the analysis of the static structure of software systems generated from multiple relations between classes and interfaces. This approach continues filling the space left by previous approaches known from literature and supports global analysis tasks in particular. Finally, five systems were visualized and analyzed. Future work should consider the most suitable match between the approches and their appropriate tasks, and which combinations of the approaches would be beneficial. As evaluations would enrich our insight, systematical evaluations of these visualization should be performed.

Abbreviations

Symbol	Description
DAG	**D**irected **A**cyclic **G**raph
DT	**D**own **T**ree
IMMV	**I**nteractive **m**ulti-**m**atrix **v**isualization
MMV	**m**ulti-**m**atrix **v**isualization
ntCS	**N**on-**t**rivial **c**yclic **s**ubgraphs
PNLV	**p**arallel **n**ode-**l**ink **v**isualization
SCC	**S**trongly **C**onnected **C**omponent
SMLNV	**S**mall-**M**ultiples **N**ode-**L**ink **V**isualization
wCC	**W**eakly **C**onnected **C**omponent
UT	**U**p **T**ree

Bibliography

[1] Stripes-1.5.7, 2012. https://github.com/StripesFramework/stripes/releases; Online; accessed 29-April-2015.

[2] JFtp-1.59, 2014. http://sourceforge.net/projects/j-ftp/files/jftp/; Online; accessed 29-April-2015.

[3] Checkstyle-6.5, 2015. http://checkstyle.sourceforge.net/; Online; accessed 29-April-2015.

[4] JUnit-4.12, 2015. https://github.com/junit-team/junit/releases; Online; accessed 29-April-2015.

[5] GraphML Working Group . GraphML, 2000. http://graphml.graphdrawing.org/; Online; accessed 12-December-2012.

[6] abego Software GmbH. abego TreeLayout , 2011. http://treelayout.sourceforge.net/; Online; accessed 8-May-2015.

[7] Ala Abuthawabeh. Software matrix layout. Master's thesis, University of Kaiserslautern, Germany, 2012.

[8] Ala Abuthawabeh, Fabian Beck, Dirk Zeckzer, and Stephan Diehl. Finding Structures in Multi-Type Code Couplings with Node-Link and Matrix Visualizations. In *Proceedings of the first IEEE Working Conference on Software Visualization 2013*, VISSOFT 2013. IEEE Computer Society, 2013.

[9] Ala Abuthawabeh and Dirk Zeckzer. IMMV: An Interactive Multi-Matrix Visualization for Program Comprehension. In *Proceedings of the first IEEE Working Conference on Software Visualization 2013*, VISSOFT 2013. IEEE Computer Society, 2013.

[10] Ala Abuthawabeh and Dirk Zeckzer. An Improved Decomposition and Drawing Process for Optimal Topological Visualization of Directed Graphs. In *Proceedings of the 31th Spring Conference on Computer Graphics*, SCCG'15, pages 111–118. ACM, 2015.

[11] Ala Abuthawabeh and Dirk Zeckzer. SMNLV: A Small-Multiples Node-Link Visualization Supporting Software Comprehension by Displaying Multiple Relationships in Software Structure, 2015.

[12] Ragaad AlTarawneh, Max Langbein, Shah Rukh Humayoun, and Hans Hagen. TopoLayout-DG: A Topological Feature-Based Framework for Visualizing Inside Behavior of Large Directed Graphs. In *Proceedings of SIGRAD 2014, Visual Computing, June 12-13, 2014, Göteborg, Sweden*, pages 87–90, 2014.

[13] Keith Andrews, Martin Wohlfahrt, and Gerhard Wurzinger. Visual Graph Comparison. In *IV '09: Proceedings of the 13th Conference on Information Visualisation*, pages 62–67. IEEE Computer Society, 2009.

[14] D. Archambault and H.C. Purchase. The mental map and memorability in dynamic graphs. In *Pacific Visualization Symposium (PacificVis), 2012 IEEE*, pages 89–96, 2012.

[15] D. Archambault, H.C. Purchase, and B. Pinaud. Animation, small multiples, and the effect of mental map preservation in dynamic graphs. *Visualization and Computer Graphics, IEEE Transactions on*, 17(4):539–552, 2011.

[16] Daniel Archambault. Structural differences between two graphs through hierarchies. In *GI '09: Proceedings of Graphics Interface 2009*, pages 87–94. Canadian Information Processing Society, 2009.

[17] Daniel Archambault, Tamara Munzner, and David Auber. TopoLayout: Multilevel Graph Layout by Topological Features. *IEEE Transactions on Visualization and Computer Graphics*, 13(2):305–317, 2007.

[18] Daniel Archambault, Tamara Munzner, and David Auber. GrouseFlocks: Steerable Exploration of Graph Hierarchy Space. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):900–913, 2008.

[19] Daniel Archambault, Tamara Munzner, and David Auber. Tugging Graphs Faster: Efficiently Modifying Path-Preserving Hierarchies for Browsing Paths. *IEEE Transactions on Visualization and Computer Graphics*, 17(3):276–289, 2011.

[20] Christian Bachmaier, Franz-Josef Brandenburg, Wolfgang Brunner, and Raymund Fülöp. Coordinate assignment for cyclic level graphs. In *Computing and combinatorics*, pages 66–75. Springer, 2009.

[21] Christian Bachmaier, Franz-Josef Brandenburg, Wolfgang Brunner, and Raymund Fülöp. Drawing Recurrent Hierarchies. *J. Graph Algorithms Appl.*, 16(2):151–198, 2012.

[22] Christian Bachmaier, Franz-Josef Brandenburg, Wolfgang Brunner, and Gergö Lovsz. Cyclic Leveling of Directed Graphs. In *Graph Drawing*, volume 5417 of *Lecture Notes in Computer Science*, pages 348–359. Springer Berlin Heidelberg, 2009.

[23] Christian Bachmaier, FranzJ. Brandenburg, Michael Forster, Paul Holleis, and Marcus Raitner. Gravisto: Graph Visualization Toolkit. In Jnos Pach, editor, *Graph Drawing*, volume 3383 of *Lecture Notes in Computer Science*, pages 502–503. Springer Berlin Heidelberg, 2005.

[24] Barak Naveh. JGraphT, 2013. http://www.jgrapht.org; Online; accessed 26-September-2014.

[25] Vladimir Batagelj, Franz-Josef Brandenburg, Walter Didimo, Giuseppe Liotta, Pietro Palladino, and Maurizio Patrignani. Visual Analysis of Large Graphs Using (X, Y)-Clustering and Hybrid Visualizations. *IEEE Transactions on Visualization and Computer Graphics*, 17(11):1587–1598, 2011.

[26] F. Beck, R. Petkov, and S. Diehl. Visually exploring multi-dimensional code couplings. In *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*, pages 1 –8, sept. 2011.

[27] Fabian Beck. *Understanding Multi-Dimensional Code Couplings*. PhD thesis, University of Trier, Trier, Germany, 2013.

[28] Fabian Beck and Stephan Diehl. Visual Comparison of Software Architectures. In *SOFTVIS '10: Proceedings of the ACM 2010 Symposium on Software Visualization*, pages 183–192. ACM, 2010.

[29] Fabian Beck and Stephan Diehl. On the congruence of modularity and code coupling. In *SIGSOFT/FSE '11 and ESEC '11: Proceedings of the 19th ACM SIGSOFT Symposium on the Foundations of Software Engineering and 13rd European Software Engineering Conference*, pages 354–364. ACM, 2011.

[30] Fabian Beck and Stephan Diehl. Visual comparison of software architectures. *Information Visualization*, 12(2):178–199, 2013.

[31] Moritz Becker and Isabel Rojas. A Graph Layout Algorithm for Drawing Metabolic Pathways. *Bioinformatics*, 17:461–467, 2001.

[32] Romain Bourqui, Ludovic Cottret, Vincent Lacroix, David Auber, Patrick Mary, Marie-France Sagot, and Fabien Jourdan. Metabolic network visualization eliminating node redundance and preserving metabolic pathways. *BMC systems biology*, 1(1):29, 2007.

[33] Romain Bourqui, Paolo Simonetto, and Fabien Jourdan. A Stable Decomposition Algorithm for Dynamic Social Network Analysis. In Fabrice Guillet, Gilbert Ritschard, DjamelAbdelkader Zighed, and Henri Briand, editors, *Advances in Knowledge Discovery and Management*, volume 292 of *Studies in Computational Intelligence*, pages 167–178. Springer Berlin Heidelberg, 2010.

[34] Ulrik Brandes, Tim Dwyer, and Falk Schreiber. Visualizing Related Metabolic Pathways in Two and a Half Dimensions Graph Drawing. In Giuseppe Liotta, editor, *Graph Drawing*, volume 2912 of *Lecture Notes in Computer Science*, chapter 10, pages 111–122. Springer Berlin / Heidelberg, 2004.

[35] Ulrik Brandes and Boris Köpf. Fast and Simple Horizontal Coordinate Assignment. In Petra Mutzel, Michael Jünger, and Sebastian Leipert, editors, *Graph Drawing*, volume 2265 of *Lecture Notes in Computer Science*, pages 31–44. Springer Berlin Heidelberg, 2002.

[36] S. Bremm, T. von Landesberger, M. Hess, T. Schreck, P. Weil, and K. Hamacherk. Interactive visual comparison of multiple trees. In *Visual Analytics Science and Technology (VAST), 2011 IEEE Conference on*, pages 31–40, 2011.

[37] Christoph Buchheim, Michael Jünger, and Sebastian Leipert. Improving Walker's Algorithm to Run in Linear Time. In MichaelT. Goodrich and StephenG. Kobourov, editors, *Graph Drawing*, volume 2528 of *Lecture Notes in Computer Science*, pages 344–353. Springer Berlin Heidelberg, 2002.

[38] Michael Burch, Corinna Vehlow, Fabian Beck, Stephan Diehl, and Daniel Weiskopf. Parallel Edge Splatting for Scalable Dynamic Graph Visualization. *IEEE Transactions on Visualization and Computer Graphics*, 17(12):2344–2353, December 2011.

[39] Michael Burch and Daniel Weiskopf. A flip-book of edge-splatted small multiples for visualizing dynamic graphs. In *Proceedings of the 7th International Symposium on Visual Information Communication and Interaction*, VINCI '14, pages 29:29–29:38, New York, NY, USA, 2014. ACM.

[40] P. Caserta and O. Zendra. Visualization of the static aspects of software: A survey. *Visualization and Computer Graphics, IEEE Transactions on*, 17(7):913–933, July 2011.

[41] Rylan Cottrell, Brina Goyette, Reid Holmes, Robert J. Walker, and Jrg Denzinger. Compare and contrast: Visual exploration of source code examples. In *Proceedings of the 5th IEEE International Workshop on Visualizing Software for Understanding and Analysis, VISSOFT 2009, September 25, 2009, Edmonton, Alberta, Canada*, pages 29–32. IEEE Computer Society, 2009.

[42] Stephan Diehl. *Software Visualization - Visualizing the Structure, Behaviour, and Evolution of Software.* Springer, 2007.

[43] Stephan Diehl and Carsten Görg. Graphs, They Are Changing. In *Graph Drawing*, volume 2528 of *Lecture Notes in Computer Science*, pages 23–31. Springer-Verlag, 2002.

[44] Sabri Skhiri dit Gabouje and Esteban Zimányi. A New Constraint-Based Compound Graph Layout Algorithm for Drawing Biochemical Networks. In Fernando Ferri, editor, *Visual Languages for Interactive Computing: Definitions and Formalizations*, chapter 19, pages 407–424. IGI Global, 2007.

[45] Don Ho. Notepad++ v6.1.6, 2012. http://notepad-plus-plus.org/; Online; accessed 3-December-2012.

[46] S. Ducasse and M. Lanza. The class blueprint: visually supporting the understanding of glasses. *Software Engineering, IEEE Transactions on*, 31(1):75 – 90, jan. 2005.

[47] Holger Eichelberger. Nice class diagrams admit good design? In *Proceedings ACM 2003 Symposium on Software Visualization, San Diego, California, USA, June 11-13, 2003*, pages 159–167, 2003.

[48] Markus Eiglsperger, Carsten Gutwenger, Michael Kaufmann, Joachim Kupke, Michael Jünger, Sebastian Leipert, Karsten Klein, Petra Mutzel, and Martin Siebenhaller. Automatic layout of UML class diagrams in orthogonal style. *Information Visualization*, 3(3):189–208, 2004.

[49] Markus Eiglsperger, Michael Kaufmann, and Frank Eppinger. An approach for mixed upward planarization. *J. Graph Algorithms Appl.*, 7(2):203–220, 2003.

[50] Geoffrey Ellis and Alan Dix. An Explorative Analysis of User Evaluation Studies in Information Visualisation. In *BELIV '06: Proceedings of the 2006 AVI Workshop on Beyond Time and Errors: Novel Evaluation Methods for Information Visualization*, pages 1–7. ACM, 2006.

[51] Niklas Elmqvist, Thanh-Nghi Do, Howard Goodell, Nathalie Henry, and Jean-Daniel Fekete. ZAME: Interactive Large-Scale Graph Visualization. In *IEEE VGTC Pacific Visualization Symposium 2008 (PacificVis 2008)*, pages 215–222, March 2008.

[52] U. Erdemir, U. Tekin, and F. Buzluca. E-quality: A graph based object oriented software quality visualization tool. In *Visualizing Software for Understanding and Analysis (VISSOFT), 2011 6th IEEE International Workshop on*, pages 1 –8, sept. 2011.

[53] F. van Ham. *Interactive Visualization of Large Graphs.* PhD thesis, Technische Universiteit Eindhoven, 2005.

[54] Y. Frishman and A. Tal. Online Dynamic Graph Drawing. *IEEE Transactions on Visualization and Computer Graphics*, 14(4):727–740, July 2008.

[55] Johannes Fuchs, Fabian Fischer, Florian Mansmann, Enrico Bertini, and Petra Isenberg. Evaluation of alternative glyph designs for time series data in a small multiple setting. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, CHI '13, pages 3237–3246, New York, NY, USA, 2013. ACM.

[56] E. R. Gansner, S. C. North, and K. P. Vo. Daga program that draws directed graphs. *Software: Practice and Experience*, 18(11):1047–1062, November 1988.

[57] Mohammad Ghoniem, Jean-Daniel Fekete, and Philippe Castagliola. A comparison of the readability of graphs using node-link and matrix-based representations. In *Proceedings of the IEEE Symposium on Information Visualization*, INFOVIS '04, pages 17–24, Washington, DC, USA, 2004. IEEE Computer Society.

[58] Mohammad Ghoniem, Jean-Daniel Fekete, and Philippe Castagliola. On the Readability of Graphs Using Node-Link and Matrix-Based Representations: A Controlled Experiment and Statistical Analysis. *Information Visualization*, 4(2):114–135, 2005.

[59] Michael Gleicher, Danielle Albers, Rick Walker, Ilir Jusufi, Charles D. Hansen, and Jonathan C. Roberts. Visual comparison for information visualization. *Information Visualization*, 10(4):289–309, 2011.

[60] J.L. Gross and J. Yellen. *Graph Theory & Its Applications.* Discrete Mathematics And Its Applications. Crc, 1999.

[61] Carsten Gutwenger, Michael Jünger, Karsten Klein, Joachim Kupke, Sebastian Leipert, and Petra Mutzel. A new approach for visualizing UML class diagrams. In *Proceedings of the 2003 ACM Symposium on Software Visualization*, SoftVis '03, pages 179–188, June 2003.

[62] IFA Informatik and Erich Gamma. JHotDraw, 1996, 1997. `http://www.jhotdraw.org/`; Online; accessed 12-December-2012.

[63] W. Javed and N. Elmqvist. Exploring the design space of composite visualization. In *Pacific Visualization Symposium (PacificVis), 2012 IEEE*, pages 1–8, 2012.

[64] jclassinfo. `http://jclassinfo.sourceforge.net/`; Online; accessed 8-May-2015.

[65] D. Johnson. Finding All the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing*, 4(1):77–84, 1975.

[66] Peter D Karp, John D Lowrance, Thomas M Strat, and David E Wilkins. The Grasper-CL graph management system. *LISP and Symbolic Computation*, 7(4):251–290, 1994.

[67] Peter D Karp and Suzanne Paley. Automated Drawing of Metabolic Pathways. In H. Lim, C. Cantor, and R. Robbins, editors, *Third International Conference on Bioinformatics and Genome Research*, pages 225–238, 1994.

[68] Johannes Kehrer, Harald Piringer, Wolfgang Berger, and Meister Eduard Gröller. A model for structure-based comparison of many categories in small-multiple displays. *IEEE Transactions on Visualization and Computer Graphics*, 19(12):2287–2296, December 2013.

[69] René Keller, Claudia M. Eckert, and P. John Clarkson. Matrices or node-link diagrams: which visual representation is better for visualising connectivity models? *Information Visualization*, 5(1):62–76, March 2006.

[70] K. Koffka. *Principles of gestalt psychology.* International library of psychology, philosophy, and scientific method. Harcourt, Brace and Company, 1935.

[71] H. Lam, E. Bertini, P. Isenberg, C. Plaisant, and S. Carpendale. Empirical Studies in Information Visualization: Seven Scenarios. *IEEE Transactions on Visualization and Computer Graphics*, 18(9):1520–1536, 2012.

[72] Antoine Lambert, Jonathan Dubois, and Romain Bourqui. Pathway preserving representation of metabolic networks. *Computer Graphics Forum*, 30(3):1021–1030, 2011.

[73] J. Laval, S. Denier, S. Ducasse, and A. Bergel. Identifying cycle causes with enriched dependency structural matrix. In *Reverse Engineering, 2009. WCRE '09. 16th Working Conference on*, pages 113–122, 2009.

[74] Jannik Laval and Stéphane Ducasse. Resolving cyclic dependencies between packages with enriched dependency structural matrix. *Software: Practice and Experience*, 44(2):235–257, 2014.

[75] Jannik Laval, Jean-Rémy Falleri, Philippe Vismara, and Stéphane Ducasse. Efficient Retrieval and Ranking of Undesired Package Cycles in Large Software Systems. *Journal of Object Technology*, 11(1):1–24, 2012.

[76] Richard Müller, Pascal Kovacs, Jan Schilbach, Ulrich W. Eisenecker, Dirk Zeckzer, and Gerik Scheuermann. A structured approach for conducting a series of controlled experiments in software visualization. In *Proceedings of the 5th International Conference on Information Visualization Theory and Applications, IVAPP 2014, Lisbon, Portugal, 5-8 January, 2014.*, pages 204–209, 2014.

[77] Christopher R. Myers. Software systems as complex networks: Structure, function, and evolvability of software collaboration graphs. *Physical Review E*, 68(4):046116+, 2003.

[78] Matthias Naab, Thomas Forster, Jens Knodel, and Dirk Muthig. Static evaluation of software architectures. Technical Report 078.05E, Fraunhofer IESE, 2005.

[79] NetBeans. Visual Library. http://graph.netbeans.org; Online; accessed 29-April-2015.

[80] A. Johannes Pretorius and Jarke J. van Wijk. Visual Inspection of Multivariate Graphs. *Computer Graphics Forum*, 27(3):967–974, 2008.

[81] Dennie Reniers, Lucian Voinea, Ozan Ersoy, and Alexandru Telea. The solid* toolset for software visual analytics of program structure and metrics comprehension: From research prototype to product. *Sci. Comput. Program.*, 79:224–240, January 2014.

[82] Jonathan C. Roberts. State of the art: Coordinated & multiple views in exploratory visualization. In *Proceedings of the Fifth International Conference on Coordinated and Multiple Views in Exploratory Visualization*, CMV '07, pages 61–71, Washington, DC, USA, 2007. IEEE Computer Society.

[83] Markus Rohrschneider, Christian Heine, André Reichenbach, Andreas Kerren, and Gerik Scheuermann. A novel grid-based visualization approach for metabolic networks with advanced focus&context view. In *Graph Drawing*, pages 268–279. Springer Berlin Heidelberg, 2010.

[84] Falk Schreiber. Visual comparison of metabolic pathways. *Journal of Visual Languages and Computing*, 14(4):327–340, 2003.

[85] John Stasko, Carsten Görg, and Zhicheng Liu. Jigsaw: supporting investigative analysis through interactive visualization. *Information Visualization*, 7(2):118–132, April 2008.

[86] Klaus Stein, Rene Wegener, and Christoph Schlieder. Pixel-Oriented Visualization of Change in Social Networks. In *Proceedings of the 2010 International Conference on Advances in Social Networks Analysis and Mining*, ASONAM '10, pages 233–240, Washington, DC, USA, 2010. IEEE Computer Society.

[87] Kozo Sugiyama, Shojiro Tagawa, and Mitsuhiko Toda. Methods for Visual Understanding of Hierarchical System Structures. *IEEE Transactions on Systems, Man and Cybernetics*, 11(2):109–125, Feb 1981.

[88] Jayme Luiz Szwarcfiter and Peter Ernst Lauer. Finding the elementary cycles of a directed graph in O(n+m) per cycle. Technical Report 060, University of Newcastle upon Tyne (GB), 1974.

[89] Roberto Tamassia, editor. *Handbook of Graph Drawing and Visualization*. Discrete Mathematics and Its Applications. Chapman & Hall/CRC, 2007.

[90] R. Tarjan. Enumeration of the Elementary Circuits of a Directed Graph. *SIAM Journal on Computing*, 2(3):211–216, 1973.

[91] TechSmith. Camtasia Studio 8, 2012. http://www.techsmith.com/camtasia.html; Online; accessed 3-December-2012.

[92] James C. Tiernan. An Efficient Search Algorithm to Find the Elementary Circuits of a Graph. *Commun. ACM*, 13(12):722–726, December 1970.

[93] Christian Tominski, James Abello, and Heidrun Schumann. Cgvan interactive graph visualization system. *Computers & Graphics*, 33(6):660–678, 2009.

[94] E.R. Tufte. *The visual display of quantitative information*. Number v. 914 in The Visual Display of Quantitative Information. Graphics Press, 1983.

[95] E.R. Tufte. *Envisioning information*. Number v. 914 in Envisioning Information. Graphics Press, 1990.

[96] E.R. Tufte. *Beautiful Evidence*. Graphics Press, 2006.

[97] Stef van den Elzen and Jarke J. van Wijk. Small multiples, large singles: A new approach for visual data exploration. *Computer Graphics Forum*, 32:191–200, 2013.

[98] T. von Landesberger, A. Kuijper, T. Schreck, J. Kohlhammer, J. J. van Wijk, J. D. Fekete, and D. W. Fellner. Visual Analysis of Large Graphs: State-of-the-Art and Future Research Challenges. *Computer Graphics Forum*, 30(8):1719–1749, 2011.

[99] Michelle Q. Wang Baldonado, Allison Woodruff, and Allan Kuchinsky. Guidelines for using multiple views in information visualization. In *Proceedings of the working conference on Advanced visual interfaces*, AVI '00, pages 110–119, New York, NY, USA, 2000. ACM.

[100] Katja Wegner and Ursula Kummer. A new dynamical layout algorithm for complex biochemical reaction networks. *BMC Bioinformatics*, 6(1):1–12, 2005.

[101] Dirk Zeckzer. Visualizing Software Entities Using a Matrix Layout. In *Proceedings of the 5th international symposium on Software visualization*, SOFTVIS '10, pages 207–208, New York, NY, USA, 2010. ACM.

List of Figures

List of Tables

List of Algorithms

Part V

Appendix

Appendix A

First Pretest

No	Question	Participant Answer	Correct Answer	Result
1	Which classes or interfaces are included in the package (org.jhotdraw.gui.datatransfer)?	AWTClipboared, Abstract-Clipboared, AbstractTransferable	AWTClipboared, Abstract-Clipboared, Abstract-Transferable, ClipboaredUtil, CompositTransferable, Image-Transferable, InputStream-Transferable	wrong
2	Which different types of dependencies exist *from* (org.jhotdraw.gui.datatransfer. AWTClipboard) *to* (org.jhotdraw.gui.datatransfer. AbstractClipboard)? please consider the direction.	method-call (Usage), aggregation, code-clone, semantic	method-call (Usage), aggregation, code-clone, semantic	correct
3	Is there any code clone dependency between the files of packages (org.jhotdraw.gui.datatransfer) and (org.jhotdraw.gui.plaf) in any direction?	(*from* datatransfer *to* palatte: no; *from* pallete *to* datatransfer: yes)	(*from* datatransfer *to* palatte: no; *from* pallete *to* datatransfer: yes)	correct
4	Please choose an aggregation dependency of (org.jhotdraw.gui.fontchooser. FontFamilyNode) and retrieve the fields that create this dependency in the source code?	line 29: private FontCollectionNode parent;	line 29: private FontCollectionNode parent;	correct

Table A.1: Trial results for IMMV using the JHotDraw dataset

No	Question	Participant Answer	Correct Answer	Result
1	Which classes or interfaces are included in package (org.jhotdraw.gui.datatransfer)?	GenericListner, SheetEvent, SheetListner	GenericListner, SheetEvent, SheetListner	correct
2	Which different types of dependencies exist *from* (org.jhotdraw.gui. datatransfer.AWTClipboard) *to* (org.jhotdraw.gui.datatransfer. AbstractClipboard) please consider the direction?	Inheritance, Evolutionary, Semantic	Inheritance, Evolutionary, Semantic	correct
3	Is there any code clone dependency between the files of package (org.jhotdraw.gui.datatransfer) and (org.jhotdraw.gui.plaf) in any direction?	No		
4	Please choose an aggregation dependency of (org.jhotdraw.gui. fontchooser.FontFaceNode) and retrieve the fields that create this dependency in the source code?	Line 28: private FontFamilyNode parent;	Line 28: private FontFamilyNode parent;	correct

Table A.2: Trial results for PNLV using the JHotDraw dataset

Dataset	ID	IMMV Explanation
JUnit	1	Many edges in the builder package edited at the same time. Many semantic similarity couplings exist.
	2	Blocks of evolutionary couplings exist between packages model and runners and within the model package.
	3	Nearly all classes of the builder package inherit from the model package classes and from the statements package classes.
	4	Semantic similarity couplings exist within theories package.
	5	Code clones and semantic similarity couplings exist between top level (junit package) and the other packages.
	6	Many types of couplings exist off the diagonal.

Table A.3: Results of the IMMV paper test

Dataset	ID	PNLV Explanation
JFtp	1	Many inheritance edges directed from gui package to framework package (most of them directed to the same class).
	2	A strong connection of parts of the gui package to the framework package (with gaps).
	3	A class in the net package is used from different packages.
	4	Large fan-in evolutionary couplings exists for some classes.
	5	Code clone couplings exist between different packages.
	6	Many classes have semantic similarity couplings, but no inheritance, aggregation, or usage couplings.

Table A.4: Results of the PNLV paper test

ID & Time	Reason	Exploration Process	Finding
3 Start 16:10 End 16:14	There are inheritance edges.	Finding an abstract class Runners Builder. Opening different files. Finding an outlier. Finding abstract class statement. ExceptException has inheritance and aggregation.	A set of classes for doing similar things.
4 Start 16:14 End 16:20	There are a lot of relations.	Opening the two respective classes. looking at the the source code.	Inheritance perhaps leads to semantic matching even smaller than before (see 3).
5 Start 16:20 End 16:24	The block exists off the diagonal	Comparing two files to find the clones	Two comparison classes in different packages. Something are different. Hint: create a superclass for these or rename them.

Table A.5: Results of the interactive IMMV test

ID & Time	Reason	Exploration Process	Finding
1 Start 16:46 End 16:49	The first one is inheritance.	Highlighting the target	There are different variants of dialogs → Basic GUI class.
6 Start 16:49 End 16:53		Highlighting the connected classes. Checking the assumption that there are no structural dependencies. Opening files	Importing the same packages that do similar things. Making the dependency explicit.
4 Start 16:53 End 16:57		Highlighting all respective entities. Comparing to other graphs. Opening classes (the first was the wrong ones).	Using the logger instead (the logger is not used).

Table A.6: Results of the interactive PNLV test

No	Question	Answer
1	Gender	Male
2	Age	29
3	Your highest degree obtained up to now	Master
4	Your major subject	computer science
5	Number of Software Engineering courses you attended	0
6	Years of experience in Programming	10
7	Programming languages you use regularly	Java
8	Number of classes/ files that were in the largest project you worked on:	51-100 classes/files
9	Size of team/group you usually develop software together:	$\leqslant$ 3 developers
10.a	Are you familiar with the term "design pattern"?	Yes
10.b	b. If yes, what kind of design patterns do you use regularly	None
11	Visual representations you regularly use for software development:	UML, Sketches

Table A.7: Results of the general questionnaire

No	Question	Answer
1	Which visualization technique do you prefer for analyzing software projects? Why?	PNLV, because focusing on single relation type is easier using the separated relation views (columns) . For larger projects, i'd probably prefer IMMV because PNLV can get messy when displaying too many edges.
2	The visualization is useful for analyzing software projects	(IMMV: agree; PNLV: agree).
3	To enhance the tool, which interaction features should be added?	(IMMV: showing class names without selecting classes, when selecting a square in the IMMV both names should be visible, zooming , folding / unfolding, when selecting/ hovering over summary pixels highlight related classes, smaller borders for selection frame ; PNLV: zooming, folding/unfolding)
4	What would you suggest for improving the underlying visualization approaches?	PNLV: edges-bundling
5	Which visualization technique is easier to understand? Why?	PNLV: it is easier to follow the edges.
6	Did you have any problems understanding particular aspects of the visualizations?	No
7	Which are the most interesting insights into the software system you found?	(IMMV: two similar classes (3); PNLV: semantic similarities, but no relations in the code itself (6), logging available but not used(5)).
8	Which of the insights of (7) would you have with other tools or without tool support?	IMMV: diffTool (Input of two classes with the same name). In general hard to discover.
9	Which representation do you think is more convenient to read? why?	PNLV: see (5)
10	Did you have any problems reading the visualization due to some form of visual impairment (color blindness, blurred vision, etc.) If yes , please specify?	No
11	Would you like to use the tool in the future for your daily software development work?	(PNLV: yes, to improve my code, package structure, etc. ;IMMV: No, for smaller projects PNLV is easier to use and sufficient).
12	In your opinion, what is the most promising area of application for the visualization techniques?	(PNLV: improving the architecture/design, IMMV: improving the architecture/design).
13	Did you use similar visualization tools in your work? Please state them?	No

Table A.8: Results of the final questionnaire

Appendix B

Second Pretest

No	Question	Participant Answer	Correct Answer	Result
1	Which different types of dependencies exist *from* (org.jhotdraw.gui.datatransfer. AWTClipboard) *to* (org.jhotdraw.gui. datatransfer.AbstractClipboard)? please consider the direction.	Inheritance, semantic, and evolution	Inheritance, semantic, and evolution	correct
2	Is there any code clone dependency between the files of package (org.jhotdraw.gui.datatransfer) and (org.jhotdraw.gui.plaf) in any direction?	No	No	correct

Table B.1: Trial results for the PNLV using the JHotDraw dataset

No	Question	Participant Answer	Correct Answer	Result
1	Which different types of dependencies exist *from* (org.jhotdraw.gui. fontchooser.FontFamilyNode) *to* (org.jhotdraw.gui. fontchooser. FontCollectionNode)? please consider the direction.	Usage, aggregation, code-clone, semantic	Usage, aggregation, code-clone, semantic	correct
2	Is there any evolutionary coupling between the classes or interfaces of package (org.jhotdraw.gui.datatransfer) and (org.jhotdraw.gui.event) in any direction?	No	No	correct

Table B.2: Trial results for the IMMV using the JHotDraw dataset

Dataset	ID	IMMV Explanation
Stripes	1	There are clusters within tag package. They have different couplings such as code clones, semantic similarity, usage, and evolutionary.
	2	Classes have only code clones with few exceptions.
	3	There is cluster within the layout package. It has semantic similarity and evolutionary couplings.
	4	There are clusters within the bean package. They have evolutionary couplings but also with other packages controls and config.
	5	There is vertical line with respect to util package. It has usage and aggregation.

Table B.3: Results of the IMMV paper test

Dataset	ID	PNLV Explanation
Checkstyle	1	Many classes from the API package are extended by others.
	2	API classes are aggregated similarly.
	3	API classes are used similarly.
	4	There is high semantic similarity in the indentation package.
	5	There is high semantic similarity.
	6	There is similar structure to 4 and 5 for evolutionary coupling.
	7	There are few incoming inheritance edges for many classes.
	8	There is x shape visual structure in code coupling and semantic similarity.
	9	There is difference between code coupling and semantic similarity.
	10	API package co-changes with others.
	11	All of the are in general not very helpful.
	12	Semantic similarity is similar to evolutionary coupling.

Table B.4: Results of the PNLV paper test

ID & Time	Exploration Process	Finding
4 Start 15:22 End 15:28	Selecting bean package	There is some evolutionary coupling in the bean package for every classes. This confirms previous hypothesis (no explanation).
1 Start 15:29 End 15:39	Selecting different classes	There are similar names of the classes but there are no inheritance couplings within the packages.
5 Start 15:39 End 15:41	Selecting the util package then selecting class log	log class is used all over the project. There are usage and aggregation couplings.

Table B.5: Results of the interactive IMMV test

ID & Time	Exploration Process	Finding
1 Start 16:46 End 16:38	Selecting classes with high fan-in. Looking at the colored lines	APICheck and AbstractFile-SetCheck are important classes.
9 Start 16:39 End 16:46	Highlighting the coding package. Selecting a class that is connected to others by code clone but not by semantic similarity . Opening two files in editor. He cannot find code clone. The vocabularies are not similar.	There are more code clones within the package than semantic similarity couplings. There are unusual, critical different vocabularies in clones.
6 Start 16:47 End 16:51	Selecting indentation package and naming package	Indentation package is independent with respect to development. Classes are also similar. Refactoring is needed. Similar for naming package but more code clone couplings to other packages

Table B.6: Results of the interactive PNLV test

No	Question	Answer
1	Gender	Male
2	Age	29
3	Your highest degree obtained up to now	Master
4	Your major subject	Computer science
5	Number of Software Engineering courses you attended	5
6	Years of experience in Programming	15
7	Programming languages you use regularly	Java
8	Number of classes/ files that were in the largest project you worked on:	51-100 classes/files
9	Size of team/group you usually develop software together:	Alone
10	Which visual representations or visualization tools do you regularly use for software development:	None
11	Did you have any problems reading the visualization due to some form of visual impairment (color blindness, blurred vision, etc.) if yes, please specify.	No

Table B.7: Results of the general questionnaire

No	Question	Answer
1	What do you like and don't you like in both visualizations?	(IMMV) multiple dependencies (in one direction) can be seen in one cell, difficult to find the dependences into the other direction, needs a lot of spaces (scrollbars), strongly needs color legend. if you select a single class you get a quick overview about other dependencies classes with multiple dependencies types. (PNLV) easier to explore correlations between the dependency types. different dependency directions can be seen very fast.. in detail if is difficult to check if several classes with the same dependency type also have other dependencies to each other. selecting single classes is hard(very small rectangles)
2	Which visualization technique do you prefer for analyzing software projects? Why?	(PNLV) I had the feeling that it is more intuitive for the IMMV vis . I had a much steeper learning curve . I don't know if the node links vis also works in larger projects but in every moment i thought that i had a good overview of the project.
3	The visualization is useful for analyzing software projects:	(IMMV) Neutral. (PNLV) Agree.
4	To enhance the tool, which interaction features should be added?	(IMMV) if you select cell $a \rightarrow b$, $b \rightarrow a$lso should slightly be selected. Don't use tooltips, they are to slow. selecting packages horizontally and vertically at the same time. (PNLV) merge different dependency graphs.
5	What would you suggest for improving the underlying visualization approaches?	(IMMV) perhaps it would be helpful to hide on demand the upper or lower triangle to reduce redundant information.
6	Which visualization technique is easier to understand? Why?	(PNLV) because the dependency direction follows the reading direction and in the IMMV vis. it is confusing that you have redundant information because of the duplicated hierarchy.
7	Which are the most interesting insights into the software system you found?	(IMMV) core packages which have many dependencies to other packages. code clones and semantic similarity similar fragment don't correlate much. (PNLV) core packages which have many dependencies to other packages. a package which could outsourced. a package which could be refactored.

8	Would you like to use the tool in the future for your daily software development work?	(PNLV) yes, I was surprised about some assumptions I have found . it would be interesting to use this on code I know. (IMMV) No, I only use visualization very rarely and thus an self-explaining tool would be better to me. this visualization has a steep learning curve.
9	In your opinion, what is the most promising area of application for the visualization techniques?	(PNLV) program comprehension, debugging/bug fixing, and improving the architecture. (IMMV) program comprehension and improving the architecture.

Table B.8: Results of the final questionnaire

Curriculum Vitae of Ala Abuthawabeh

Personal Details

Name	Abuthawabeh
First Name	Ala
Title	M.Sc.
Work Address	University of Kaiserslautern FB Informatik Postfach 3049 67653 Kaiserlautern

Academic Record

2005-2007	Master's degree, Computer Information Systems, Arab Academy for Banking and Financial Sciences
2001-2005	Bachelor's degree, Computer Information Systems, Al al-byte University

Professional Record

2009-present	PhD student, Computer Graphics and HCI Group, University of Kaiserslautern
2009	Software Developer, Al al-byte University
2007-2009	Lab Supervisor, Al al-byte University
2005-2007	Computer Science Teacher, Ministry of Education Jordan

Publications

[1] Ala Abuthawabeh and Dirk Zeckzer. SMNLV: A Small-Multiples Node-Link Visualization Supporting Software Comprehension by Displaying Multiple Relationships in Software Structure, 2015.

[2] Ala Abuthawabeh and Dirk Zeckzer. An Improved Decomposition and Drawing Process for Optimal Topological Visualization of Directed Graphs. In *Proceedings of the 31th Spring Conference on Computer Graphics*, SCCG'15, pages 111-118. ACM, 2015.

[3] Ala Abuthawabeh, Fabian Beck, Dirk Zeckzer, and Stephan Diehl. Finding Structures in Multi-Type Code Couplings with Node-Link and Matrix Visualizations. In *Proceedings of the First IEEE Working Conference on Software Visualization 2013*, VISSOFT 2013. IEEE Computer Society, 2013.

[4] Ala Abuthawabeh and Dirk Zeckzer. IMMV: An Interactive Multi-Matrix Visualization for Program Comprehension. In *Proceedings of the First IEEE Working Conference on Software Visualization 2013*, VISSOFT 2013. IEEE Computer Society, 2013.

[5] Ala Abuthawabeh. Software matrix layout. Master's thesis, University of Kaiserslautern, Germany, 2012.

January 27, 2016

Erklärung

Ich versichere, dass ich die vorliegende Dissertation selbst angefertigt und alle von mir benutzten Hilfsmittel in der Arbeit angegeben habe, daß ich weder die Dissertation selbst noch Teile hiervon als Prüfungsarbeit für eine staatliche oder andere wissenschaftliche Prüfung eingereicht habe, und dass ich die gleiche oder eine andere Abhandlung nicht bei einem anderen Fachbereich oder einer anderen Universität als Dissertation eingereicht habe.

Kaiserlautern, den 18. Oktober 2015
Ort, Datum

Unterschrift