# CHISIO: A VISUAL FRAMEWORK FOR COMPOUND GRAPH EDITING AND LAYOUT

A THESIS

SUBMITTED TO THE DEPARTMENT OF COMPUTER ENGINEERING

AND THE INSTITUTE OF ENGINEERING AND SCIENCE

OF BILKENT UNIVERSITY

IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF

MASTER OF SCIENCE

By

Cihan Küçükkeçeci

June, 2007

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Assoc. Prof. Dr. Uğur Doğrusöz(Advisor)

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Dr. Kıvanç Dinçer

I certify that I have read this thesis and that in my opinion it is fully adequate, in scope and in quality, as a thesis for the degree of Master of Science.

Dr. Markus Schaal

Approved for the Institute of Engineering and Science:

Prof. Dr. Mehmet B. Baray
Director of the Institute

# ABSTRACT

## CHISIO: A VISUAL FRAMEWORK FOR COMPOUND GRAPH EDITING AND LAYOUT

Cihan Küçükkeçeci
M.S. in Computer Engineering
Supervisor: Assoc. Prof. Dr. Uğur Doğrusöz
June, 2007

Graphs are data models, widely used in many areas from networking to biology to computer science. Visualization, interactive editing ability and layout of graphs are critical issues when analyzing the underlying relational information.

There are many commercial and non-commercial graph visualization tools. However, overall support for compound or hierarchically organized graph representations is very limited.

We introduce a new open-source editing and layout framework named Chisio for compound graphs. Chisio is developed as a free, easy-to-use and powerful academic graph visualization tool, supporting various automatic layout algorithms. It is written in Java and based on Eclipse's Graphical Editing Framework (GEF).

Chisio can be used as a finished generic compound graph editor with standard graph editing facilities such as zoom, scroll, add or remove graph objects, move, and resize. Object property and layout options dialogs are provided to modify existing graph object properties and layout options, respectively. In addition, printing or saving the current drawing as a static image and persistent storage facilities are supported. Saved graphs or GraphML formatted files created by other tools can be loaded into Chisio. Furthermore, a highlight mechanism is provided to emphasize subgraphs of users interest.

The framework has an architecture suitable for easy customization of the tool for end-users' specific needs as well. Also Chisio offers several layout styles from the basic spring embedder to hierarchical layout to compound spring embedder to circular layout. Furthermore, new algorithms are straightforward to add, making Chisio an ideal test environment for layout algorithm developers.

*Keywords:* information visualization, graph layout, graph editing, software system, graph editor, compound graphs.

# ÖZET

# CHISIO: BİLEŞİK ÇİZGE DÜZENLEMESİ VE YERLEŞTİRMESİ İÇİN GÖRSEL BİR ÇERÇEVE

Cihan Küçükkeçeci
Bilgisayar Mühendisliği, Yüksek Lisans
Tez Yöneticisi: Doçent Dr. Uğur Doğrusöz
Haziran, 2007

Çizgeler, ağ oluşturmaktan, biyolojiye, bilgisayar bilimlerine, birçok alanda sıkça kullanılan veri modelleridir. Görselleştirme, etkileşimli düzenleme kabiliyeti ve çizgelerin yerleştirilmesi, temeldeki ilişkisel bilgilerin çözümlenmesinde kritik konulardır.

Birçok ticari ve ticari olmayan çizge görselleştirme araçları mevcuttur. Ancak bileşik veya sıradüzensel olarak düzenlenmiş çizge gösterimini sağlayan araç sayısı çok sınırlı sayıdadır.

Biz adı Chisio olan yeni bir açık-kaynak düzenleme ve yerleştirme yapan çerçeve tanıtıyoruz. Chisio bedava, kullanımı kolay ve etkili, otomatik yerleşim algoritmalarını destekleyen bir akademik çizge görselleştirme aracı olarak geliştirildi. Java'da yazıldı ve Eclipse'in Grafiksel Düzenleme Çerçevesi'ni (GEF) temel olarak aldı.

Chisio yakınlaştırma, kaydırma, çizge nesneleri ekleme ya da çıkarma, hareket ettirme, boyutunu değiştirme gibi standart çizge düzenleyici özellikleri ile tamamlanmış özelleşmemiş bileşik çizge düzenleyicisi olarak kullanılabilir. Var olan çizge nesnelerinin özelliklerini ve yerleşim seçeneklerini değiştirmek için nesne özellikleri ve yerleşim seçenekleri pencereleri sunulmaktadır. Ayrıca, kullanılmakta olan çizgeyi yazdırma ya da resim olarak ve kalıcı bellekte saklamak da mümkündür. Saklanan çizgeler ya da diğer araçlar tarafından yaratılmış GraphML biçimindeki dosyalar Chisio'ya yüklenebilir. Bunlardan başka, vurgulama mekanizması ile kullanıcıların ilgilendiği altçizgelerin vurgulanması da mümkündür.

Çerçeve, son kullanıcıların özel ihtiyaçları için kolayca özelleştirilebilmeye olanak sağlayacak bir mimariye de sahiptir. Ayrıca Chisio yay gömmeli'den sıradüzensel yerleşime, bileşik yay gömmeliye, dairesel yerleşime çeşitli yerleşim

stilleri sunmaktadır. Bundan başka, yerleşim algoritması geliştirenler için Chisio'yu ideal bir sınama ortamı yapacak şekilde yeni algoritmalar doğruca eklenebilir.

*Anahtar sözcükler*: bilgi görselleştirme, çizge yerleşimi, çizge düzenleme, yazılım sistemi, çizge düzenleyici, bileşik çizgeler.

# Acknowledgement

I would like to express my deepest gratitudes to my supervisor Assoc. Prof. Uğur Doğrusöz, for his guidance and feedbacks during the preparation of this thesis. It has been a great experience and privilege for me to work with him and get benefit from his valuable mentorship.

I also would like to thank Dr. Kıvanç Dinçer and Dr. Markus Schaal for reviewing the manuscript of this thesis and spending their valuable time.

Above all, I am very grateful for the endless love and support of my parents Gülşen and Muhsin Küçükkeçeci, and my dearest brother, Onur and sister, Deniz. I feel stronger and happier with their love.

# Contents

**1 Introduction** **1**

1.1 Motivation . . . . . 5

1.2 Background Information . . . . . 5

1.2.1 Graphs . . . . . 5

1.2.2 Graph Drawing . . . . . 6

1.2.3 Graph Editing Framework (GEF) . . . . . 8

**2 Related Work** **10**

2.1 Commercial Tools . . . . . 10

2.2 Non-Commercial Tools . . . . . 13

**3 Chisio Architecture** **16**

3.1 The Model-View-Controller (MVC) Architecture . . . . . 18

3.2 Chisio Models . . . . . 19

3.3 Chisio Figures . . . . . 20

3.4 Chisio Editparts . . . . . 22

3.5 Chisio Actions and Commands . . . . . 24
3.6 Inspector Windows . . . . . 24
3.7 Highlight Mechanism . . . . . 26
3.8 Animation in Layout . . . . . 27
3.9 Use Cases . . . . . 28

**4 Chisio as an Editor** **30**

4.1 Drawings . . . . . 30
4.1.1 Nodes . . . . . 31
4.1.2 Compound Nodes . . . . . 32
4.1.3 Edges . . . . . 32
4.2 Chisio Tools . . . . . 34
4.2.1 Select Tools . . . . . 34
4.2.2 Zoom Tools . . . . . 35
4.3 Changing Topology . . . . . 35
4.3.1 Creating Graph Objects . . . . . 37
4.3.2 Deleting Graph Objects . . . . . 38
4.3.3 Transferring Graph Objects . . . . . 39
4.3.4 Reconnecting an Edge . . . . . 40
4.4 Changing Geometry . . . . . 41
4.5 Highlighting . . . . . 43

4.6 Cluster IDs . . . . . 44
4.7 Persistent Storage . . . . . 45
4.8 Static Images and Printing . . . . . 47

**5 Chisio Customization 48**

5.1 Customize Editor . . . . . 48
5.1.1 Customizing User Interface . . . . . 48
5.1.2 File Operations . . . . . 60
5.1.3 Menu Operations . . . . . 65
5.2 Extendible Layout Architecture . . . . . 69
5.2.1 Chisio Layout Architecture . . . . . 69
5.2.2 Adding New Layout Algorithms . . . . . 69
5.2.3 New Layout Options . . . . . 72
5.2.4 Changing Defaults For Layouts . . . . . 75

**6 Layout in Chisio 77**

6.1 CoSE Layout . . . . . 77
6.1.1 CoSE Layout Steps . . . . . 78
6.1.2 Layout Options . . . . . 78
6.1.3 CoSE Layout Improvements . . . . . 79
6.2 Cluster Layout . . . . . 81
6.2.1 Cluster Layout Steps . . . . . 81

6.2.2 Layout Options . . . . . 81
6.2.3 Cluster Layout Details . . . . . 82
6.3 CiSE Layout . . . . . 83
6.3.1 CiSE Layout Steps . . . . . 83
6.3.2 Layout Options . . . . . 84
6.4 Circular Layout . . . . . 84
6.4.1 Circle Layout Steps . . . . . 85
6.4.2 Layout Options . . . . . 85
6.5 Spring Layout . . . . . 86
6.5.1 Understanding the Spring Layout Algorithm . . . . . 86
6.5.2 Layout Options . . . . . 87
6.6 Hierarchical Layout . . . . . 88
6.6.1 Hierarchical Layout Steps . . . . . 88
6.6.2 Layout Options . . . . . 89
6.6.3 Hierarchical Layout Improvements . . . . . 90

**7 Conclusion** **94**

7.1 Contribution . . . . . 94
7.2 Future Work . . . . . 95

# List of Figures


1.1 A computer network drawn using a compound graph (courtesy of Tom Sawyer Software) . . . . . 2

1.2 A financial chart (courtesy of Tom Sawyer Software) . . . . . 3

1.3 Same graph before (left) and after (right) layout . . . . . 3

1.4 Different layouts for the same graph . . . . . 4

1.5 Compound graph with inter-graph and intra-graph edges . . . . . 6

1.6 Same graph with different drawings . . . . . 7

1.7 GEF MVC Architecture . . . . . 8

1.8 GEF is on top of Draw2d and SWT . . . . . 9

3.1 A UML Class Diagram illustrating the compound graph model used in Chisio . . . . . 19

3.2 A UML Class Diagram illustrating the figure relations in Chisio . 21

3.3 Chisio system architecture . . . . . 22

3.4 A UML Sequence Diagram illustrating creation of a node with rectangle shape in Chisio . . . . . 23

3.5 A UML Sequence Diagram for running CoSE layout in Chisio . . 24

3.6 A UML Class Diagram illustrating the inspector dialogs in Chisio 25

3.7 Layers in GEF . . . . . 26

3.8 A UML Sequence Diagram for highlighting a node in Chisio . . . 27

3.9 A UML Use Case Diagram for Chisio as an editor . . . . . 28

3.10 A UML Use Case Diagram for inspector dialogs in Chisio . . . . 29

4.1 Basics of drawing in Chisio . . . . . 31

4.2 A node and its properties window or inspector . . . . . 31

4.3 A compound node (right) and its properties window . . . . . 32

4.4 Examples of edges with different styles (left) and available edge types (right) . . . . . 33

4.5 Edge properties window or edge inspector . . . . . 33

4.6 Graph properties window . . . . . 33

4.7 Example selection handles for all types of graph objects . . . . . 34

4.8 Example use of the Marquee Selection Tool . . . . . 35

4.9 Before marquee zoom to the area specified by the red rectangle . . 36

4.10 After marquee zooming into the area as described by Figure 4.9 . 36

4.11 Graph pop-up menu includes zoom-in and zoom-out operations . 37

4.12 Create a node by simply clicking on the drawing canvas, where you would like your new node to be placed; before the creation (left) and after the creation (right) . . . . . 37

4.13 Create an edge by first clicking on the source node and then on the target node . . . . . 38

4.14 Remove compound; before and after . . . . . 38

4.15 Move/Transfer Mode combo . . . . . 39

4.16 When transferring a node into a compound node, the compound node's background color changes to indicate the transfer; initially (left), when the transfer of the selected node is about to be performed 39

4.17 Clone operation; drag selected nodes pressing the CTRL key in the transfer mode and click to clone . . . . . 40

4.18 Create compound (right) from selected nodes (left) . . . . . 40

4.19 Edge reconnection; after selection of the edge (left), during dragging (middle) and after reconnection (right) . . . . . 41

4.20 Move a node by selecting and dragging it . . . . . 41

4.21 Resize a node; before (left) and after (right) . . . . . 42

4.22 Compound nodes are automatically resized . . . . . 42

4.23 Bendpoint creation; after selection (left), during drag of the small handle (middle), and upon completion with the release of the mouse (right) . . . . . 42

4.24 Bendpoint deletion; initially the edge has two bends (left); upon drag and release of the left bend, it is deleted to leave the edge with only one bend (right) . . . . . 43

4.25 Highlighting objects . . . . . 43

4.26 Creating a new cluster . . . . . 44

4.27 New cluster ID of the node as confirmed by the node properties window . . . . . 44

4.28 Coloring with cluster IDs results in three clusters, each with a unique color . . . . . 45

4.29 A Highlighted node for example GraphML format . . . . . 46

4.30 An edge for example GraphML format . . . . . 47

4.31 Save as image menu item . . . . . 47

5.1 Sample UIs for two diamond shaped nodes . . . . . 51

5.2 Sample screenshot of the Edge Inspector after addition of the new attribute "Arrow Size" . . . . . 58

5.3 Chisio Layout Architecture . . . . . 70

6.1 Sample CoSE layout produced by Chisio . . . . . 78

6.2 CoSE layout options . . . . . 79

6.3 Create bendpoints as needed checkbox . . . . . 80

6.4 Bendpoints are automatically created for loop edges and multi edges 80

6.5 Sample Cluster layout produced by Chisio, where nodes are color-coded by their clusters . . . . . 81

6.6 Cluster layout options . . . . . 82

6.7 Sample CiSE layout produced by Chisio, where nodes are color-coded by their clusters . . . . . 83

6.8 CiSE layout options . . . . . 84

6.9 Sample Circular layout produced by Chisio . . . . . 85

6.10 Circular layout options . . . . . 86

6.11 Sample Spring layout produced by Chisio . . . . . 87

6.12 Spring layout options . . . . . 87

6.13 Sample Hierarchical layout produced by Chisio . . . . . 88

6.14 Hierarchical layout options . . . . . 89

6.15 Hierarchical layout with compound support . . . . . 91

6.16 Same graph without (top) and with (bottom) improvement in Hierarchical layout by pushing up nodes . . . . . 92

6.17 Non-uniform node size support in Hierarchical layout . . . . . 92

6.18 Same graph without (left) and with (right) bendpoints in Hierarchical layout . . . . . 93

# List of Tables

2.1 Chisio compared to popular commercial tools . . . . . 11

2.2 Chisio layout compared to popular commercial tools' layout . . . 12

2.3 Chisio compared to popular non-commercial tools . . . . . 14

2.4 Chisio layout compared to popular commercial tools' layout . . . 15

# Chapter 1

# Introduction

*Graphs* are simply a set of objects (nodes) and a set of relations (edges that connect nodes to each other) among them. By using this simple but powerful data structure many real world problems, from network to communication to data flow charts, can be modeled.

*Compound graphs* are graphs with nested child graphs inside. These graphs have compound nodes which have inner nodes and edges in its child graph (Figure 1.1 and 1.2). Multiple nesting can be implemented by adding a compound node into another compound node and so on.

*Graph visualization* is a research area for drawing graphs according to their topological information [15]. These drawings must be understandable for easy interpretation and analysis. At this point, geometrical information of the graph objects comes into prominence. Location of the nodes, style of the edges change the clarity of the drawing. So, nodes must be moved or resized by user interaction or automatically. This process is known as *graph layout*. By using automatic graph layouts, highly complicated graphs can be visualized in an understandable fashion in a few seconds (Figure 1.3).

Different applications may require different layout styles. For example, drawing a flow chart generally gives the best result with hierarchical layout. Because

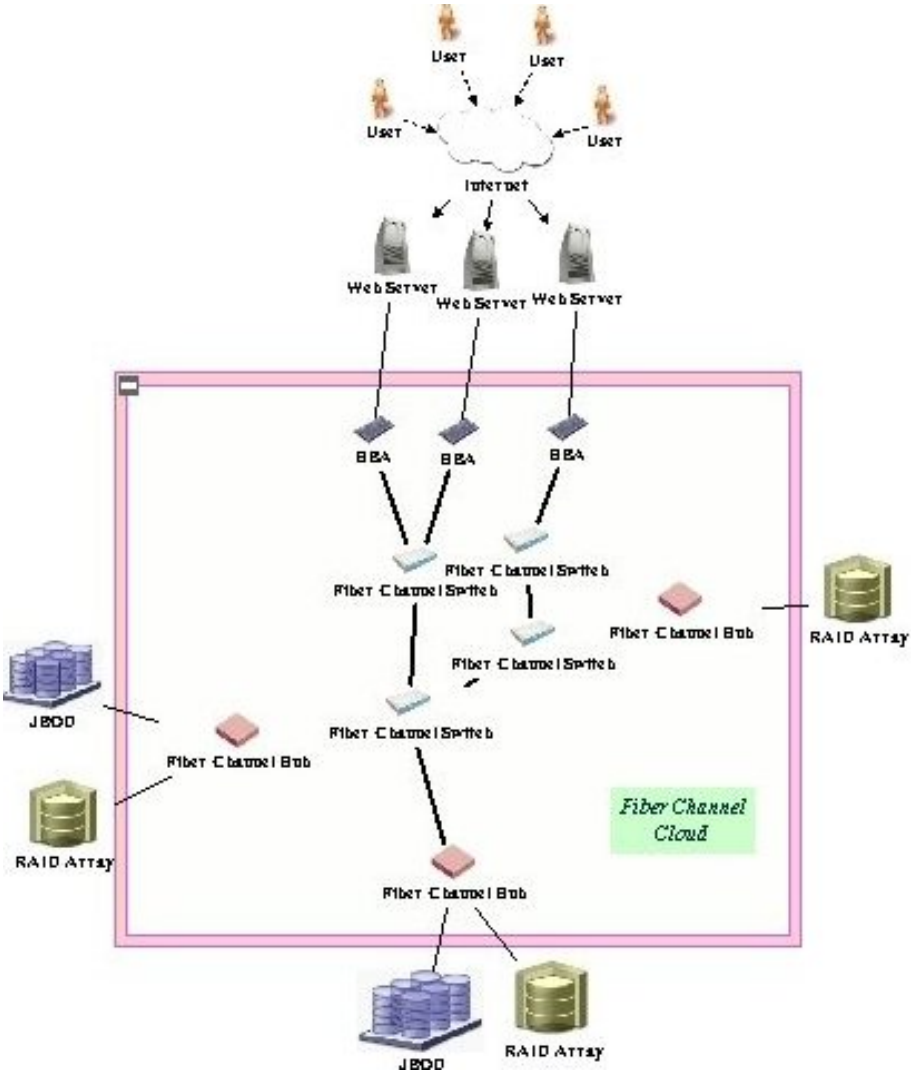


Figure 1.1: A computer network drawn using a compound graph (courtesy of Tom Sawyer Software)

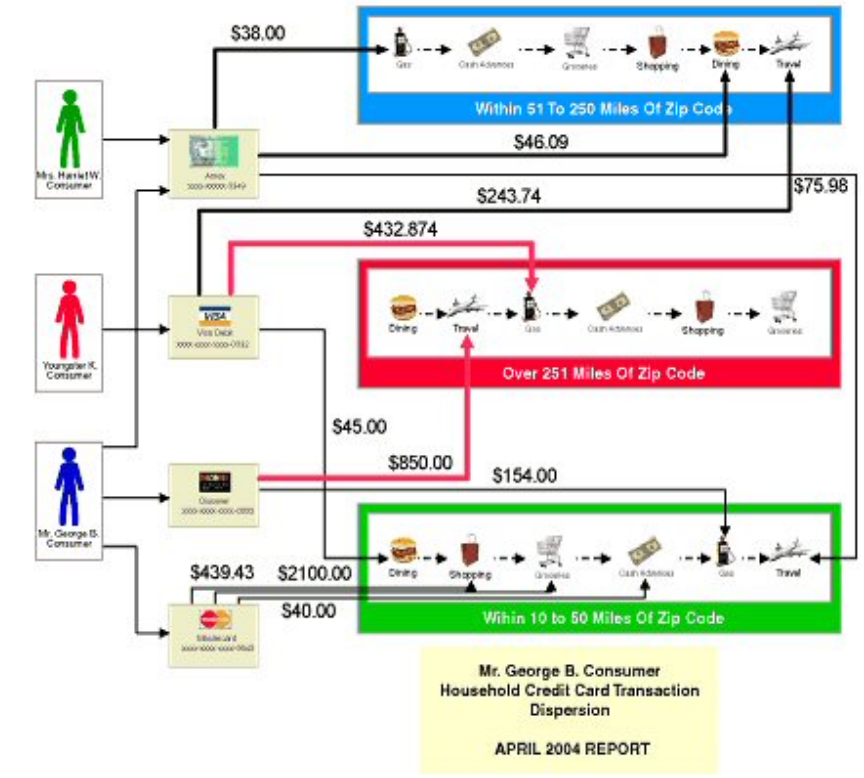


Figure 1.2: A financial chart (courtesy of Tom Sawyer Software)

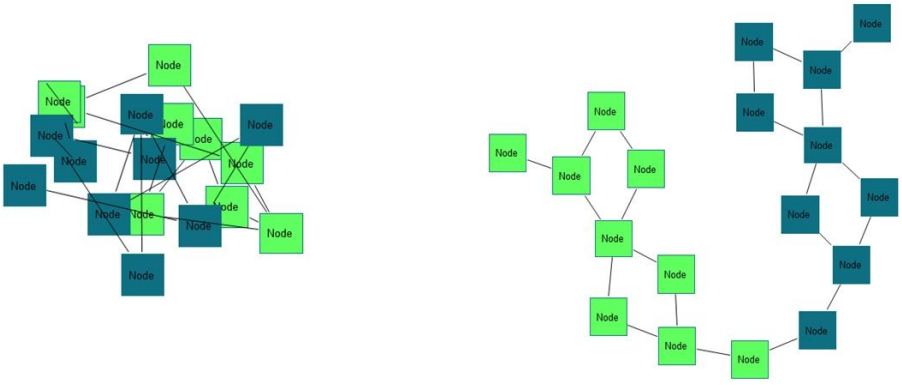


Figure 1.3: Same graph before (left) and after (right) layout

the relational information can be understood easily by using the level information that hierarchical layout reveals.

Hence for every particular graph application, we need specialized graph layout algorithms which consider the structural information and semantics of the graphs. This seems that we need a high number of algorithms in a generic graph layout tool. But there are several layout algorithms which are powerful enough to give best drawings in a wide range of applications. These are Radial, Circular, Hierarchical and Spring Embedder. Every one of them gives different results for the same graph (Figure 1.4).

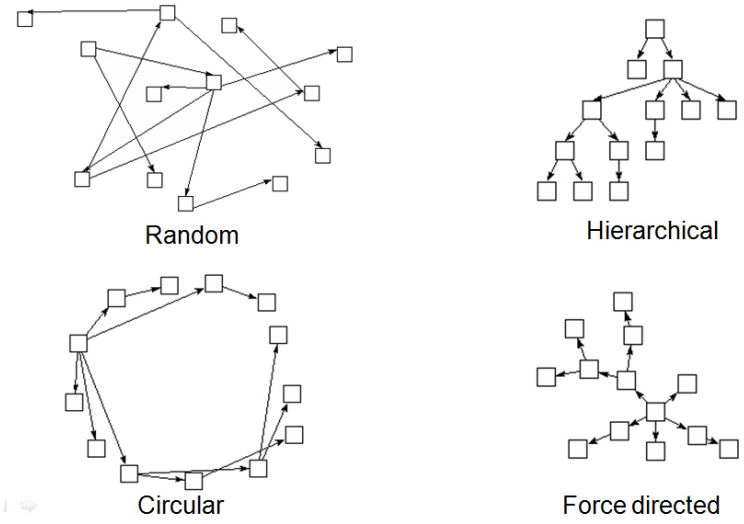


Figure 1.4: Different layouts for the same graph

Graph visualization tools are developed to display and manipulate graphs. Some of them only support display while others also support editing. Layout support is another crucial capability for these tools.

Compound support is another aspect for graph visualization tools. Managing compound graphs for both editing and layout operations is a main design constraint which must be considered from the beginning.

## 1.1 Motivation

Over the years, an abundant number of such tools have been made available for consumption both commercially and academically [6]. However, only a few, if any, non-commercial systems seem to address compound structures of graphs both in regards to editing and layout capabilities. Chisio should fill an important gap in this field.

Another motivation is creating an abstract layout mechanism making the development of new layout algorithms possible. For instance, with the help of step by step animation ability, layout phases can be tracked easily.

## 1.2 Background Information

### 1.2.1 Graphs

A graph $G = (V, E)$, is a set of nodes ($V$), and edges ($E$). An edge $(u, v) \in E$ connects the nodes, $u \in V$ and $v \in V$, where $u$ is the source and $v$ is the target.

If source and target nodes are the same for an edge $(u, v)$, then this edge is called a *loop edge*. If there are several edges between $u$ and $v$, then these edges are called *multi-edges*.

*Incoming edges* for a node is the list of edges that this node is the target node for. *Outgoing edges* for a node is the list of edges that this node is the source node for. Thus, an edge $(u, v)$ is in the outgoing edge list of node $u$ and in the incoming edge list of node $v$.

If a graph is a *directed graph*, edges are directional, from source node to target node. Every edge is in either incoming edges or outgoing edges for its source and target nodes.

In an *undirected graph*, an edge has no sense of direction. An undirected

graph can be represented by a directed graph if every undirected edge $(u, v)$ is represented by two directed edges $(u, v)$ and $(v, u)$. Generally, graphs are assumed to be undirected unless otherwise specified.

For compound graphs, a node can have a child graph in it called a compound node. There is a *parent-child* relationship between inner nodes of this compound node and itself. Let $G_1 = (V_1, E_1)$ and $G_2 = (V_2, E_2)$. If there is a compound node $c \in V_1$ and the child graph in it is $G_2$, an inner node $u \in V_2$ is the *child node* of $c$ and $c$ is the *parent node* of $u$ (Figure 1.5).

If an edge's source and target nodes belong to the same graph, then this edge is an *intra-graph edge*. Otherwise, this edge is called an *inter-graph edge* (Figure 1.5).

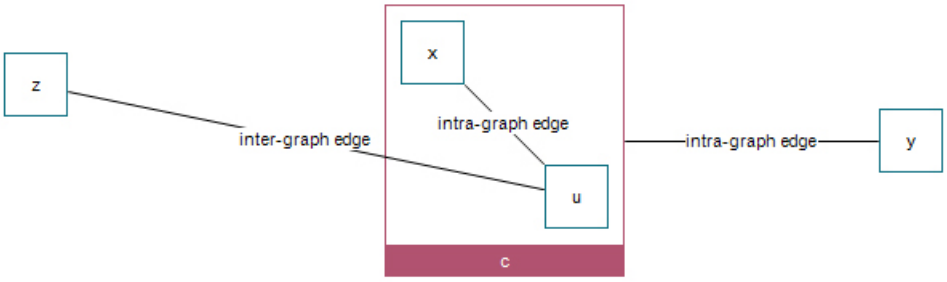


Figure 1.5: Compound graph with inter-graph and intra-graph edges

### 1.2.2 Graph Drawing

A graph $G = (V, E)$ has a topological information. Location, size and view of nodes cannot be extracted from this information. Drawing tools assign views for nodes and edges, add them geometrical information, make the graph more understandable by drawing it as a picture.

Simply put, *graph drawing* is a transformation from topological information to geometrical information. Graph $G = (V, E)$ is mapped to 2D or 3D objects. It is obvious that, a graph can have different drawings (Figure 1.6).

A node can be drawn as a point, rectangle, circle, etc. An edge can be drawn

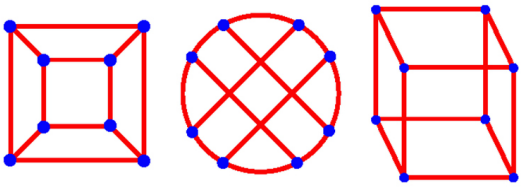

Figure 1.6: Same graph with different drawings

as a straight line, orthogonal polygonal path or arbitrary curve, etc. There can be label, color or extra view properties for the objects according to drawing.

If there is no edge crossings on the drawing then it is called *planar drawing*. If a graph can be drawn like *planar drawing*, then graph is a *planar graph*.

*Planarity* is one of the important issues for readability of the graphs. The lesser *edge crossings*, the more graph is readable. Because tracking of the edges is easier when there are not any edge crossings.

*Minimization of the drawing area* is another parameter for readability. When drawing area is maximized, graph must be zoomed out to be viewed completely. This will decrease the recognition of nodes, and also understandability.

*Total edge length* is important, too. It is hard to track long edges in a drawing.

*Symmetry* also increases the understandability of the drawings of some graphs. Sometimes, symmetry can be preferred instead of edge crossing minimization (Figure 1.6).

Generally these kinds of aesthetics are naturally associated with optimization problems, which are computationally hard. Testing planarity takes linear time while testing upward planarity is NP-hard. Minimizing crossings is an NP-hard problem, too. Minimizing bends in planar orthogonal drawing is NP-hard in general while it is polynomial time for a fixed embedding. Thus many approximation strategies and heuristics have been devised [5].

Above parameters and other aesthetic parameters increase the readability of the drawings. To maintain these constraints, nodes and edges must be located

carefully. That brings us to *layout* of the graphs and automatic layout algorithms are developed to maintain these parameters.

There are many kinds of graph applications to solve different real world problems. Best drawing for each one can change according to semantics and structural information of the graphs. For this purpose, many automatic layout algorithms like Radial, Hierarchical and Force-directed have been devised. These algorithms run with a best performance for a wide range of applications. In many cases, users may interact with the graphs to improve the drawings manually after running automatic layouts.

### 1.2.3 Graph Editing Framework (GEF)

Chisio was built on top of the Eclipse Graph Editing Framework (GEF) [9]. There is a model to store necessary information, a view to visualize that model and a controller to bridge model and figure. This architecture is called MVC (Model-View-Controller). When MVC is mapped to GEF, controllers are editparts, views are figures and models are models, as well (Figure 1.7).

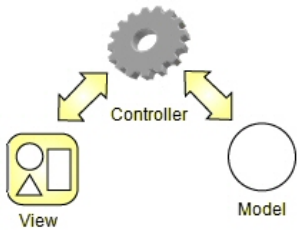


Figure 1.7: GEF MVC Architecture

*The Model* stores the data which is edited or viewed by the user. Because of the MVC architecture, model does not know either figure or editpart. Just a listener tracks the changes on the model and propagates them to editpart.

*The View* is the user interface part of the model. Each model has a view and user interacts with this view. When user changes the view, change is also transfered to the model.

*The Editpart* is the brain of the system. It connects the model and view and they do not know each other.

GEF is developed over Draw2d and SWT layer's of Eclipse. SWT is the base canvas, that objects can be drawn. Draw2d draws the objects on SWT, rendering and other graphic operations are done by this layer. And GEF is at top of both to give the interaction ability with drawn figures (Figure 1.8).

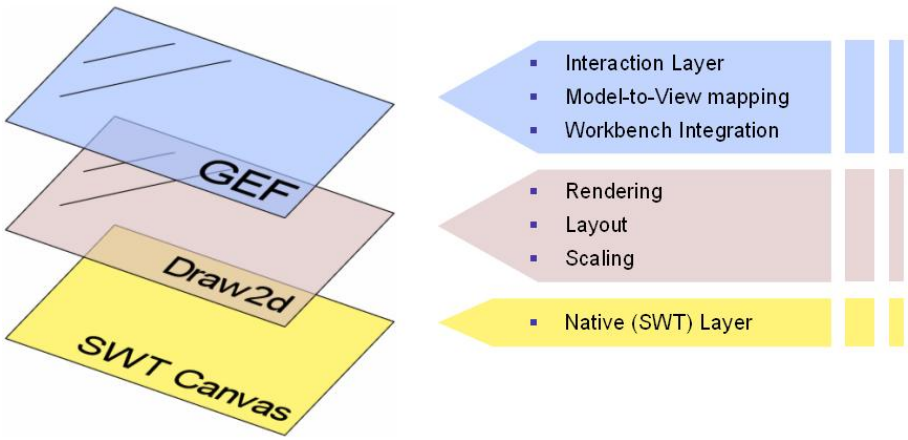


Figure 1.8: GEF is on top of Draw2d and SWT

# Chapter 2

# Related Work

There exist many commercial and non-commercial graph visualization tools. Tom Sawyer's TSV [17], yWorks' yFiles [20] and ILOG's JViews Diagrammer [13] are well-known commercial tools. Their capabilities and user interfaces are generally much better than non-commercial ones like uDraw [18], Graphviz [11], VGJ [19] and Graphlet [10]. Even though Chisio is a new academic tool, its features and capabilities are comparable to most non-commercial and some commercial tools.

## 2.1 Commercial Tools

Compound graph support, improved visualization facilities and several layout styles make Chisio a distinguished visualization tool. Here is a comparison of Chisio with some popular commercial tools (Table 2.1). It is certain that, capabilities of Chisio are comparable to these commercial tools.

Table 2.1 shows that most of commercial tools support zooming, scaling, basic editing, save/load operations and layout. Some of them have clustering facility and highlight support, too.

Table 2.2 shows that most layout styles are supported by almost all commercial tools. If a tool has compound support, compound layout is supported too.

| | Move/Resize/ Create/Delete | Zoom/ Scale | Compound Support | Highlighting | Save/ Load | Clustering/ Grouping | Object Inspector |
|---|---|---|---|---|---|---|---|
| TSV | + | + | + | + | + | + | + |
| JViews | + | + | + | + | + | + | + |
| yEd | + | + | + | - | + | + | + |
| aiSee | + | + | + | - | + | - | + |
| Chisio | + | + | + | + | + | + | + |

Table 2.1: Chisio compared to popular commercial tools

| | Radial | Circular | Hierarchical | Spring Embedder | Orthogonal | Compound | Cluster |
|---|---|---|---|---|---|---|---|
| TSV | + | + | + | + | + | + | + |
| JViews | + | + | + | + | + | + | + |
| yEd | - | + | + | + | + | + | + |
| aiSee | - | - | + | - | - | - | - |
| Chisio | - | + | + | + | - | + | + |

Table 2.2: CHISIO layout compared to popular commercial tools' layout

## 2.2 Non-Commercial Tools

In non-commercial arena, compound graph capability is rarely supported. Main motivation of Chisio is already this gap in this area. On the other hand, limited number of layout styles are implemented in non-commercial tools (Table 2.3). Generally, graph layout is done by using one of the well known graph layout algorithms.

Table 2.3 shows that, most of non-commercial tools do not support compound graphs, highlighting and clustering while basic operations are available.

Table 2.4 shows that, generally hierarchical layout is supported by non-commercial tools. Radial layout, spring embedded layout, orthogonal layout and circular layout are rarely supported. Compound layout and cluster layout are not addressed in the popular non-commercial tools that we compared Chisio to.

| | Move/Resize/ Create/Delete | Zoom/ Scale | Compound Support | Highlighting | Save/ Load | Clustering Grouping | Object Inspector |
|---|---|---|---|---|---|---|---|
| Graphviz | + | + | - | - | + | + | - |
| Prefuse | + | + | ± | - | + | - | - |
| Gravisto | + | + | - | - | + | - | + |
| JGraphEd | + | + | - | - | + | - | + |
| VGJ | + | + | - | - | + | + | + |
| GDToolkit | + | + | - | - | + | - | + |
| uDraw | + | + | - | - | + | - | + |
| VCG | - | + | - | - | + | - | - |
| Chisio | + | + | + | + | + | + | + |

Table 2.3: Chisio compared to popular non-commercial tools

| | Radial | Circular | Hierarchical | Spring Embedder | Orthogonal | Compound | Cluster |
|---|---|---|---|---|---|---|---|
| Graphviz | + | - | + | + | - | - | - |
| Prefuse | + | + | + | + | - | - | - |
| Gravisto | + | + | + | + | + | - | - |
| JGraphEd | - | - | - | - | - | - | - |
| VGJ | - | - | + | - | - | - | - |
| GDToolkit | - | - | + | - | + | - | - |
| uDraw | - | - | + | - | - | - | - |
| VCG | - | + | - | - | + | - | - |
| Chisio | - | + | + | + | - | + | + |

Table 2.4: CHISIO layout compared to popular commercial tools' layout

# Chapter 3

# Chisio Architecture

Chisio graph model is based on a simple graph definition as described in Section 3.2. It is designed neatly to make model structure understandable and customizable when necessary. One can easily extend these basic classes to create specialized node and edge types.

Figures have child-parent relation between them. This relation is specifically designed for easily customization of node UIs for various shape types. To customize a figure, just add new figures into it. Node shapes' figures are created inside `NodeFigure`. For new shapes, create a new figure similar to implemented shapes' figures and simply use it as a shape figure described in Section 5.1.1.1.

Menu items, pop-up menu items and toolbar items are all connected with an `Action` class. To customize the UI of Chisio, simply create new actions similar to already implemented ones and add them to top menubar, toolbar or pop-up menu as described in Section 5.1.3. Top menubar, pop-up menu and toolbar are packed into specific classes to easily understand what is going on in them.

Inspector windows for changing graph objects' UI properties are also designed carefully and packed into a basic abstract inspector class. Node, edge, compound and graph inspectors are extended from this basic class and make possible to simply create new items and add them to table. Height of the inspector window,

size of table, location of it, locations of buttons are all calculated automatically to make customization simple. New items are added easily as described in Section 5.1.1.3.

File loading and saving operations are also redesigned for easy customization. Abstract file reader and writer classes are implemented and all operations are packed into `GraphMLWriter` and `GraphMLReader` classes which are extended from these abstract classes. For example, `SaveAction` just creates an instance of `GraphMLWriter` class and calls overridden method to write the graph to xml file. These generic approaches simplify the customization of file operations as described in Section 5.1.2.

The l-structure described in layout architecture (Section 5.2.1) is another generic solution that is developed in Chisio to make it more customizable. Chisio graph model and l-structure are separated completely and l-structure is developed in a generic manner as it can be easily used in another application with another graph model. The connection between Chisio graph model and l-structure is established by `AbstractLayout` class. This class behaves like a bridge between them.

All layout algorithms in Chisio are developed for the l-structure and they all run on `LGraph`'s. Therefore, Chisio Layout Architecture can be decoupled from Chisio easily. This is an important point to emphasize the customization ability of Chisio layout.

Layout developers do not need to know Chisio graph model to develop new layout algorithms. They can implement their own algorithms by using generic l-structure which is a very basic implementation of compound graph data structure. Independent development of layout algorithms from Chisio model makes easier to implement new algorithms and they can be added to Chisio as described in Section 5.2.2 in a few steps.

## 3.1 The Model-View-Controller (MVC) Architecture

CHISIO is based on GEF which is built on the MVC architecture (Section 1.2.3). MVC is used to separate the user interface from functional part of the system. So that changes to the user interface do not affect data handling, and that the data can be reorganized without changing the user interface.

In MVC, business logic is the *model*, data representation is the *view*. Decoupling of them is done by using the *controller* concept.

- **Model:** The model object knows about all the data that need to be displayed. It represents enterprise data of an application and the business rules that govern access to and updates of this data. Model is not aware about the presentation of data and how that data will be displayed.
- **View :** The view represents the presentation of the application. The view object refers to the model. It is not dependent on the application logic. It remains same if there is any modification in the business logic.
- **Controller:** Whenever the user sends a request for something, it always goes through the controller. The controller is responsible for intercepting the requests from the view and passes it to the model for the appropriate action. After the action has been taken on the data, the controller is responsible for directing the appropriate view to the user. In GUIs, the views and the controllers often work very closely.

As CHISIO is an easily customizable tool, the MVC architecture is the right choice as an architectural pattern. For example, we can easily add new user interface properties like new shapes for nodes without changing the model part.

## 3.2 Chisio Models

In graphs, all the nodes, edges and compound nodes are graph objects. In compound graphs, a compound node is also a simple node. These simple facts have yielded the design in Figure 3.1 for our model structure.

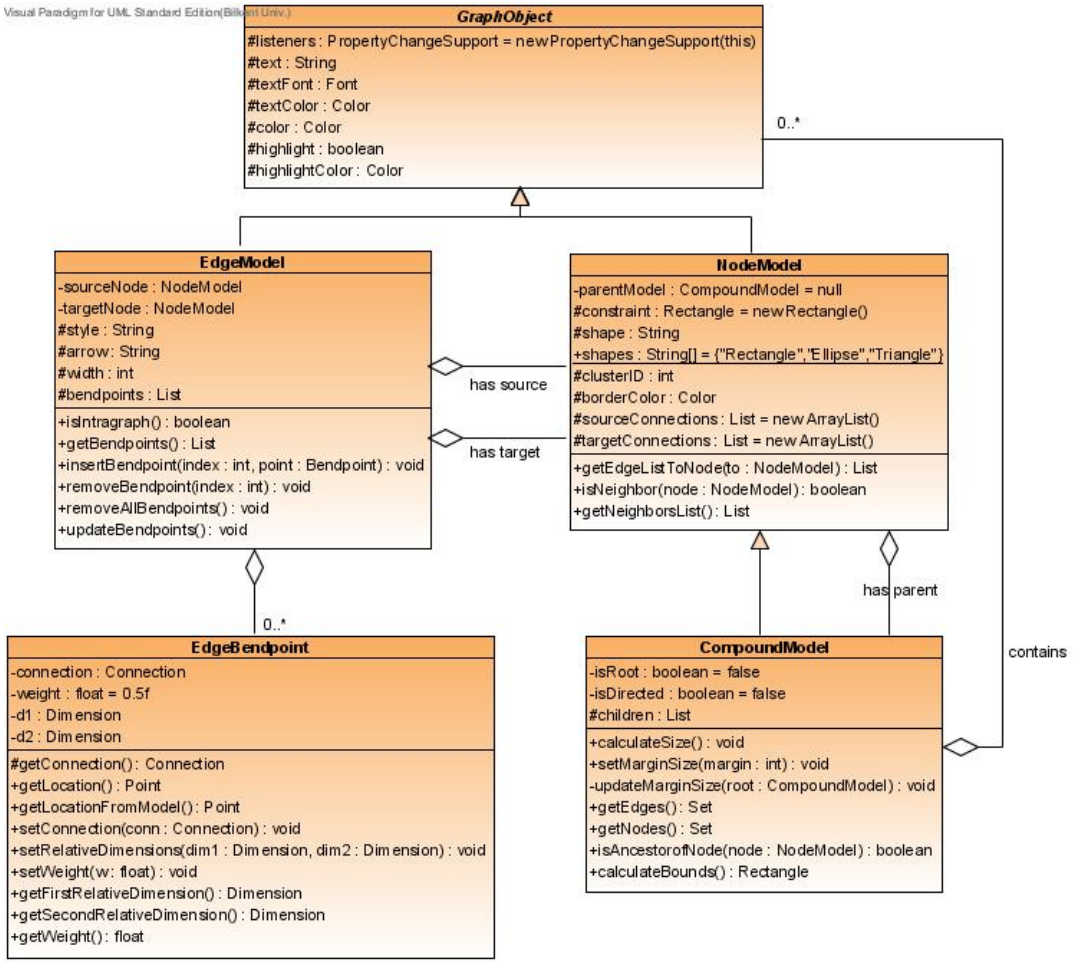


Figure 3.1: A UML Class Diagram illustrating the compound graph model used in Chisio

A compound node manages a list of child graph objects. A dedicated one is set as the root of the nesting hierarchy. Through recursive use of compound nodes as child objects, an arbitrary level of nesting can be created.

Each edge has a source node and a target node, while each node has lists of source and target connections which are the incoming and outgoing edges of the

node.

CHISIO has bendpoint support for flexible edge routing. An edge has a list of `EdgeBendpoint`'s to implement this feature.

## 3.3 Chisio Figures

In figure hierarchy, `EdgeFigure` and `NodeFigure` are separated. Because, `EdgeFigure` is extended from `PolylineConnection` while `NodeFigure` is just a `Figure`. So they are not connected unlike the way they are in the model.

`CompoundFigure` is extended from `NodeFigure` to save the relational information between them as done in model. It just overrides some methods to disable the abilities that are inherited from `NodeFigure` like `updateShape` method.

For available shape types for a node like rectangle, circle and triangle, there exist `Figure` classes to draw these shapes. In CHISIO, they were implemented as inner classes of `NodeFigure` class but they could be implemented separately as well (Figure 3.2).

There is a hierarchical structure of figure classes in GEF. A figure has a list of child figures. By adding a figure into the children list of another figure, they can be merged. For example, `NodeFigure` should normally have two children. One of them is for the text and the other for the shape. To draw a node with rectangular shape, we create a `Label` and a `RectangleFigure` then add them to children list of `NodeFigure`.

To easily add new shapes, new figure classes which draw those shapes must be created and added to the node figure.

The above structure is similar for edges. To add new properties to edge UI, child-parent relation is used.

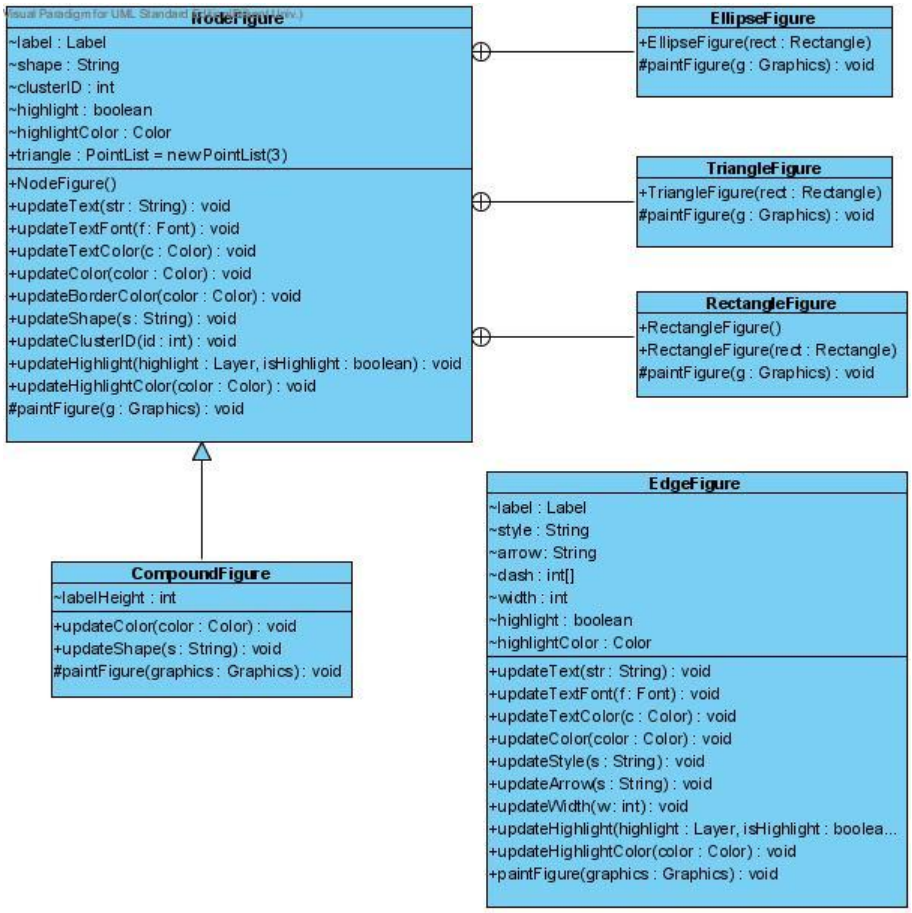


Figure 3.2: A UML Class Diagram illustrating the figure relations in CHISIO

## 3.4 Chisio Editparts

Editparts are the controllers and they behave as a bridge between nodes and figures. When a new object is created, first, corresponding editpart is created by the editpart factory. Then, newly created editpart creates the related figure for this object (Figure 3.3).

The sequence diagram for creating a new node with default values is explained in Figure 3.4. In this scenario, "Create Simple Node" menu item is selected and mouse is clicked on the editor to create a new node at specified location. Default shape is rectangle so `RectangleFigure` is created for new node.

Editparts process and respond to events, typically user actions, and may invoke changes on the model. For example, when user interacts with the user interface in some way (e.g., resizes a node), corresponding editpart for that node handles the input event from the user interface, via its registered listeners. Then, interacted node's model is updated in a way appropriate to the user's action.

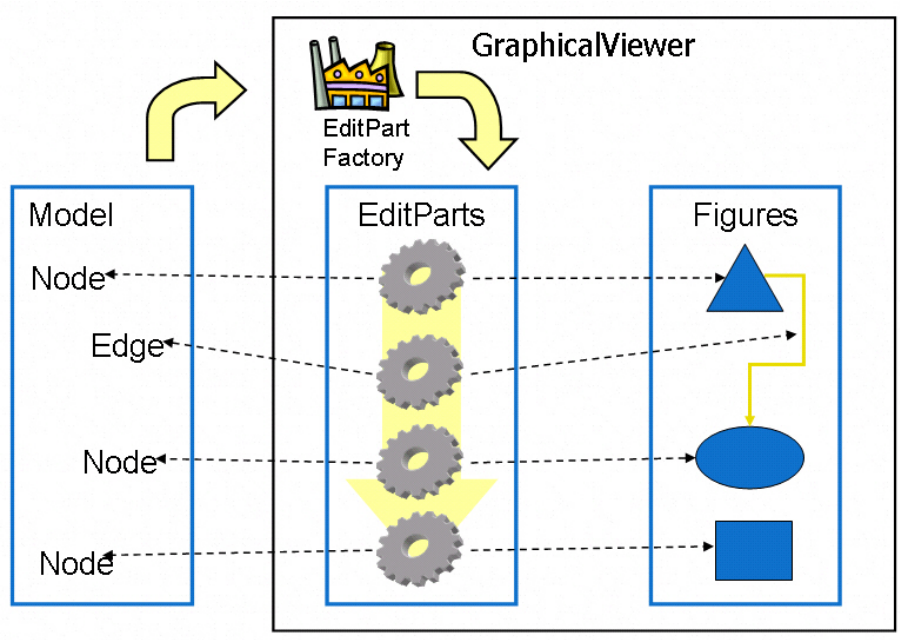


Figure 3.3: Chisio system architecture

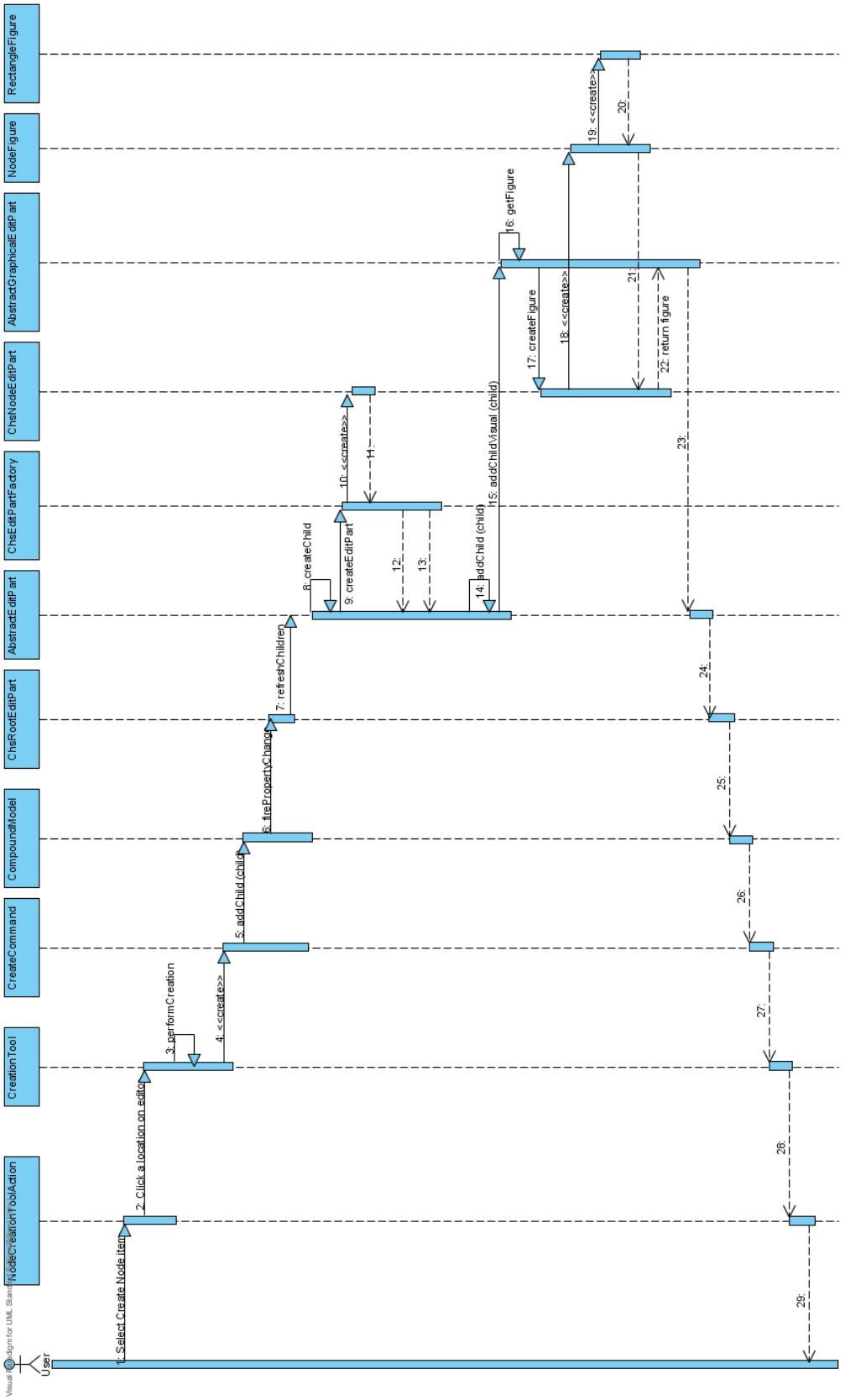


Figure 3.4: A UML Sequence Diagram illustrating creation of a node with rectangle shape in Chisio

## 3.5 Chisio Actions and Commands

The editor UI of CHISIO has a top menubar, toolbar and pop-up menus. All the items in these menus are connected with an `Action` to do the related operations.

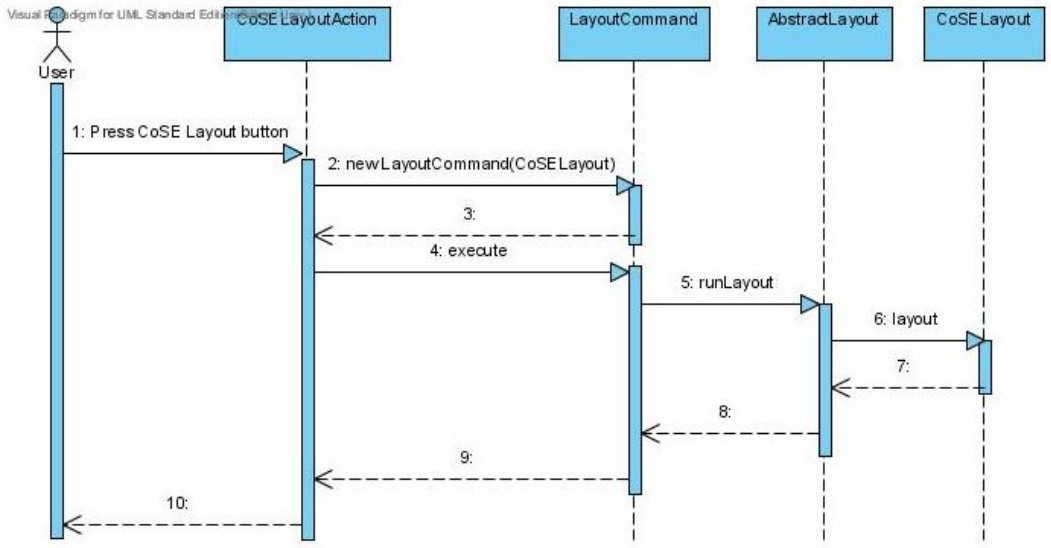


Figure 3.5: A UML Sequence Diagram for running CoSE layout in CHISIO

Generally an action creates and executes a corresponding `Command` to do the operation. For example, when CoSE layout is run by pressing the "CoSE Layout" button on toolbar, `CoSELayoutAction` is called. This action executes the `LayoutCommand` to run the CoSE layout algorithm. Corresponding scenario's sequence diagram is in Figure 3.5

## 3.6 Inspector Windows

There are many properties dialogs in CHISIO like "Node Properties", "Edge Properties", "Compound Properties", "Graph Properties" and "Layout Properties". All these dialogs, except "Layout Properties" have similar interface. There is a table that can include items like text, combo, numeric text, font and color.

We have designed a basic inspector dialog with an interface which makes

possible to insert new items easily. Graph objects' inspectors are extended from this basic class (Figure 3.6). The height of the dialog, locations of the table and buttons are all calculated in the basic inspector class according to the number of items in the table. Detailed information can be found in Section 5.1.1.3.

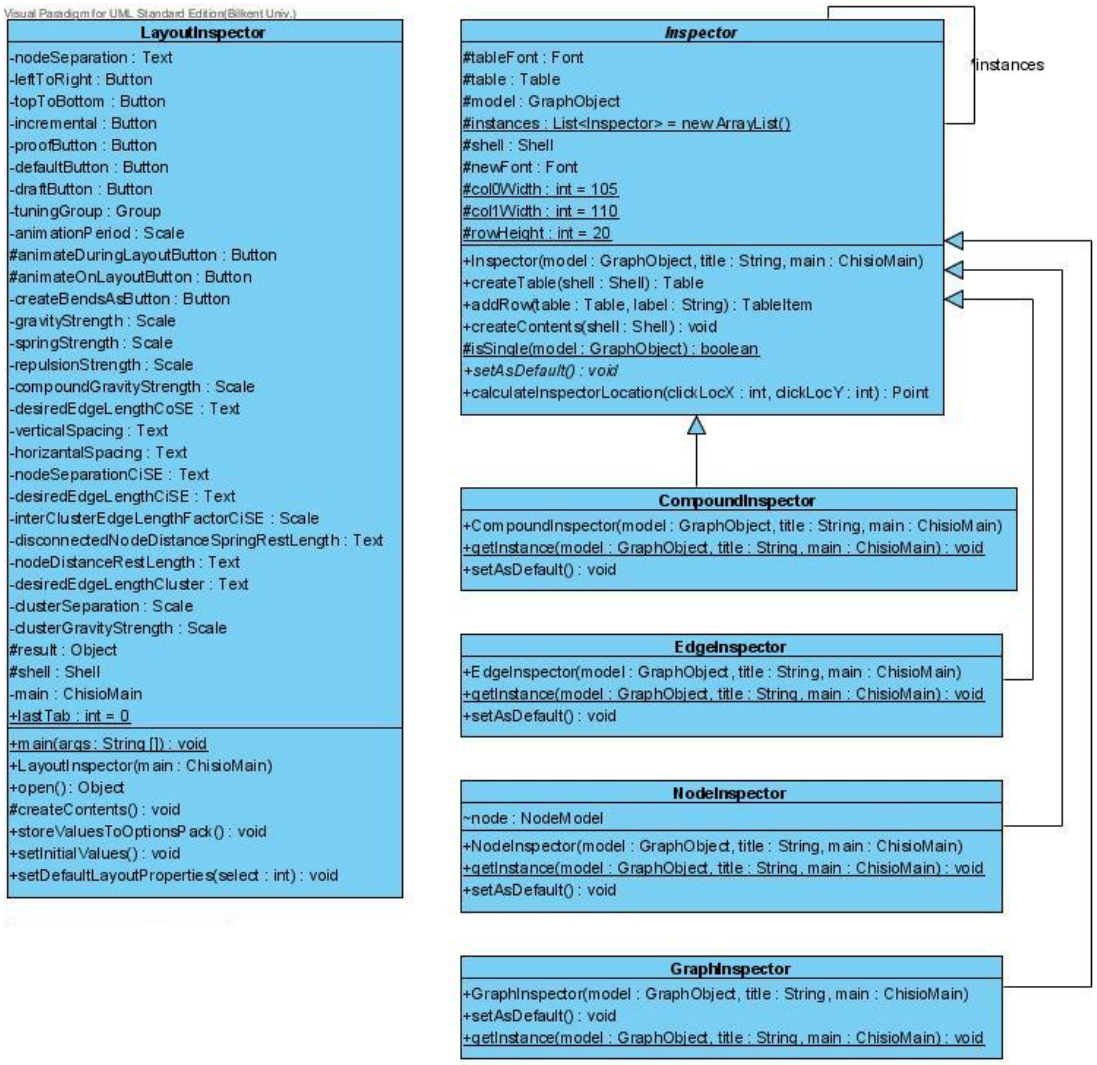


Figure 3.6: A UML Class Diagram illustrating the inspector dialogs in CHISIO

Each graph object can have only one inspector dialog. So we have used Singleton [8] design pattern in inspector dialogs. All instance are kept in a static list in the basic inspector class to track the inspector dialogs' lifetimes.

## 3.7 Highlight Mechanism

Several layers have been defined in GEF. The structure of layers (top-to-bottom) for root is shown in Figure 3.7.

Highlight mechanism in CHISIO to emphasize the user's subgraph of interest is added by creating our own highlight layer. Our new layer is extended from the basic class `Layer`. Layers are just a transparent figure intended to be added exclusively to a `LayeredPane`, who has the responsibility of managing its layers.

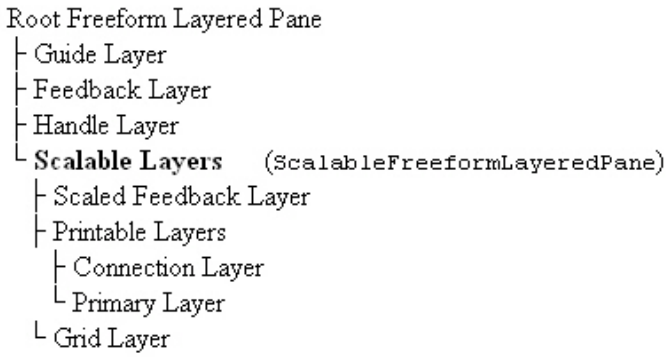


Figure 3.7: Layers in GEF

Scalable layers has the scaling ability. Because of the zooming and scaling support in CHISIO we have inserted our new highlight layer under Scalable Layers. When zoom level is changed, highlight layer is automatically scaled according to the new scale.

Also, we have added the highlight layer before all the other Scalable Layers. Because other layers may require user interaction (e.g connection layer which draws the edges), which highlight layer does not require. It will just draw a highlight effect around the highlighted graph object.

For both of the nodes, compound nodes and edges, there are two highlight figures to draw the highlighting effect around the objects. They are defined as a subclass in `HighlightLayer`. For example, when a node is highlighted, a new `HighlightNodeFigure` is created for highlighted node and added into the `HighlightLayer` (Figure 3.8).

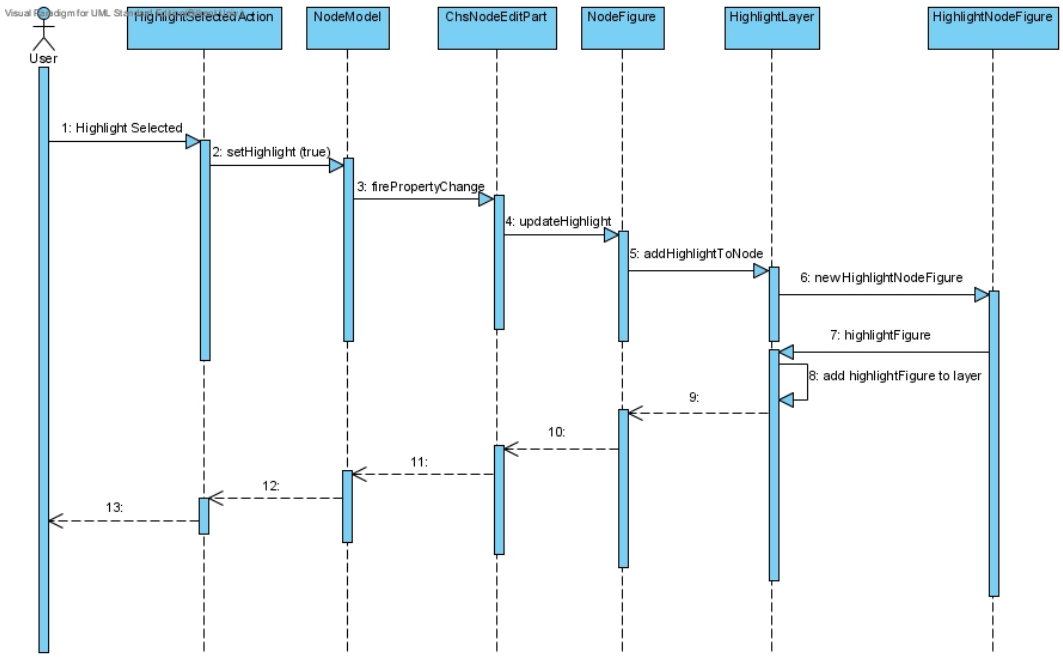


Figure 3.8: A UML Sequence Diagram for highlighting a node in CHISIO

## 3.8 Animation in Layout

In layout operations, animation feature is supported at all. Nodes and edges "smoothly" slide to their new locations with this support. This feature is basically achieved by a step-by-step updating of the view.

When animation is enabled, locations of objects before and after layout are kept. At the end of layout, while updating the view according to model, update process is done step-by-step with some time delays to give an animation effect to the view.

Also there is an extended version of animation which makes possible to track the nodes' moves in layout steps, called "Animation during layout". It is useful for tuning and debugging iterative layout algorithms such as spring embedders.

Animation during layout is achieved by creating a new method, `animate`, and calling it from the layout steps that we wish to track. `animate` method simply updates the view by using animation period parameter. Until the period time is

not completed, view does not updated. By changing the period parameter, the time delay can be decreased to track layout steps more detailed.

## 3.9 Use Cases

CHISIO is a graph editor for end-users with many features. Zooming, highlighting, clustering, GraphML file loading and saving, basic operations like create, delete and move are some of them. All capabilities of CHISIO are shown in the Use Case Diagram in Figure 3.9.

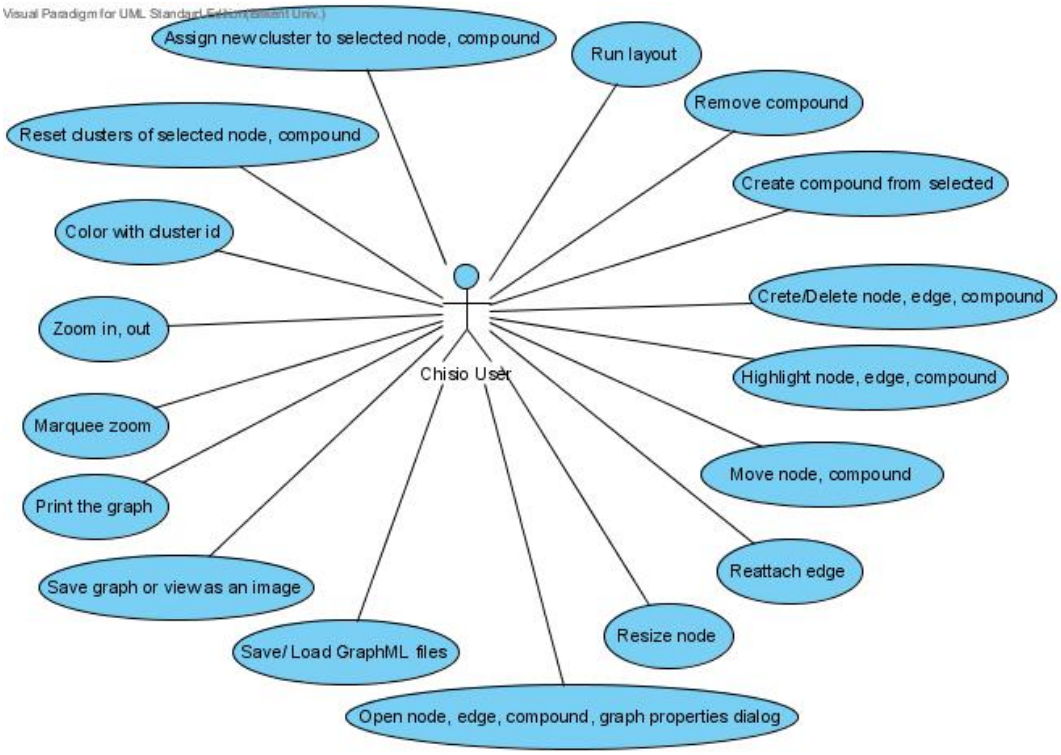


Figure 3.9: A UML Use Case Diagram for CHISIO as an editor

Flexible object UI is another important feature of CHISIO. UI properties of nodes, edges, compound nodes can be easily changed by using inspector dialogs. The abilities of inspector dialogs are shown in the Use Case Diagram in Figure 3.10.

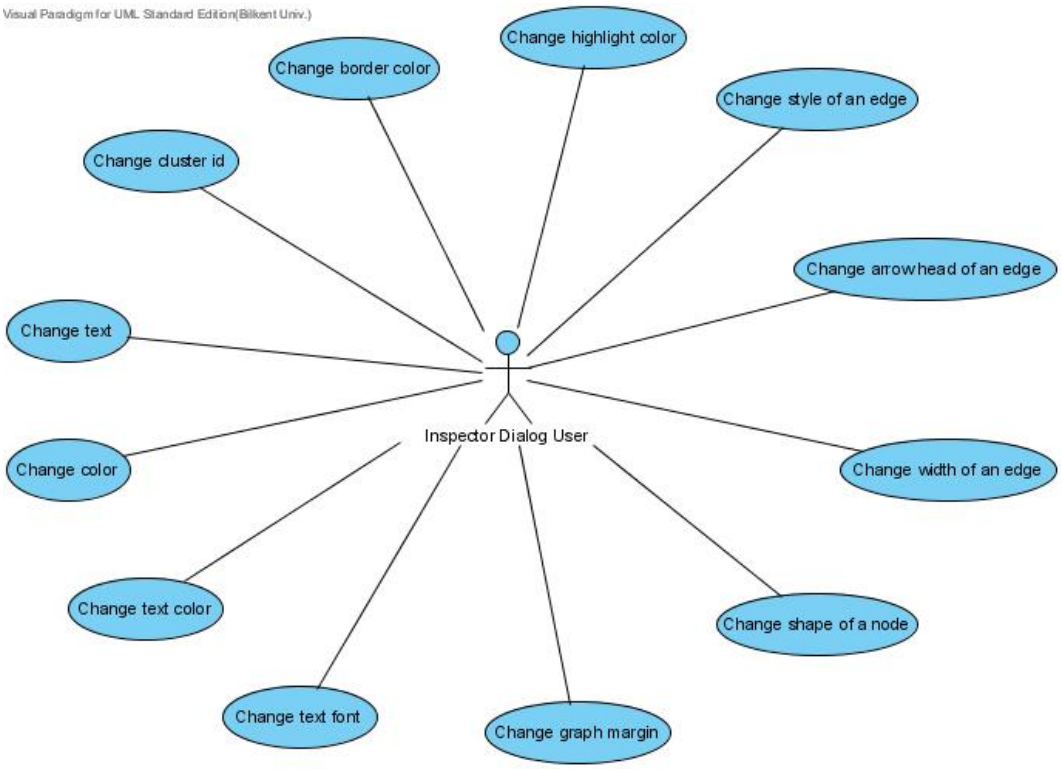


Figure 3.10: A UML Use Case Diagram for inspector dialogs in Chisio

# Chapter 4

# Chisio as an Editor

Chisio is a graph visualization tool for creating, editing and layout of compound or hierarchically structured graphs. The tool features user-friendly, interactive creation and manipulation of compound graphs. In addition, a number of popular graph layout algorithms, including ones designed by our group, have been implemented.

## 4.1 Drawings

Nodes, edges and compound nodes in Chisio all have distinct properties and UIs, which can be changed by its graphical user interface or through programming. Each node is drawn as a rectangle, ellipse or triangle. Edges can be drawn in a variety of styles. Compound nodes are always drawn with a rectangle, where the name text is displayed on its bottom margin and its geometry is auto-calculated using the geometry of its contents to tightly bound its contents plus user-defined child graph margins. Figure 4.1 shows the basics of drawing compound graphs in Chisio.

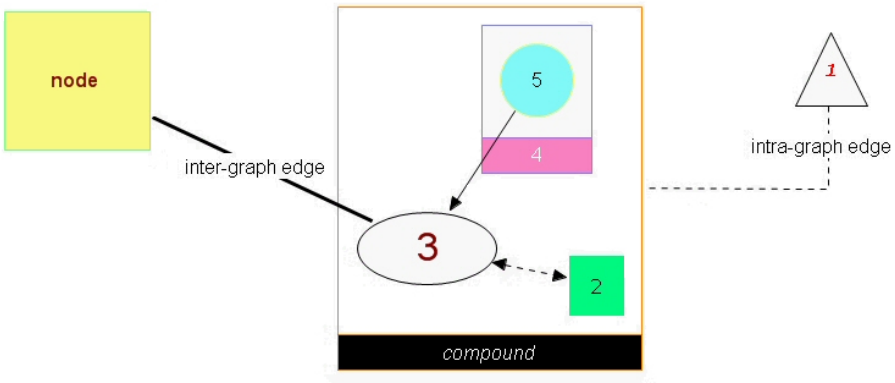


Figure 4.1: Basics of drawing in CHISIO

### 4.1.1 Nodes

Nodes are represented by a rectangle shape as default. But there are other predetermined shapes that can be added to the graph by user interaction. These are ellipse and triangle. If you would like to add more types of shapes, you can easily add them by following the instructions in the Chisio Programmer's Guide [4].

Figure 4.2 shows a sample "Node Properties" window. By using this window, you can change your node's appearance. This window can be opened up by either double clicking the node or by using the node's pop-up menu.

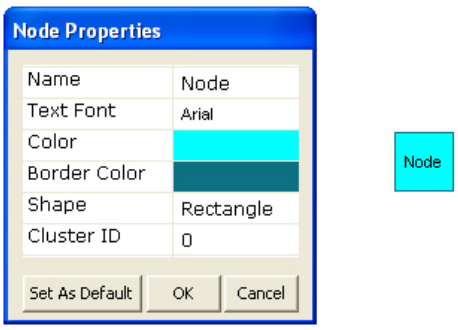


Figure 4.2: A node and its properties window or inspector

You can also change the label (font type, size, and color), color, border color, shape and cluster ID of a node by using its "Properties" window. "Set As Default" button is used for setting this node's properties as the default for the nodes that are to be created later on.

### 4.1.2 Compound Nodes

Similar to a simple node, a compound node's properties may be changed via the "Compound Properties" window (Figure 4.3). This window may also be opened up similar to the "Node Properties" window by double-clicking on it or through its pop-up menu.


Figure 4.3: A compound node (right) and its properties window

You can change the label, color and border color of a compound node through this window. The size of a compound node is auto-calculated by the geometry of its contents, and the compound node is always just large enough to tightly bound its contents plus some user-defined margins. "Set As Default" button is used for setting this compound's properties as the default for the compounds that are to be created later on.

### 4.1.3 Edges

Both directed and undirected graphs can be visualized in CHISIO. Edges are assumed to be connected to the center of their source and target nodes. An edge is drawn clipped, according to the specific shape and position of its source and target nodes (Figure 4.4).

The appearance of edges can be changed by using the "Edge Properties" window. This window can be opened up by double clicking the edge or by the pop-up menu (Figure 4.5). You can change the label, color, style (Solid, Dashed), arrow type (None, Source, Target, Both) and width (thickness) through this

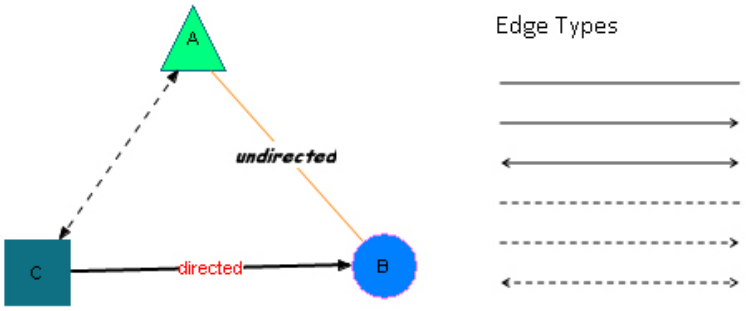


Figure 4.4: Examples of edges with different styles (left) and available edge types (right)

window. "Set As Default" button is used for setting this edge's properties as default for the edges that are to be created later on.

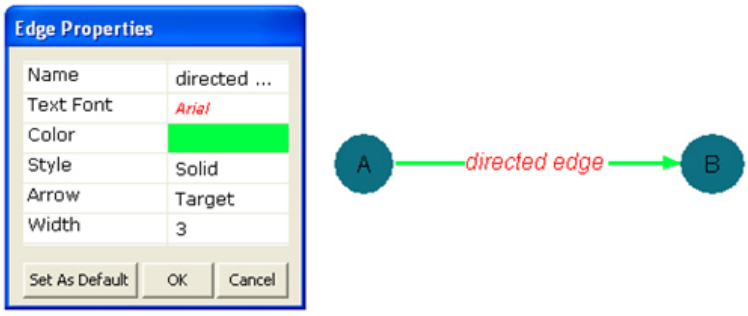


Figure 4.5: Edge properties window or edge inspector

*Graphs*

There are certain general properties independent of specific graph objects. These can be adjusted through the "Graph Properties" window (Figure 4.6).

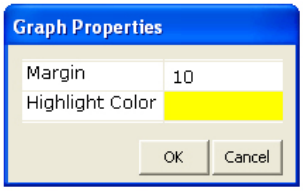


Figure 4.6: Graph properties window

Specifically, margins and highlight color (Section 4.5) can be customized through this window. Graph margins are used to separate graph objects from

their bounding boxes (such as nodes in a compound graph). Change of this value affects the margins around CHISIO drawings when you fit them into the window.

## 4.2 Chisio Tools

There are several tools to interact with the graphs; namely: Select Tools, Zoom Tools and Create Tools. Create Tools will be explained in Changing Topology (Section 4.3).

### 4.2.1 Select Tools

The "Select Tool" can be chosen from the top menubar using "Edit — Select Tool" or from the toolbar menu. This tool is used to select nodes, edges and compound nodes. Multiple selections are supported. When you select an object, handle points are drawn around the object. Handle points for nodes are filled. This means that, node can be resized. Handle points for compound nodes are empty as they can not be resized (Figure 4.7).

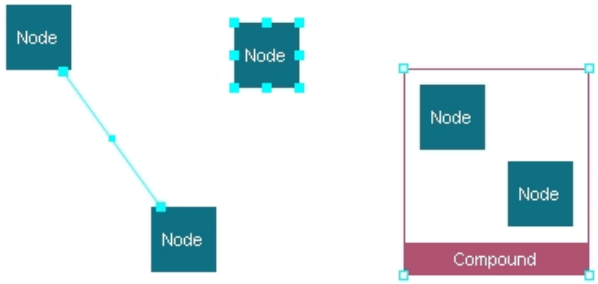


Figure 4.7: Example selection handles for all types of graph objects

When you press the left mouse button on the drawing canvas outside the boundaries of any graph objects, and drag your mouse, the "Select Tool" is automatically switched to the "Marquee Selection Tool". When you release the mouse button, all simple and compound nodes and edges that are completely included in the marquee selection area is selected (Figure 4.8).

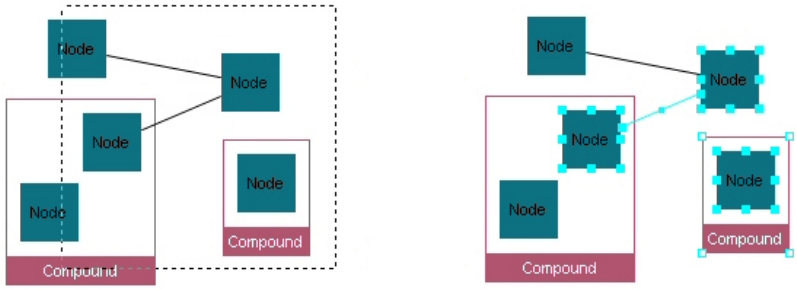


Figure 4.8: Example use of the Marquee Selection Tool

### 4.2.2 Zoom Tools

CHISIO has several zooming facilities: marquee zoom, zoom-in, zoom-out, zoom to specified level, and fit-in-window.

The "Marquee Zoom Tool" can be chosen from the top menubar using "Edit — Marquee Zoom Tool" or from the toolbar menu. This tool is used to zoom into a specified rectangular area of the graph (Figure 4.9 and Figure 4.10).

Other zoom operations like zoom-in, zoom-out and zoom to specified level are also supported. These operations can be found in the top menubar under "View — Zoom" and in the toolbar menu. In addition, graph pop-up menu provides zoom-in and zoom-out capabilities (Figure 4.11).

Fit-in-window operation is another useful zoom operation, which shows the whole graph in the window by properly scaling it. You can fit your graph in window from the top menubar using "View — Fit in window" or from the toolbar menu.

## 4.3 Changing Topology

You can interactively change the topological information of your graphs. You can create and delete graph objects easily. In addition, nodes and compounds can be transferred from one graph (root or child) to another (root or child). Finally

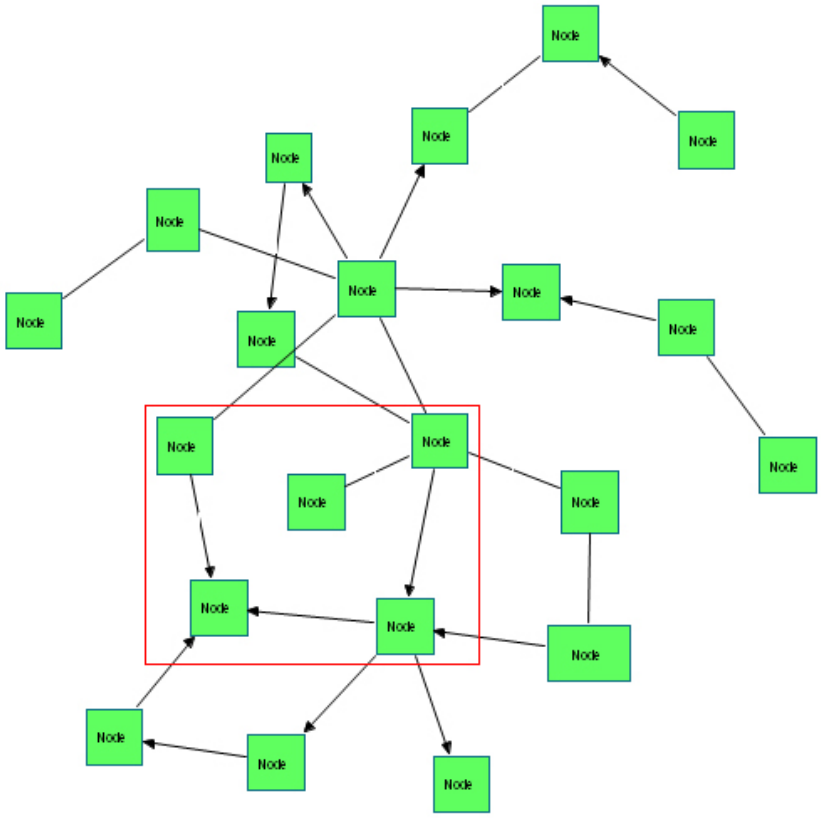


Figure 4.9: Before marquee zoom to the area specified by the red rectangle

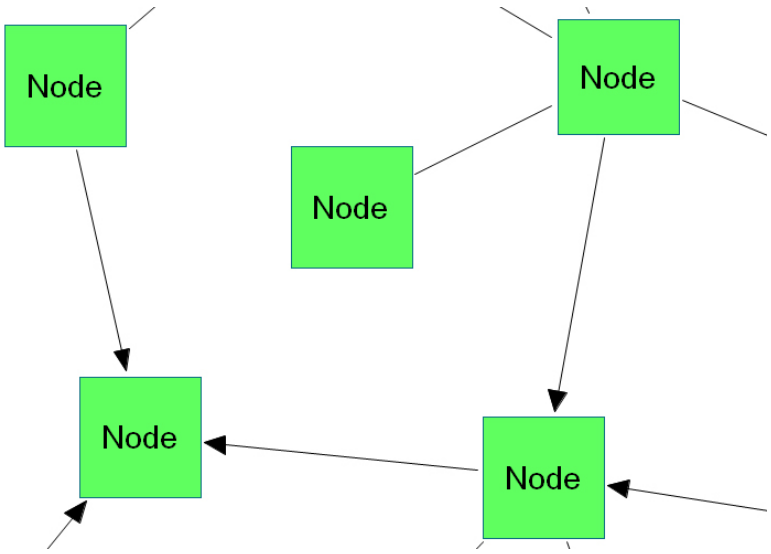


Figure 4.10: After marquee zooming into the area as described by Figure 4.9

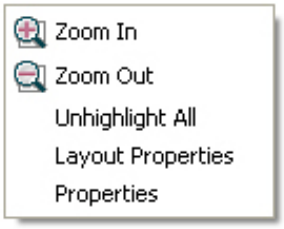


Figure 4.11: Graph pop-up menu includes zoom-in and zoom-out operations

edges can be reconnected by changing their source and/or target nodes.

### 4.3.1 Creating Graph Objects

Creating a new node or a new compound can be done from the top menubar under the "Edit" menu or by using the toolbar menu. When you select creation tools, the cursor is changed to emphasize the create operation. When you click any place on the drawing canvas, the creation is performed (Figure 4.12).


Figure 4.12: Create a node by simply clicking on the drawing canvas, where you would like your new node to be placed; before the creation (left) and after the creation (right)

Creating a new edge can be done from the top menubar under the "Edit — Create Edge" menu or by using the toolbar menu. When you select the edge creation tool, the cursor is changed to emphasize the edge creation. You must click first on the source node for this new edge. Your second click must be on to the target node for this edge to complete the creation of the new edge (Figure 4.13).

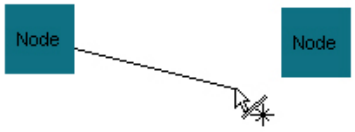


Figure 4.13: Create an edge by first clicking on the source node and then on the target node

### 4.3.2 Deleting Graph Objects

To delete nodes, compound nodes or edges, you must select them first. Then you can delete them in one of the following ways:

- Pressing "DEL" button on your keyboard;
- Using the toolbar menu item ;
- Using "Edit — Delete Selected" in the top menubar;
- Using "Delete" item under the node pop-up menu.

Multiple graph objects may be deleted at once using selection of these objects together. One may also remove existing compound nodes while keeping their child nodes and edges at their current locations. This can be done from the top menubar using "Edit — Remove Compound" or by using the toolbar menu after selecting the compound node(s) to be removed (Figure 4.14).

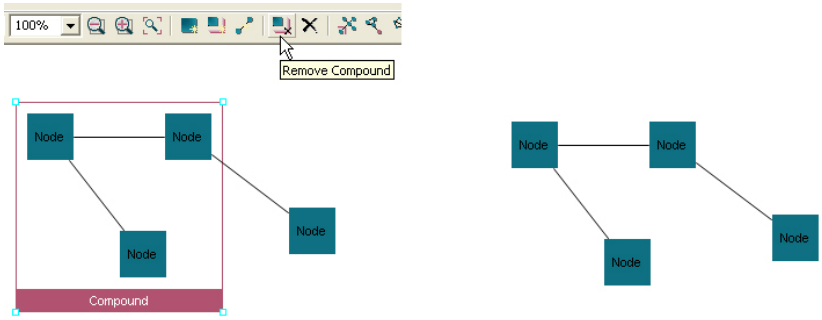


Figure 4.14: Remove compound; before and after

### 4.3.3 Transferring Graph Objects

Another way to change the topology is to move a node from one compound node (or root graph) to another compound node (or root graph). In other words, you can change the parent of a node.

This can only be done when the mode of the selection tool is "Transfer Mode". You can change the mode to "Transfer Mode" from the toolbar menu (Figure 4.15) or from the top menubar using "Edit — Transfer Mode".

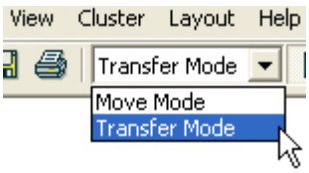


Figure 4.15: Move/Transfer Mode combo

When in the "Transfer Mode", drag operations might mean change of ownership or transfer. So you can transfer a node/compound node A into another compound node B by dragging and dropping A onto (within the bounds of) the compound node B. When you select a node and move it on to a compound node, compound nodes change background color to cyan to indicate that you will be transferring the selected node into the highlighted compound node should you release it at this location (Figure 4.16).

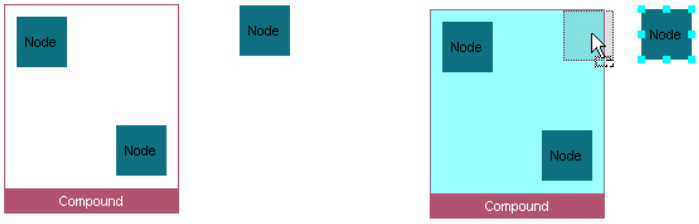


Figure 4.16: When transferring a node into a compound node, the compound node's background color changes to indicate the transfer; initially (left), when the transfer of the selected node is about to be performed

Also when you are in the "Transfer Mode", you can clone the selected graph objects by pressing the CTRL key while dragging them. When you press the

CTRL key, the cursor changes to indicate the cloning operation (Figure 4.17). Cloning is not possible in the "Move Mode".

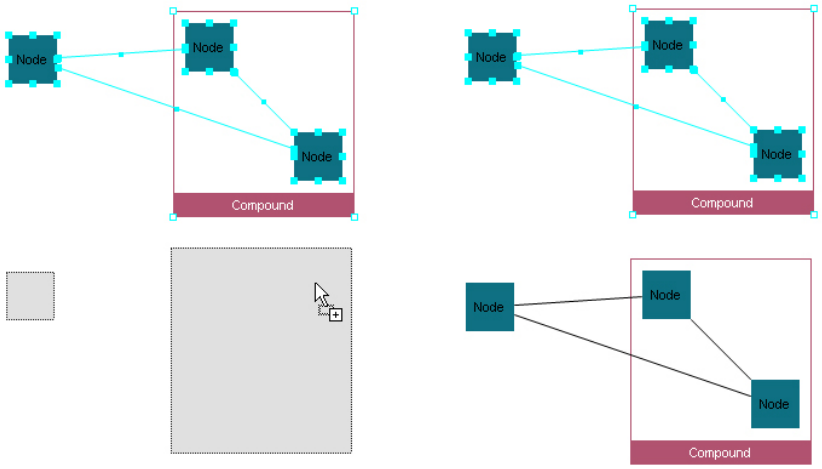


Figure 4.17: Clone operation; drag selected nodes pressing the CTRL key in the transfer mode and click to clone

Another transfer operation is via the "Create Compound from Selected" operation in the node pop-up menu. You can create a compound node from a set of selected nodes and compounds easily. This operation is also available through the top menubar using "Edit — Create Compound from Selected" (Figure 4.18).

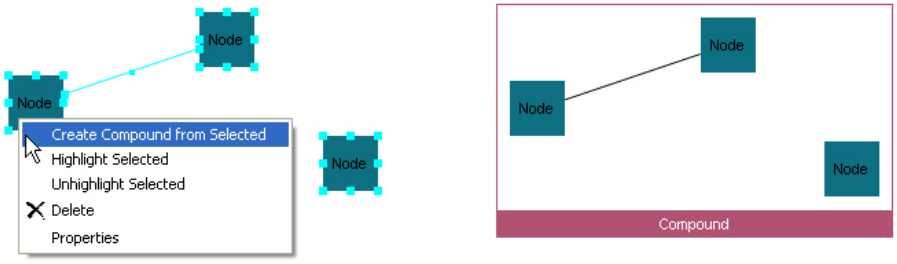


Figure 4.18: Create compound (right) from selected nodes (left)

### 4.3.4 Reconnecting an Edge

You can change the source or target of an edge after creation of it. When an edge is selected, its handle points become visible. When you move your mouse to the handle point associated with the target or source node, the cursor is changed

into a plus icon. You can click, drag and drop this point onto its new target or source. While dragging, the cursor indicates potential new target or source for this edge (Figure 4.19).

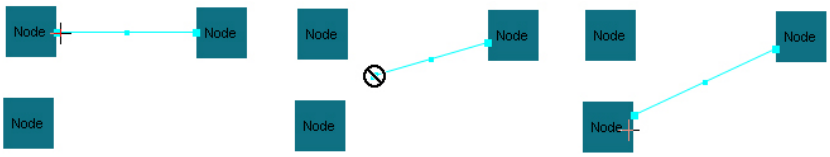


Figure 4.19: Edge reconnection; after selection of the edge (left), during dragging (middle) and after reconnection (right)

## 4.4 Changing Geometry

You can change the geometry of nodes by moving or resizing them. The geometry of the edges, on the other hand, can be changed by re-routing them by creating new bendpoints, by moving or deleting existing ones. The layout operation also changes the geometry of graph objects as will be explained later on.

To move a node, first you must select it. Then click anywhere on the node and start dragging; you will see a "ghost shape" of your node as you drag it. You can drop it to the location you like. Upon release of the mouse the operation is completed (Figure 4.20). We assume the "Move Mode" here; during "Transfer Mode" this might mean a transfer operation as explained earlier in Section 4.3 in Transferring Graph Objects.


Figure 4.20: Move a node by selecting and dragging it

To resize a node, first you must select it upon which its handle points will be visible. By using these points, you can resize the node. Just select a handle point and move it; a ghost shape will appear indicating the new size of the node upon release of the mouse (Figure 4.21).

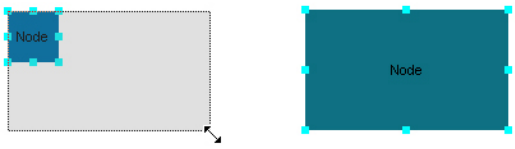


Figure 4.21: Resize a node; before (left) and after (right)

You can not resize compound nodes as their geometry is auto-calculated by the geometry of its contents as explained earlier. This is indicated by empty handles upon selection of compound nodes (Figure 4.22).

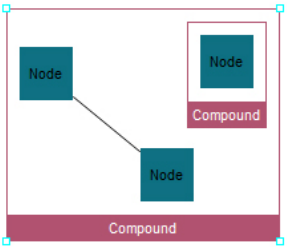


Figure 4.22: Compound nodes are automatically resized

Both move and resize operations may be applied to multiple objects simultaneously for convenience.

CHISIO supports bendpoints to freely route edges around other graph objects. You can create, delete and relocate bendpoints. When an edge is selected two handles of two different sizes becomes visible. The large ones (except for the end points of the edge used for reconnecting edges) correspond to the bends of the edge. The smaller ones, on the other hand, can be clicked on and dragged to create new bendpoints (Figure 4.23). Initial position of the newly created bendpoint will be right in between the two nearest existing bends on the edge but it will move with the mouse.

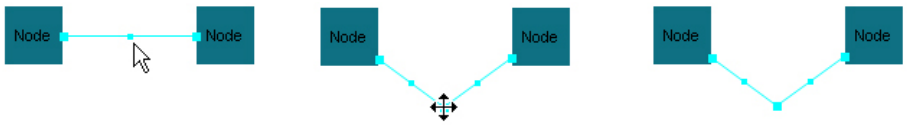


Figure 4.23: Bendpoint creation; after selection (left), during drag of the small handle (middle), and upon completion with the release of the mouse (right)

You can relocate existing bendpoints by dragging them to new locations as you like. If the final location of a bendpoint upon release of the mouse is to be aligned with its current neighbors of the bendpoint to make a straight line (or very near such a position), then the bendpoint is automatically deleted (Figure 4.24).

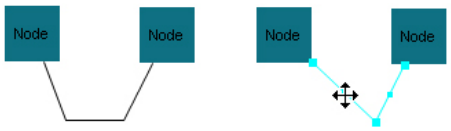


Figure 4.24: Bendpoint deletion; initially the edge has two bends (left); upon drag and release of the left bend, it is deleted to leave the edge with only one bend (right)

## 4.5 Highlighting

Graph objects may be highlighted using the highlight facility to differentiate them from others (e.g. a path, a cycle or a particular subgraph of interest). Selected objects may be highlighted through the "Highlight Selected" item under the node pop-up menu or using "View — Highlight Selected" in the top menubar. Multiple objects may be highlighted together (Figure 4.25).

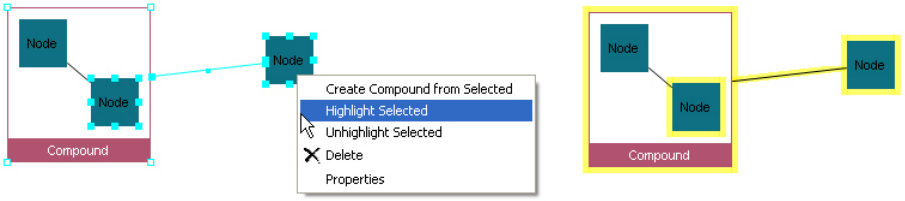


Figure 4.25: Highlighting objects

You can change the highlight color from the graph pop-up menu (Figure 4.11). Thus different objects may be highlighted with different colors if desired.

You can remove highlights of all objects by selecting "Unhighlight All" item in the top menubar or from the graph pop-up menu. Alternatively, a subset of currently highlighted objects may be unhighlighted by first selecting them and then choosing "View — Unhighlight Selected" in the top menubar or "Unhighlight Selected" in the node pop-up window.

## 4.6 Cluster IDs

You can group or cluster a set of nodes in a CHISIO graph. To do that, simply select the nodes you would like to assign a new cluster to, and then select "Cluster — Assign Selected to New Cluster" from the top menubar (Figure 4.26). This operation gives a new, unused cluster ID to selected nodes.

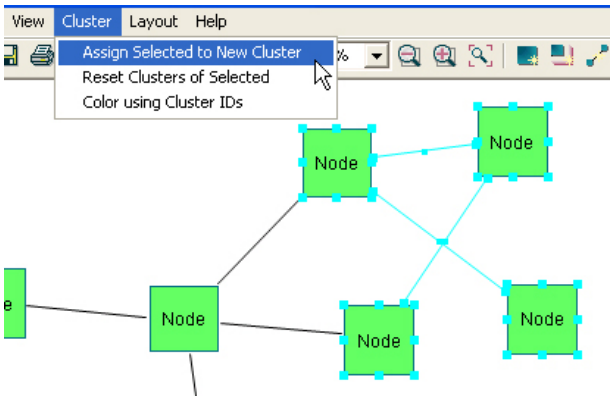


Figure 4.26: Creating a new cluster

You can find out about the new cluster IDs of these nodes through the node inspector (Figure 4.27). You may change the cluster ID of a node as needed from this window as well.

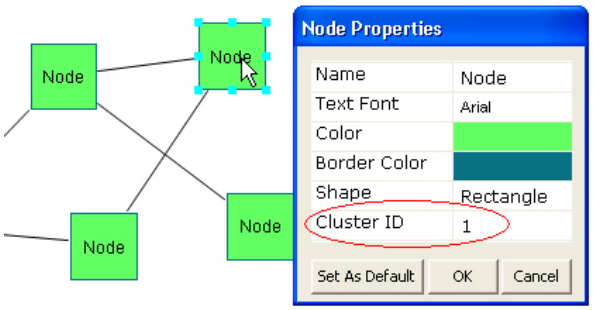


Figure 4.27: New cluster ID of the node as confirmed by the node properties window

You can also color-code the nodes in your CHISIO graph according to their cluster IDs. Nodes in the same cluster are colored with the same, unique color. Simply select "Cluster — Color using ClusterIDs" from the top menubar (Figure 4.28).

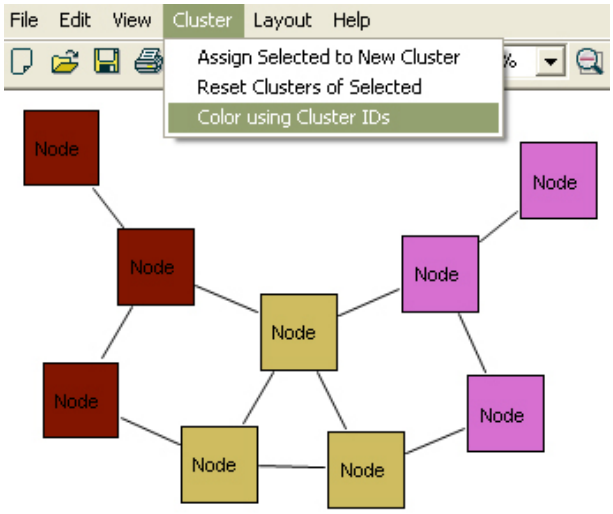


Figure 4.28: Coloring with cluster IDs results in three clusters, each with a unique color

You can reset the cluster information of selected nodes. If ClusterID of a node is "0", this means that node does not belong to any cluster. Select the nodes you want to reset their cluster information, and then select "Cluster — Reset Clusters of Selected".

## 4.7 Persistent Storage

You can save your graphs into a file for later use. Graphs are saved in GraphML file format [3], which is an XML-based graph format. In addition, your own files in GraphML format, possibly created by other programs may be loaded up into CHISIO. Drag and drop is supported for convenient file loading as well.

Every property of a graph object is written into the file in a regular format. Below you can find parts of a GraphML file for specific examples.

*Example 1:* A highlighted node (Figure 4.29)

...

```
<node id="n0">
```

```
  <data key="x">108</data>
  <data key="y">90</data>
  <data key="height">40</data>
  <data key="width">40</data>
  <data key="color">14 112 130</data>
  <data key="borderColor">14 112 130</data>
  <data key="text">Node</data>
  <data key="textFont">1|Arial|8|0|WINDOWS|1|-11|0|0|0|0|0|0|0|1|0|0|0|0|Arial</data>
  <data key="textColor">0 0 0</data>
  <data key="clusterID">0</data>
  <data key="highlightColor">255 255 0</data>
  <data key="shape">Rectangle</data>
</node>

...
```

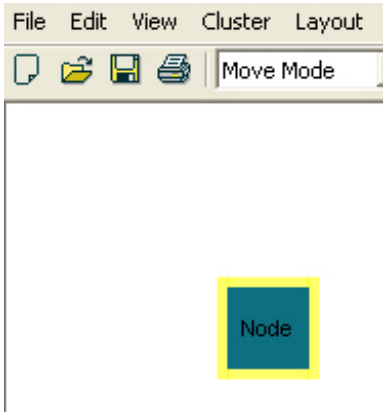


Figure 4.29: A Highlighted node for example GraphML format

*Example 2:* A dashed edge (Figure 4.30)

```
...

<edge id="e0" source="n0" target="n1">
  <data key="color">255 128 192</data>
  <data key="text">edge</data>
  <data key="textFont">1|Comic Sans MS|12|2|WINDOWS|1|-16|0|0|0|400|1|0|0|0|3|2|1|66|
    Comic Sans MS</data>
  <data key="textColor">0 0 255</data>
  <data key="style">Dashed</data>
  <data key="arrow">Target</data>
  <data key="width">2</data>
</edge>

...
```


Figure 4.30: An edge for example GraphML format

## 4.8 Static Images and Printing

You may save the current drawings of your graphs as static images as well. Supported image formats are BMP and JPEG. You have the option of recoding the entire drawing or only the currently viewable part of the drawing (Figure 4.31).

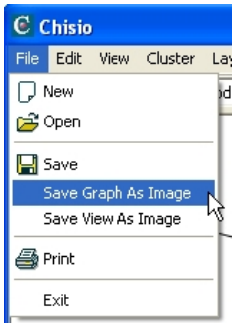


Figure 4.31: Save as image menu item

In addition, you can print your graphs from the top menubar "File — Print" or the toolbar menu. Print operation prints the whole graph in regardless of the currently viewable part of the drawing.

# Chapter 5

# Chisio Customization

## 5.1 Customize Editor

One can customize Chisio for their specific needs by adding new node/edge types or by modifying existing nodes/edges (UI and attributes) [4]. In addition, you may customize the menus to add new functionality as well as modifying node and edge menus and property inspectors.

### 5.1.1 Customizing User Interface

#### 5.1.1.1 Adding New Shapes

There are several shapes available for nodes in Chisio: Rectangle, Circle and Triangle. These shapes may be sufficient for some applications but many applications will require different types of shapes for drawing graphs of your own. In this part, we provide an example on how you can add a new shape to Chisio nodes.

If you want to put an image as a figure, then you should create a figure which draws an image file. For example suppose there is an image file `node.bmp` in C:

drive. We use it as a node shape in the following figure class.

```
import org.eclipse.draw2d.*;
import org.eclipse.draw2d.geometry.*;
import org.eclipse.swt.graphics.Image;

public class ImageFigure extends Figure
{
  public ImageFigure()
  {
    super();
  }

  public ImageFigure(Rectangle rect)
  {

    setBounds(rect);
  }

  protected void paintFigure(Graphics g)
  {
    Rectangle r = getParent().getBounds().getCopy();
    setBounds(r);
    Image image = new Image(null, "C:\node.bmp");
    g.drawImage(image, r.getLocation());
  }
}
```

But if we want to draw our own shape, then we must create our own figure by providing the code for painting it. Our example shape is a *diamond*.

First, we need to write a figure class for the new diamond shape, which extends from the existing class `Figure`. Then we need to override the `paintFigure` method as shown below:

```
public class DiamondFigure extends Figure
{
  public DiamondFigure(Rectangle rect)
  {
    setBounds(rect);
  }

  protected void paintFigure(Graphics g)
  {
    Rectangle r = getParent().getBounds().getCopy();
```

```
    setBounds(r);

    PointList points = new PointList();
    points.addPoint(new Point(r.x + r.width / 2, r.y));
    points.addPoint(new Point(r.x + r.width, r.y + r.height / 2));
    points.addPoint(new Point(r.x + r.width / 2, r.y + r.height));
    points.addPoint(new Point(r.x, r.y + r.height / 2));

    g.fillPolygon(points);
    g.drawPolygon(points);
  }
}
```

Above code segment implements a figure that draws a diamond. You can add this code into `NodeFigure` class as other figure classes for different shapes do (`RectangleFigure`, `EllipseFigure` and `TriangleFigure` classes).

Now we need to assign a name string to our new shape by adding it into the static array in `NodeModel` as shown below:

```
public static String[] shapes = {
  "Rectangle",
  "Ellipse",
  "Triangle",
  "Diamond"};
```

`NodeFigure` class has an `updateShape` method. This method decides which figure will be drawn. So we need to add our diamond figure into this method. In addition, we must associate our newly added name string to our new diamond figure.

```
public void updateShape(String s)
{
  this.shape = s;
  this.removeAll();

  if (shape.equals(NodeModel.shapes[0]))
  {
    add(new RectangleFigure(getBounds()));
  }
  else if (shape.equals(NodeModel.shapes[1]))
  {
```

```
    add(new EllipseFigure(getBounds()));
  }
  else if (shape.equals(NodeModel.shapes[2]))
  {
    add(new TriangleFigure(getBounds()));
  }
  else if (shape.equals(NodeModel.shapes[3]))
  {
    add(new DiamondFigure(getBounds()));
  }

  add(label);
}
```

Here, we must be careful about the index of the newly added shape name "Diamond". It is in the 4th place of the static array, so if shape's type is the same with the element in the 4th place of static array (`NodeModel.shapes[3]`), this means that shape is a diamond and we create our new diamond figure.

By updating this method, we have completed addition of a new shape to CHISIO nodes. See Figure 5.1 for examples of the new node shape.


Figure 5.1: Sample UIs for two diamond shaped nodes

#### 5.1.1.2 Adding New Attributes

Each graph object (simple node, compound node or edge) in CHISIO has several attributes. Existing attributes may be sufficient for drawings of many types of graphs. However specific applications are most likely to have other attributes. For example, one may want to maintain a dynamic size for the arrow heads of directed edges.

We first need to update the `EdgeModel` class to handle such a new attribute for edges. Specifically we need to add a field named `arrowSize` (say, with type int).

In addition, a default value for this new field is needed (DEFAULT_ARROW_SIZE). Default values are defined in the "Class Variables" section of each class.

Furthermore, a constant variable is needed for firing the property change. When we change the arrow size, this change is to be propagated to the corresponding edit-part. So, an identifier is defined in "Class Constants" section (P_ARROW_SIZE).

As usual, operation setter and getter methods are written for the newly created field. But in here, setter method fires the property change mentioned earlier.

```
public class EdgeModel extends GraphObject
{
  private NodeModel sourceNode;
  private NodeModel targetNode;
  protected String style;
  protected String arrow;
  protected int width;
  protected List bendpoints;
  protected int arrowSize;

  /**
   * Constructor
   */
  public EdgeModel()
  {
    super();
    this.text = DEFAULT_TEXT;
    this.textFont = DEFAULT_TEXT_FONT;
    this.textColor = DEFAULT_TEXT_COLOR;
    this.color = DEFAULT_COLOR;
    this.style = DEFAULT_STYLE;
    this.arrow = DEFAULT_ARROW;
    this.width = DEFAULT_WIDTH;
    this.bendpoints = new ArrayList();
    this.arrowSize = DEFAULT_ARROW_SIZE;
  }

  ...

  public int getArrowSize()
  {
    return arrowSize;
  }

  public void setArrowSize(int arrowSize)
```

```
  {
    this.arrowSize = arrowSize;
    firePropertyChange(P_ARROW_SIZE, null, arrow_size);
  }

  ...

// ------------------------------------------------------------------------
// Section: Class Variables
// ------------------------------------------------------------------------
  public static String DEFAULT_TEXT = "";

  public static Font DEFAULT_TEXT_FONT =
    new Font(null, new FontData("Arial", 8, SWT.NORMAL));

  ...

  public static int DEFAULT_WIDTH = 1;

  public static int DEFAULT_ARROW_SIZE = 1;

// ------------------------------------------------------------------------
// Section: Class Constants
// ------------------------------------------------------------------------
  public static final String P_STYLE = "_style";
  public static final String P_ARROW = "_arrow";
  public static final String P_WIDTH = "_width";
  public static final String P_BENDPOINT = "_bendpoint";
  public static final String P_ARROW_SIZE = "_arrowSize";
}
```

The model including the new attribute is now ready. Let's modify the figure part accordingly now. For this, we must add a new field and an update method to set this field and change the UI. We might also need to update the `paintFigure` method, which draws the UI.

```
public class EdgeFigure extends PolylineConnection
{
  ...

  boolean highlight;
  Color highlightColor;
  int arrowSize;

  /**
   * Constructor
   */
```

```
  public EdgeFigure(String text,
    Font textFont,
    Color textColor,
    Color color,
    String style,
    String arrow,
    int width,
    Color highlightColor,
    boolean highlight,
    int arrowSize)
  {
    super();

    ...

    updateWidth(width);
    updateHighlightColor(highlightColor);
    updateArrowSize(arrowSize);
  }

  ...

  public void updateArrowSize(int newArrowSize)
  {
    // Do the operations for setting the arrow size
  }

  ...
}
```

Now that we have updated the model and figure parts, we must connect them to each other. We need to update the edit-part for propagating the changes in the model to the figure (UI). The edit-part of an edge is implemented in class `ChsEdgeEditPart`. We must update `createFigure` and `propertyChange` methods in this class.

```
protected IFigure createFigure()
{
  EdgeModel model = getEdgeModel();
  EdgeFigure eFigure = new EdgeFigure(model.getText(),
    model.getTextFont(),
    model.getTextColor(),
    model.getColor(),
    model.getStyle(),
    model.getArrow(),
    model.getWidth(),
```

```
    model.getHighlightColor(),
    model.isHighlight(),
    model.getArrowSize());

  eFigure.updateHighlight(
    (HighlightLayer) getLayer(HighlightLayer.HIGHLIGHT_LAYER),
    getEdgeModel().isHighlight());

  ...
}

public void propertyChange(PropertyChangeEvent evt)
{
  if (evt.getPropertyName().equals(EdgeModel.P_TEXT))
  {
    ((EdgeFigure)figure).updateText((String) evt.getNewValue());
  }

  ...

  else if (evt.getPropertyName().equals(EdgeModel.P_HIGHLIGHT))
  {
    ((EdgeFigure)figure).updateHighlight(
      (Layer) getLayer(HighlightLayer.HIGHLIGHT_LAYER));
  }
  else if (evt.getPropertyName().equals(EdgeModel.P_HIGHLIGHTCOLOR))
  {
    ((EdgeFigure)figure).
      updateHighlightColor((Color) evt.getNewValue());
  }
  else if (evt.getPropertyName().equals(EdgeModel.P_ARROW_SIZE))
  {
    ((EdgeFigure)figure).updateArrowSize((Integer) evt.getNewValue());
  }
}
```

Thus we have completed adding a new attribute into CHISIO edges. In the next part, we will make the necessary changes to expose this new attribute to the users through the edge inspector window.

#### 5.1.1.3 Modifying Inspector Windows

The implementation of object property inspector windows in CHISIO are based on a class named Inspector and specific inspectors extend this class (e.g.

`EdgeInspector`). If you want to add a new item to the edge inspector, first you must add it into `EdgeInspector` class as follows. We will just change the constructor of the class.

```
private EdgeInspector(GraphObject model, String title, ChisioMain main)
{
  super(model, title, main);

  TableItem item = addRow(table, "Name");
  item.setText(1, model.getText());

  ...

  item = addRow(table, "Width");
  item.setText(1, "" + ((EdgeModel) model).getWidth());

  item = addRow(table, "Arrow Size");
  item.setText(1, "" + ((EdgeModel) model).getArrowSize());

  createContents(shell);

  ...
}
```

Here we add a new row with attribute name "Arrow Size", whose value is taken from the corresponding model object. Now, we need to update `createContents` method of `Inspector` class to add interactivity to our newly added attribute. We already had five types of objects in our edge inspector tables: Combo, Text, Number, Color and Font. Our new attribute's value is a number. So we need to add it into the if-statement, which is tagged with `NUMBER`.

```
public void createContents(final Shell shell)
{
  // Create an editor object to use for text editing
  final TableEditor editor = new TableEditor(table);
  editor.horizontalAlignment = SWT.LEFT;
  editor.grabHorizontal = true;

  // Use a selection listener to get seleceted row
  table.addSelectionListener(new SelectionAdapter()
  {
    public void widgetSelected(SelectionEvent event)
    {
      // Dispose any existing editor
```

```
      Control old = editor.getEditor();

      ...

      if (item != null)
      {
        // COMBO
        if (item.getText().equals("Style")
          || item.getText().equals("Arrow")
          || item.getText().equals("Shape"))
        {
          ...
        }
        // TEXT
        else if (item.getText().equals("Name"))
        {
          ...
        }
        // NUMBER
        else if (item.getText().equals("Margin")
          || item.getText().equals("Cluster ID")
          || item.getText().equals("Width")
          || item.getText().equals("Arrow Size"))
        {
          ...
        }
        // COLOR
        else if (item.getText().equals("Border Color")
          || item.getText().equals("Color")
          || item.getText().equals("Highlight Color"))
        {
          ...
        }
        // FONT
        else if (item.getText().equals("Text Font"))
        {
          ...
        }

        table.setSelection(-1);
      }
    }
  });

  ...
}
```

We have added new attribute into our inspector window (Figure 5.2). When "OK" button is pressed, values are transferred to the model. So we need to update

the listener for the "OK" button to add the newly added arrow size attribute as below.

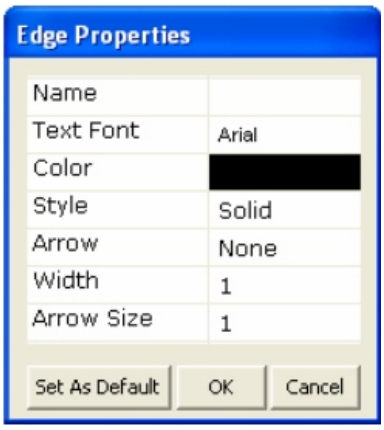


Figure 5.2: Sample screenshot of the Edge Inspector after addition of the new attribute "Arrow Size"

```
okButton.addSelectionListener(new SelectionAdapter()
{
  public void widgetSelected(final SelectionEvent e)
  {
    TableItem[] items = table.getItems();

    for (TableItem item : items)
    {
      if (item.getText().equals("Name"))
      {
        model.setText(item.getText(1));
      }

      ...

      else if (item.getText().equals("Margin"))
      {
        new ChangeMarginAction((CompoundModel)model,
          Integer.parseInt(item.getText(1))).run();
      }
      else if (item.getText().equals("Arrow Size"))
      {
        ((EdgeModel) model).
          setArrowSize(Integer.parseInt(item.getText(1)));
      }
    }

    shell.close();
  }
```

```
});
```

Furthermore, we need to update `setAsDefault` method in `EdgeInspector` class to change the default arrow size with this new value.

```
public void setAsDefault()
{
  TableItem[] items = table.getItems();

  for (TableItem item : items)
  {
    if (item.getText().equals("Name"))
    {
      EdgeModel.DEFAULT_TEXT = item.getText(1);
    }

    ...

    else if (item.getText().equals("Width"))
    {
      EdgeModel.DEFAULT_WIDTH = Integer.parseInt(item.getText(1));
    }
    else if (item.getText().equals("Arrow Size"))
    {
      EdgeModel.DEFAULT_ARROW_SIZE = Integer.parseInt(item.getText(1));
    }
  }
}
```

Viola! We have now completed adding a new item into an inspector.

#### 5.1.1.4 Changing Defaults For Nodes And Edges

There are default values for the creation of simple and compound nodes and edges. These values can be changed easily by updating a few lines.

Defaults for nodes are kept in the `NodeModel` class in the "Class Variables" section. By changing these variables, we can change the defaults for nodes.

```
// ------------------------------------------------------------------------
// Section: Class Variables
```

```
// -----------------------------------------------------------------------
  public static Dimension DEFAULT_NODE_SIZE = new Dimension(40, 40);

  public static String DEFAULT_TEXT = "Node";

  public static Font DEFAULT_TEXT_FONT =
    new Font(null, new FontData("Arial", 8, SWT.NORMAL));

  public static Color DEFAULT_TEXT_COLOR = ColorConstants.black;

  public static Color DEFAULT_COLOR = new Color(null, 14, 112, 130);

  public static Color DEFAULT_BORDER_COLOR = new Color(null, 14, 112, 130);

  public static String DEFAULT_SHAPE = shapes[0];

  public static int DEFAULT_CLUSTER_ID = 0;
```

Compound node and edge default values may be changed similarly under the "Class Variables" section of `CompoundModel` and `EdgeModel` classes, respectively.

### 5.1.2 File Operations

Persistence of newly added attributes requires special attention. Below you will find what you need to do so these attributes can be saved and loaded back properly on disk.

#### 5.1.2.1 Integrating New Attributes Into Save Operation

We have added the "arrow size" attribute for edges earlier on. Now, we will add support for this attribute in the save operation.

There is a `GraphMLWriter` class for saving our graphs as a ".graphml" file. We need to update this class.

First, we must define a `KeyType` for arrow size attribute (arrowSizeKey). Also we need to add code into `writeXMLFile` and `createEdge` methods as shown

below. If we had added a new parameter for nodes, then we should have changed `createNode` method as well.

```
public class GraphMLWriter extends XMLWriter
{
  HashMap hashMap = new HashMap();

  GraphmlType newGraphml;

  KeyType xKey;

  ...

  KeyType highlightColorKey;

  KeyType arrowSizeKey;

  public Object writeXMLFile(CompoundModel root)
  {
    // create a new graphml file
    GraphmlDocument newGraphmlDoc = GraphmlDocument.Factory.newInstance();
    newGraphml = newGraphmlDoc.addNewGraphml();

    // define the keys that will be used in xml file
    xKey = newGraphml.addNewKey();
    xKey.setId("x");
    xKey.setFor(KeyForType.NODE);
    xKey.setAttrName("x");
    xKey.setAttrType(KeyTypeType.INT);

    ...

    highlightColorKey = newGraphml.addNewKey();
    highlightColorKey.setId("highlightColor");
    highlightColorKey.setFor(KeyForType.ALL);
    highlightColorKey.setAttrName("highlightColor");
    highlightColorKey.setAttrType(KeyTypeType.STRING);

    arrowSizeKey = newGraphml.addNewKey();
    arrowSizeKey.setId("arrowSize");
    arrowSizeKey.setFor(KeyForType.NODE);
    arrowSizeKey.setAttrName("arrowSize");
    arrowSizeKey.setAttrType(KeyTypeType.INT);

    // create a new graph with our root node recursively
    GraphType newGraph = newGraphml.addNewGraph();
    createTree(newGraph, root, "");

    return newGraphmlDoc;
```

```
  }

  public void createEdge(EdgeType newEdge, EdgeModel model)
  {
    // write edge's properties into file
    DataType colorData = newEdge.addNewData();
    colorData.setKey(colorKey.getId());
    RGB rgb = model.getColor().getRGB();
    colorData.set(XmlString.Factory.newValue(rgb.red + " " +
      rgb.green + " " + rgb.blue));

    ...

    if (model.isHighlight())
    {
      DataType highlightColorData = newEdge.addNewData();
      highlightColorData.setKey(highlightColorKey.getId());
      rgb = model.getHighlightColor().getRGB();
      highlightColorData.set(XmlString.Factory.
        newValue(rgb.red + " " + rgb.green + " " + rgb.blue));
    }

    DataType arrowSizeData = newEdge.addNewData();
    arrowSizeData.setKey(arrowSizeKey.getId());
    arrowSizeData.set(XmlString.Factory.
      newValue("" + model.getArrowSize()));
  }
}
```

That's all. We now have support for our new attribute on save.

#### 5.1.2.2 Integrating New Attributes Into Load Operation

Now that we know how to save graphs with newly introduced attribute "arrow size", let's now integrate this attribute into the load operation. For this purpose, we need to update `readEdge` method in `GraphMLReader` class as shown below. If we had added a new parameter for nodes, then we should have updated the `readNode` method as well.

```
public void readEdge(EdgeType edge, EdgeModel model)
{
  int width = -1;
  String style = null;
```

```
  ...

  int arrowSize = -1;

  // read edge's properties
  DataType[] datas = edge.getDataArray();

  for (int i = 0; i < datas.length; i++)
  {
    DataType data = datas[i];

    if (data.getKey().equalsIgnoreCase("color"))
    {
      color = data.newCursor().getTextValue();
    }

    ...

    else if (data.getKey().equalsIgnoreCase("highlightColor"))
    {
      highlightColor = data.newCursor().getTextValue();
    }
    else if (data.getKey().equalsIgnoreCase("arrowSize"))
    {
      arrowSize = Integer.parseInt(data.newCursor().getTextValue());
    }
  }

  ...

  if (highlightColor != null)
  {
    String[] rgb = highlightColor.split(" ");
    int r = Integer.parseInt(rgb[0]);
    int g = Integer.parseInt(rgb[1]);
    int b = Integer.parseInt(rgb[2]);
    model.setHighlightColor(new Color(null, r, g, b));
    model.setHighlight(true);
  }

  if (arrowSize > 0)
  {
    model.setArrowSize(arrowSize);
  }
}
```

By reading the `arrowSize` field in ".graphml" files, we have added the support for loading newly created attribute "arrow size".

#### 5.1.2.3 Reading Other File Formats

CHISIO uses GraphML file format for persistent storage of graphs. There are two classes `GraphMLReader` and `GraphMLWriter` for load and save operations, which are extended from abstract classes `XMLReader` and `XMLWriter`.

If you would like to support other file formats such as GXL, GML or your own XML-based format, you must extend `XMLReader` and `XMLWriter` classes accordingly. Suppose we want to support ".gxl" files instead.

```
public class GXLReader extends XMLReader
{
  public CompoundModel readXMLFile(File xmlFile)
  {
    // parse GXL file in here and return the read root graph
  }
}

public class GXLWriter extends XMLWriter
{
  public Object writeXMLFile(CompoundModel root)
  {
    // create GXL file from root graph return the file content
  }
}
```

After writing new classes named `GXLReader` and `GXLWriter` similar to their ".graphml" counterpart, we need to change `LoadAction` and `SaveAction` classes as below.

```
public class LoadAction extends Action
{
  ...

  public void run()
  {
    ...

    File xmlfile = new File(filename);

    XMLReader reader = new GXLReader();
    CompoundModel root = reader.readXMLFile(xmlfile);
```

```
    ...
  }
}

public class SaveAction extends Action
{
  ...

  public void run()
  {
    ...

    BufferedWriter xmlFile =
      new BufferedWriter(new FileWriter(fileName));

    XMLWriter writer = new GXLWriter();
    xmlFile.write(writer.writeXMLFile(root).toString());
    xmlFile.close();

    ...
  }
}
```

In the run method of `LoadAction` class, we simply change `GraphMLReader` to `GXLReader` and in the run method of `SaveAction` class, change `GraphMLWriter` to `GXLWriter`. That's all.

### 5.1.3 Menu Operations

Each operation in the toolbar menu, pop-up menus or the main menu has an associated Action. So, if you would like to add a new menu item, you must first create an associated action for it.

For example, let's suppose we want to add a new operation for finding neighbors of a selected node and highlight them. We can write a new action for this operation as follows. Icon for this action is set in the constructor method.

```
import org.eclipse.jface.action.Action;

public class HighlightNeighborsAction extends Action
```

```
{
  ChisioMain main;

  /**
   * Constructor
   */
  public HighlightNeighborsAction(ChisioMain main)
  {
    super("Highlight Neighbors");
    this.setToolTipText("Highlight Neighbors");
    setImageDescriptor(ImageDescriptor.createFromFile(
      ChisioMain.class,
      "icon/highlight-neighbors.gif"));
    this.main = main;
  }

  public void run()
  {
    ScrollingGraphicalViewer viewer = main.getViewer();
    // Iterates selected objects; for each selected objects, highlights them
    Iterator selectedObjects =
      ((IStructuredSelection) viewer.getSelection()).iterator();

    if (selectedObjects.hasNext())
    {
      Object model = ((EditPart)selectedObjects.next()).getModel();

      if (model instanceof NodeModel)
      {
        Iterator<NodeModel> neighbors =
          ((NodeModel) model).getNeighborsList().iterator();

        while (neighbors.hasNext())
        {
          NodeModel next = neighbors.next();
          next.setHighlightColor(main.higlightColor);
          next.setHighlight(true);
        }
      }
    }
  }
}
```

Now, we can use this action to create a new menu item.

#### 5.1.3.1 New Main Menu Item

We must update `createBarMenu` method in `TopMenuBar` class for adding this action to the main menu. Suppose the new operation is to be added under the "View" menu.

```
public static MenuManager createBarMenu(ChisioMain main)
{
  chisio = main;

  MenuManager menuBar = new MenuManager("");
  MenuManager fileMenu = new MenuManager("&File");
  MenuManager editMenu = new MenuManager("&Edit");
  MenuManager viewMenu = new MenuManager("&View");
  ...

  viewMenu.add(new Separator());
  viewMenu.add(new HighlightSelectedAction(chisio));
  viewMenu.add(new HighlightNeighborsAction(chisio));
  viewMenu.add(new RemoveHighlightFromSelectedAction(chisio));
  viewMenu.add(new RemoveHighlightsAction(chisio));

  ...
}
```

#### 5.1.3.2 New Pop-up Menu Item

We must update `createActions` method in `PopupManager` class for adding this new operation to pop-up menu. This is an action for nodes/compound nodes. So we must add it to "NODE-COMPOUND POPUP" part of the code.

```
public void createActions(IMenuManager manager)
{
  EditPart ep = main.getViewer().findObjectAt(clickLocation);

  if (ep instanceof RootEditPart)
  {
    // GRAPH POPUP
    manager.add(new ZoomAction(main, 1, clickLocation));
    manager.add(new ZoomAction(main, -1, clickLocation));
    manager.add(new RemoveHighlightsAction(main));
    manager.add(new LayoutInspectorAction(main));
```

```
    manager.add(new InspectorAction(main, true));
    main.getViewer().select(ep);
  }
  else if (ep instanceof NodeEditPart)
  {
    // NODE-COMPOUND POPUP
    manager.add(new CreateCompoundFromSelectedAction(main));
    manager.add(new HighlightSelectedAction(main));
    manager.add(new HighlightNeighborsAction(main));
    manager.add(new RemoveHighlightFromSelectedAction(main));
    manager.add(new DeleteAction(main.getViewer()));
    manager.add(new InspectorAction(main, false));
  }
  else if (ep instanceof ChsEdgeEditPart)
  {
    // EDGE POPUP
    manager.add(new HighlightSelectedAction(main));
    manager.add(new RemoveHighlightFromSelectedAction(main));
    manager.add(new DeleteAction(main.getViewer()));
    manager.add(new InspectorAction(main, false));
  }
}
```

#### 5.1.3.3 New Toolbar Item

We must update `ToolbarManager` class for adding this operation to the toolbar menu. The icon for this item is already set in the constructor of `HighlightNeighborsAction` class.

```
public ToolbarManager(int style, ChisioMain main)
{
  super(style);

  add(new NewAction("New", main));
  add(new LoadAction(main));
  add(new SaveAction(main));

  ...

  add(new HighlightNeighborsAction(main));

  ...

  add(new SugiyamaLayoutAction(main));
  add(new Separator());
```

```
  update(true);
}
```

## 5.2 Extendible Layout Architecture

Chisio currently supports several layout styles from the basic spring embedder to hierarchical (Sugiyama) layout to compound spring embedder to circular layout. But, one may want to integrate a new layout operation into the existing layout architecture of Chisio. Hence before customization, the detail of the layout architecture will be given.

### 5.2.1 Chisio Layout Architecture

The basis in Chisio layout is constructed through an abstract layout class and an associated l-structure to represent compound graphs named `AbstractLayout` and `LGraphManager/LGraph/LNode/LEdge`, respectively. Here an `LGraphManager` maintains and manages a list of `LGraph` instances, which in turn maintains a list of `LNode`'s and `LEdge`'s. Notice here that similar to the Chisio structure, the top-most level is composed of a root node and a root graph (root node's child graph) composing the actual content of the graph structure. In other words, the root node and the root graph are there even if the graph is empty!

The `AbstractLayout` class converts the given Chisio model into a generic l-structure used only for layout purposes that is destroyed at the end of layout. Individual layout algorithms extend the `AbstractLayout` class and run on this l-structure. When the layout is finished, the geometry information of the l-level is transferred back to Chisio model.

### 5.2.2 Adding New Layout Algorithms

Currently, Chisio hosts the following layout styles:

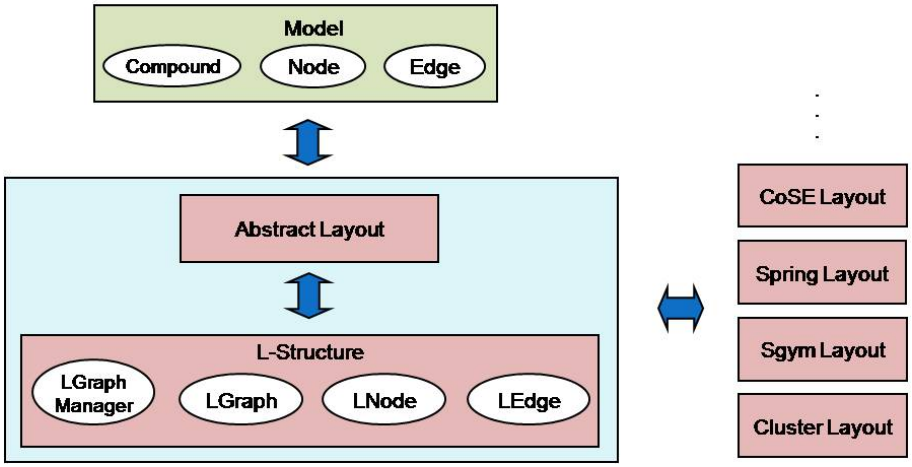


Figure 5.3: CHISIO Layout Architecture

- CoSE: Compound graph layout algorithm based on a spring embedder
- Cluster Layout: Members of each cluster is grouped and laid out together
- CiSE: Circular graph layout algorithm based on a spring embedder
- Circular Layout: AVSDF circular layout
- Spring Layout: Basic spring embedder
- Sugiyama Layout: A hierarchical layout style based on Sugiyama algorithm

Detailed information about these algorithms and our implementation can be found in Section 6. All these layouts extend `AbstractLayout` class and its abstract methods are overridden to implement the corresponding layout algorithm. If you would like to add a new layout algorithm, you must write a new class extending `AbstractLayout` as well.

For example, let's suppose we want to add random layout, which randomly scatters the nodes, as a new style.

```
public class RandomLayout extends AbstractLayout
{
  public RandomLayout(CompoundModel rootModel)
  {
```

```
    super(rootModel);
  }

  public void layout()
  {
    this.positionNodesRandomly();
  }
}
```

In here, layout method, which is abstract in the base class is to include the actual layout algorithm. If we need some initialization for layout, we can override the initialize method, which is also in `AbstractLayout`.

In case we need layout-style specific nodes and edges for our layout algorithm, we should extend `LNode` and `LEdge` classes, respectively, and override the methods `createNewLNode` and `createNewLEdge` in `AbstractLayout` as shown below.

```
import org.eclipse.draw2d.geometry.*;

public class RandomNode extends LNode
{
  public RandomNode(LGraphManager gm)
  {
    super(gm);
  }

  public RandomNode(LGraphManager gm, Point loc, Dimension size)
  {
    super(gm, loc, size);
  }

  ...

}

public class RandomLayout extends AbstractLayout
{
  public RandomLayout(CompoundModel rootModel)
  {
    super(rootModel);
  }

  public LNode createNewLNode(LGraphManager gm, NodeModel model)
  {
    return new RandomNode(gm);
  }
```

```
  public LNode createNewLNode(LGraphManager gm,
    NodeModel model,
    Point loc,
    Dimension size)
  {
    return new RandomNode(gm, loc, size);
  }

  public void layout()
  {
    this.positionNodesRandomly();
  }
}
```

### 5.2.3 New Layout Options

`LayoutOptionsPack` is a class for maintaining the layout parameters that can be customized by the user. This class has an inner class for each layout style to handle parameters belonging to that particular layout style. Hence, we should add an inner class for our new layout style (i.e. random layout).

```
public class LayoutOptionsPack implements Serializable
{
  private static LayoutOptionsPack instance;

  private General general;
  private CoSE coSE;
  private Cluster cluster;
  private CiSE ciSE;
  private AVSDF avsdf;
  private Spring spring;
  private Sgym sgym;
  private Random random;

  public class General
  {
    ...
  }

  public class CoSE
  {
    ...
  }
```

```
  ...

  public class Random
  {
    // our parameters and setter, getter methods
  }

  private LayoutOptionsPack()
  {
    this.general = new General();
    this.coSE = new CoSE();
    this.cluster = new Cluster();
    this.ciSE = new CiSE();
    this.avsdf = new AVSDF();
    this.spring = new Spring();
    this.sgym = new Sgym();
    this.random = new Random();

    setDefaultLayoutProperties();
  }

  public void setDefaultLayoutProperties()
  {
    general.setAnimationPeriod(50);
    general.setAnimationDuringLayout(
      AbstractLayout.DEFAULT_ANIMATION_DURING_LAYOUT);
    general.setAnimationOnLayout(
      AbstractLayout.DEFAULT_ANIMATION_ON_LAYOUT);

    ...

    random.set...
    random.set...
  }

  public General getGeneral()
  {
    return general;
  }

  ...

  public Random getRandom()
  {
    return random;
  }
}
```

We have updated layout options pack which gives us the ability to parameterize our layouts. We can easily change the layout properties window by adding a new tab for our new style as well. Our new layout style will use the values in layout options pack, which will be set through a new tab in the layout properties window as described below.

```
public class LayoutInspector extends Dialog
{
  ...

  protected void createContents()
  {
    ...

    // Create the UI for new layout. Buttons, sliders, checkboxes etc.

    ...
  }

  public void storeValuesToOptionsPack()
  {
    LayoutOptionsPack lop = LayoutOptionsPack.getInstance();

    // General
    lop.getGeneral().setAnimationPeriod(animationPeriod.getSelection());
    lop.getGeneral().setAnimationOnLayout(
      animateOnLayoutButton.getSelection());

    ...

    // Random
    lop.getRandom().set..
    lop.getRandom().set..
  }

  public void setInitialValues()
  {
    LayoutOptionsPack lop = LayoutOptionsPack.getInstance();

    // General
    animationPeriod.setSelection(lop.getGeneral().getAnimationPeriod());
    animateDuringLayoutButton.setSelection(
      lop.getGeneral().isAnimationDuringLayout());

    ...

    // Random
```

```
    // Take values from UI and set into layout options pack (lop)
  }

  public void setDefaultLayoutProperties(int select)
  {
    if (select == 0)
    {
      // General
      animateDuringLayoutButton.setSelection(
        AbstractLayout.DEFAULT_ANIMATION_DURING_LAYOUT);
      animationPeriod.setSelection(50);

      ...
    }

    ...

    else if (select == 7)
    {
      // Random

      // Fill the UI with the default values of layout
    }
  }
}
```

Thus, we have added a new tab for our new layout algorithm to set its associated exposed parameters.

### 5.2.4 Changing Defaults For Layouts

The default values for both exposed and non-exposed parameters of a layout style may be modified. For exposed parameters these refer to the defaults in "Layout Properties" window. For non-exposed ones, the only way to change a layout parameter is by updating the default values in the associated section of the corresponding layout class.

For example, suppose we want to change the default vertical spacing (spacing between levels) for our hierarchical (Sugiyama) layout. Then in the `SgymLayout` class, simply go to the "Class Constants" section and change the default value of the constant `DEFAULT_VERTICAL_SPACING` to its new value as shown below.

```
// ------------------------------------------------------------------------
// Section: Class Constants
// ------------------------------------------------------------------------
  public final static int DEFAULT_HORIZONTAL_SPACING = 100;

  public final static int DEFAULT_VERTICAL_SPACING = 120; // was 80

  public final static boolean DEFAULT_VERTICAL = true;
```

# Chapter 6

# Layout in Chisio

Automatic layout of graphs is extremely important with most graphs as it becomes a cumbersome operation, if not impossible, to manually layout the nodes of a graph and route the edges to produce aesthetically pleasing drawings. Chisio provides a number of popular and useful layout styles as described here.

Each layout style can be customized through the "Layout Properties" window, accessible through "Layout — Properties" in the top menubar or under the graph pop-up menu. Each style gets a tab of its own for its available options.

## 6.1 CoSE Layout

CoSE (Compound graph Spring Embedder) layout is an algorithm specifically designed for compound graphs [7]. It has been designed by our group, based on the traditional force-directed layout scheme with extensions to handle multi-level nesting, varying node sizes, and possibly other application-specific constraints.

An expanded node and its associated nested graph are represented as a single entity, similar to a "cart", which can move freely in orthogonal directions (no rotations allowed). Multiple levels of nesting is modeled with smaller carts on top of larger ones.

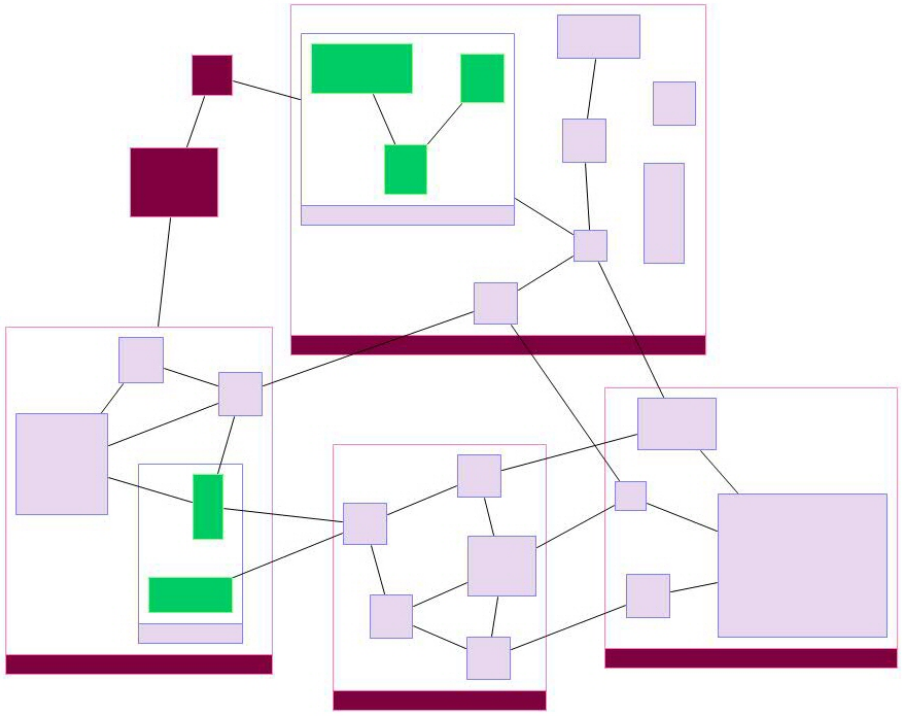

Figure 6.1: Sample CoSE layout produced by Chisio

### 6.1.1 CoSE Layout Steps

- Initialize data structures, perform random positioning for static layout and reduce trees for efficiency;
- Lay out the remaining, "skeleton graph" using a modified spring embedder as described earlier;
- Grow trees and continue iterating over the spring embedder model;
- Polish layout during this stabilization phase using a cooling schema for convergence.

### 6.1.2 Layout Options

CoSE layout options are shown in Figure 6.2.

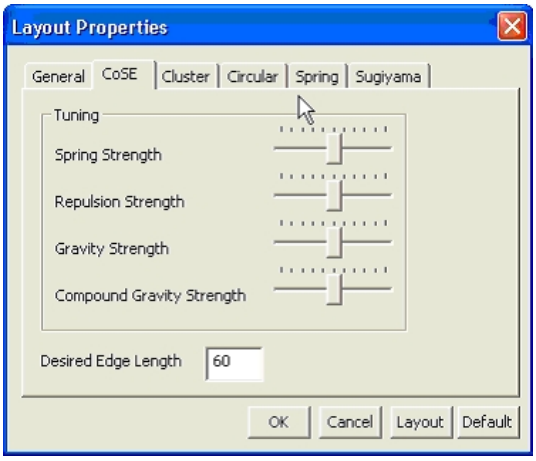


Figure 6.2: CoSE layout options

- **Spring Strength:** Constant for spring forces corresponding to edges; the higher this value is the stronger spring forces will be.
- **Repulsion Strength:** Constant for repulsion forces that are applied to node pairs; the lower this value is, the closer the nodes will be.
- **Gravity Strength:** Factor for the gravity of each graph; the higher this value is, the closer disconnected parts of the graph will be.
- **Compound Gravity Strength:** The factor for the gravity inside compound nodes; the higher this value is, the closer disconnected parts of the graph inside compound nodes will be.
- **Desired Edge Length:** Desired length of an intra-graph edge; inter-graph edges are allowed to be longer as they need to span their owner compound bounds.

### 6.1.3 CoSE Layout Improvements

#### 6.1.3.1 Bendpoint Support

We have implemented the bendpoint support for CoSE layout. Multi-edges and loop edges are considered in layout operation if the corresponding parameter is

enabled from layout options dialog (Figure 6.3).

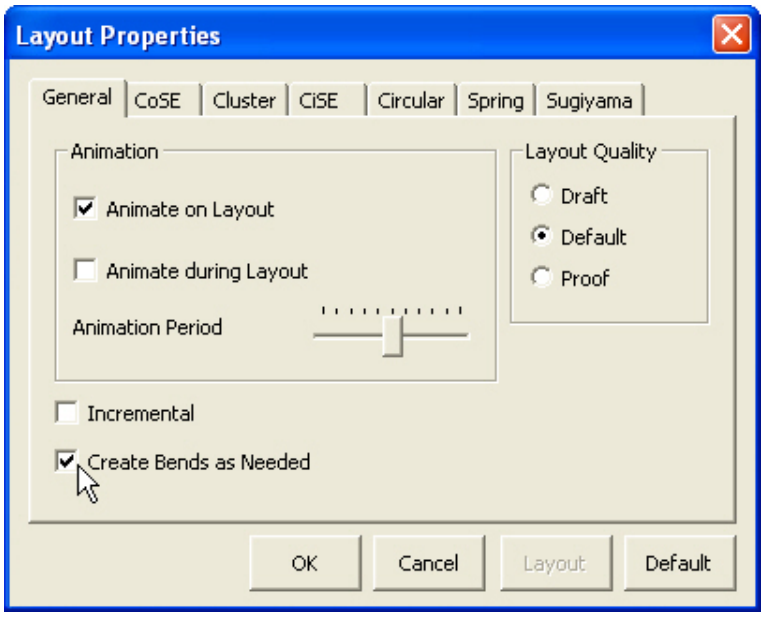


Figure 6.3: Create bendpoints as needed checkbox

Before beginning the layout algorithm, loop edges and multi edges are found. For each loop edge, two dummy nodes are created and for each multi-edge, one dummy node is created. Then CoSE layout is run for the graph including the new created dummy nodes. At the end of layout, each of created dummy nodes are replaced with a bendpoint (Figure 6.4).

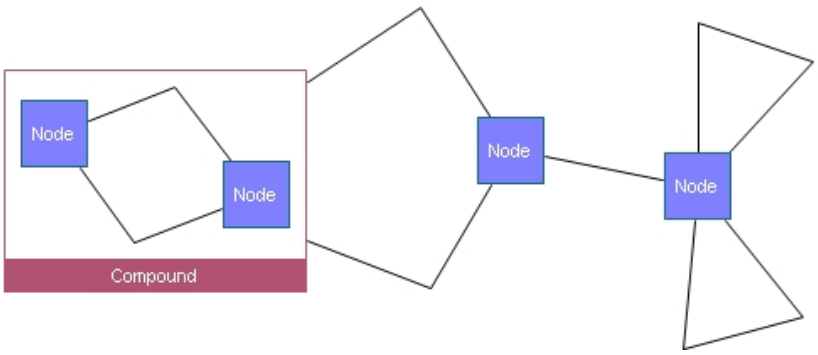


Figure 6.4: Bendpoints are automatically created for loop edges and multi edges

## 6.2 Cluster Layout

Many applications group or cluster the nodes in graphs to refer to say molecules with similar functionality or network devices in a LAN. Naturally users of such graphs would like the nodes in the same cluster to be placed near each other.

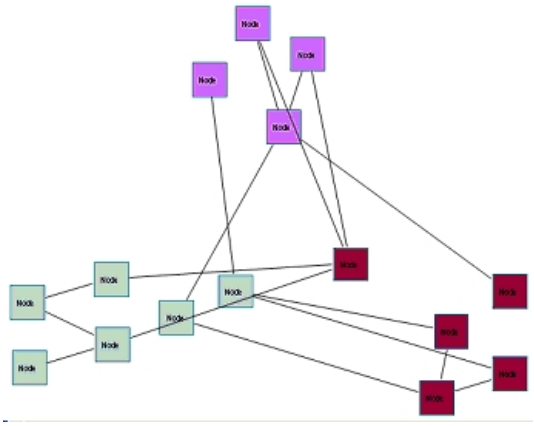

Figure 6.5: Sample Cluster layout produced by CHISIO, where nodes are color-coded by their clusters

Cluster layout uses the CoSE layout algorithm to establish this as follows:

### 6.2.1 Cluster Layout Steps

- Create a dummy compound node for each cluster;
- Move clustered nodes into these dummy compounds according to their cluster information (cluster IDs);
- Run CoSE layout for this graph;
- Remove dummy compounds and leave nodes in these compounds in their current absolute locations.

### 6.2.2 Layout Options

Cluster layout options are shown in Figure 6.6.

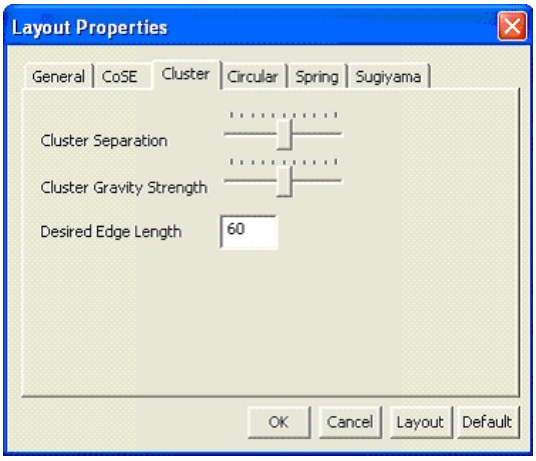


Figure 6.6: Cluster layout options

- **Cluster Separation:** Distance between neighboring clusters.
- **Cluster Gravity Strength:** The gravity inside clusters; the higher this value is, the closer the nodes in each cluster will be.
- **Desired Edge Length:** Desired length of an intra-graph edge; inter-graph edges are allowed to be longer as they need to span their owner compound/cluster bounds.

### 6.2.3 Cluster Layout Details

We have a well optimized layout style, CoSE layout, for compound graph layout. By using it, a group of nodes can be separated from others by simply adding them into a dummy compound node. This idea helps us to create a cluster layout for a set of clustered nodes.

First, nodes are moved into dummy compound nodes according to their cluster information. Then CoSE layout is run for the graph with dummy compound nodes inside. At the end, each nodes' absolute locations are kept while dummy compound nodes are removed.

## 6.3 CiSE Layout

A popular way to draw clustered graphs is in a circular fashion. In other words, a circle of appropriate size is created for each cluster and the nodes in that cluster are placed around this circle trying to minimize edge crossings. Circular layout algorithms address the issue of placing nodes of a cluster nicely around a circle, trying to minim

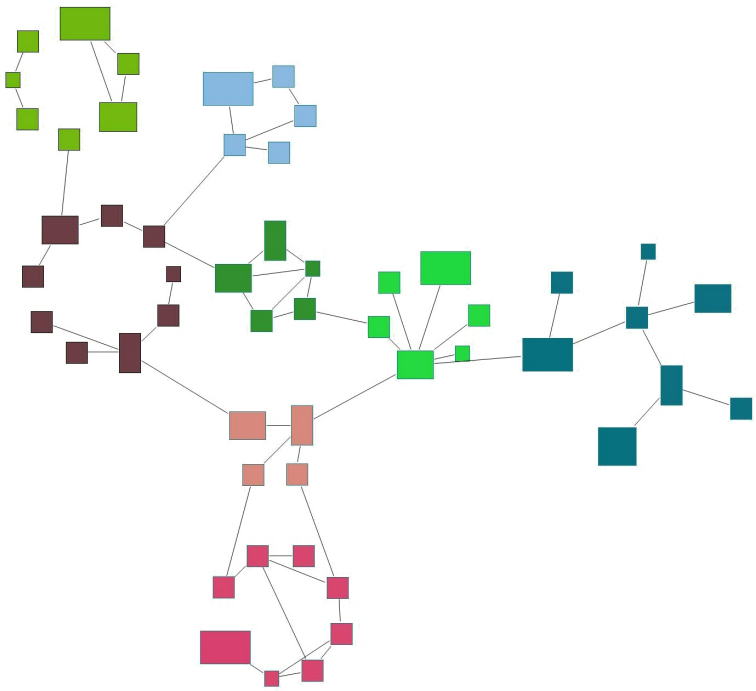

Figure 6.7: Sample CiSE layout produced by CHISIO, where nodes are color-coded by their clusters

### 6.3.1 CiSE Layout Steps

- Lay out individual clusters using Circular (AVSDF) layout as described on page 24;
- Do an initial placement of the cluster graph using a basic spring-embedder;
- Perform a modified spring embedder, where clusters/circles are allowed to move and rotate;

- Perform a modified spring embedder, where nodes in each cluster are also allowed to move (swap).

### 6.3.2 Layout Options

CiSE layout options are shown in Figure 6.8.

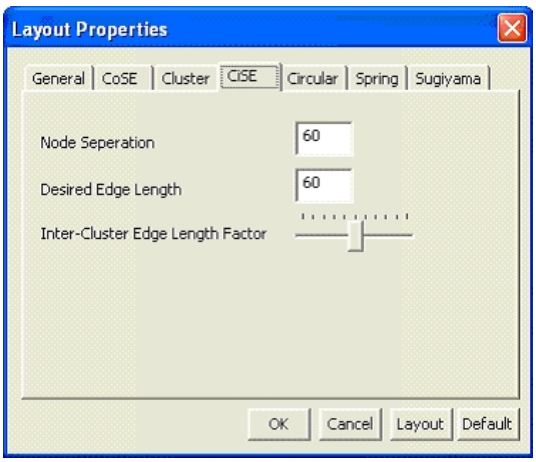


Figure 6.8: CiSE layout options

- **Node Separation:** Distance between neighboring nodes in a cluster.
- **Desired Edge Length:** Desired length of an intra-graph edge.
- **Inter-Cluster Edge Length Factor:** The proportion of inter-cluster edge length to regular edges. The higher this value is over 1.0, the longer inter-cluster edges will be with respect to intra-cluster edges.

## 6.4 Circular Layout

This layout algorithm can be used as the base of such a clustered graph layout algorithm. It is based on that of He and Sykora [12] and places all nodes of the current graph around a single circle, taking into account non-uniform node dimensions and desired node separation parameter.

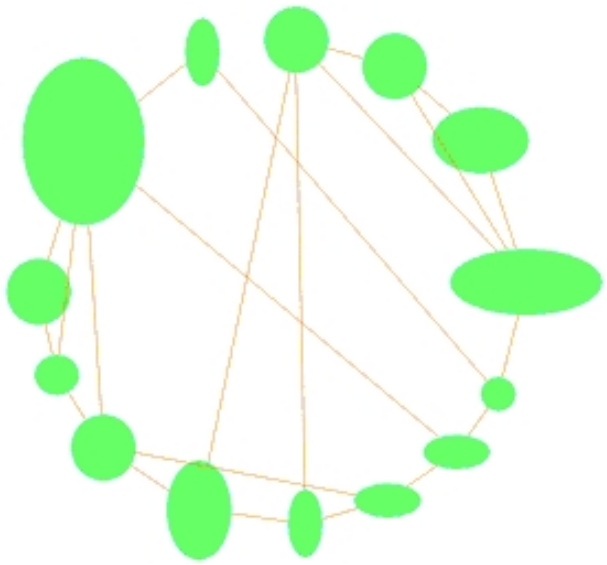

Figure 6.9: Sample Circular layout produced by Chisio

### 6.4.1 Circle Layout Steps

- Radius of the circle is calculated by using the total number of nodes, their dimensions and desired spacing between neighboring nodes;
- Each node is located onto the circle by AVSDF (adjacent vertex with smallest degree first) approach;
- Numbers of edge crossings are reduced with a post processing step.

### 6.4.2 Layout Options

Circular layout options are shown in Figure 6.10.

- **Node Separation:** Desired distance between two neighboring nodes on the circle

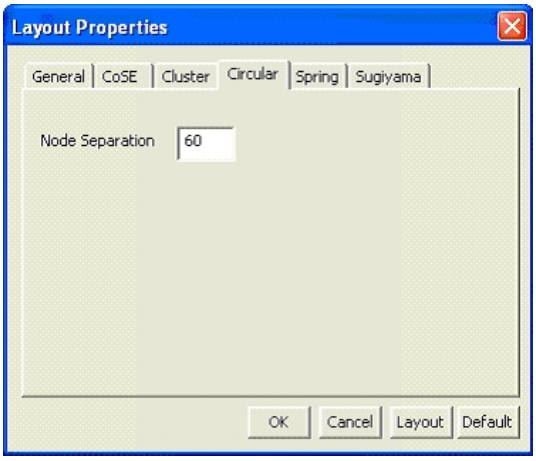


Figure 6.10: Circular layout options

## 6.5 Spring Layout

The spring layout is perhaps the simplest force-directed layout. Here the edges of the network are represented as springs. All the springs have a natural length, measured in the same units as the screen co-ordination system, which they attempt to achieve constantly. If the spring is shorter than its natural length it extends, pushing the nodes at either end of the edge apart. If the spring is longer than its natural length it contracts, pulling the nodes at either end together. The force exerted by the spring is proportional to the difference between its current length and its natural length. Nodes linked together tend to form cluster so a repulsive force is also added. The lengths are changed iteratively to obtain a well spaced out layout by minimizing the total energy.

We use an implementation by the GINY graph library [1], based on the Kamada-Kawai Spring Layout Algorithm [14] for this style.

### 6.5.1 Understanding the Spring Layout Algorithm

- Compute distances between nodes by using shortest paths algorithm;
- At each iteration, move a node in the direction, which decreases the total energy most.

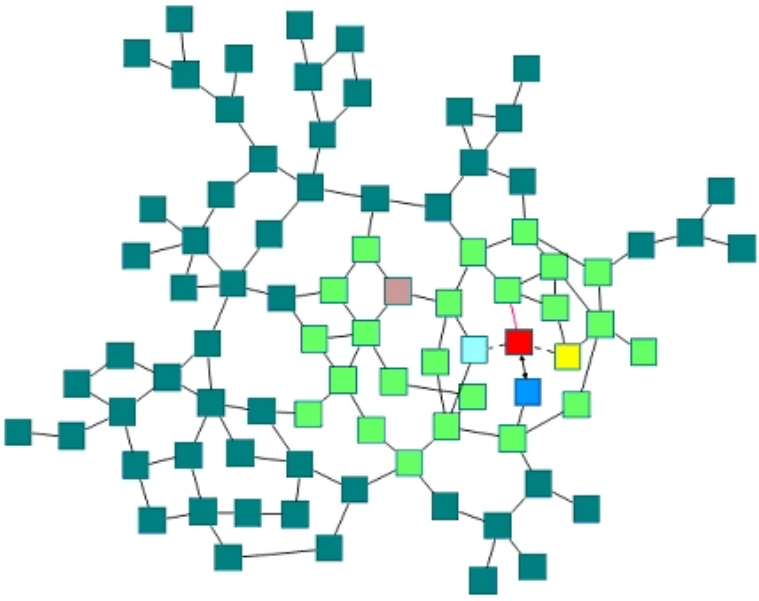

Figure 6.11: Sample Spring layout produced by CHISIO

### 6.5.2 Layout Options

Spring layout options are shown in Figure 6.12.

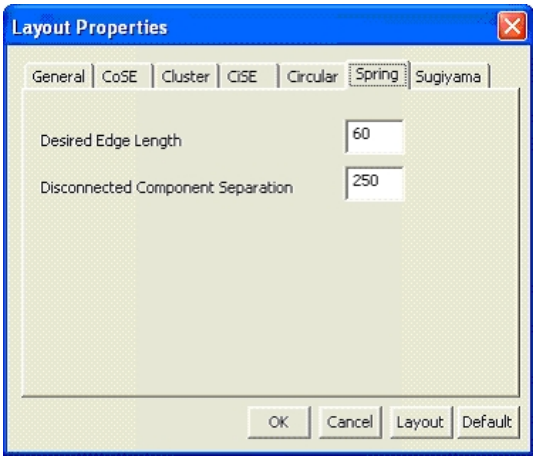


Figure 6.12: Spring layout options

- **Desired Edge Length:** Desired length of an edge
- **Disconnected Component Separation:** Adjusts the separation of disconnected parts of the graph

## 6.6 Hierarchical Layout

The hierarchical layout is by far the most popular layout style for directed graphs, emphasizing the precedence relationships among graph objects. This layout can represent organizational and information management system dependencies, as well as process models, software call graphs, and workflows. Our hierarchical layout is an implementation of the Sugiyama algorithm [16], taken from JGraph [2]. We have extended the implementation to support compound graphs and variable node dimensions.

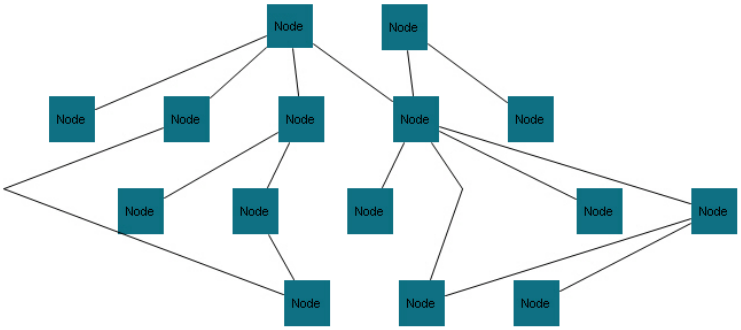


Figure 6.13: Sample Hierarchical layout produced by Chisio

### 6.6.1 Hierarchical Layout Steps

The hierarchical layout algorithm divides the nodes of a graph into levels and directs edges from top to bottom in a drawing oriented top-to-bottom as follows:

- For compound node support, inter-graph edges are converted into intra-edges;
- All roots in the graph are identified;
- By using these roots, nodes are topologically sorted with a DFS. If the graph contains cycles, the edges causing the cycles are reversed;
- Calculated topological information is used to create levels;

- If bendpoints are allowed to be created, bendpoints are created for multi level edges;
- Number of edge crossings between neighboring levels are reduced via the barycenter method;
- Levels of nodes are adjusted to minimize edge lengths;
- Spacing between levels and nodes are calculated;
- These values are converted into specific positions.

### 6.6.2 Layout Options

Hierarchical layout options are shown in Figure 6.14.

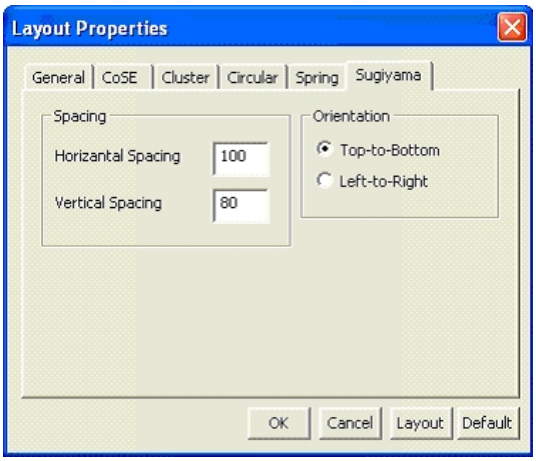


Figure 6.14: Hierarchical layout options

- **Horizontal Spacing:** The space between two nodes in the same level
- **Vertical Spacing:** The space between two subsequent levels
- **Orientation:** Whether the drawing should from left-to-right (roots at the left, leaves at the right of the drawing) or top-to-bottom.

### 6.6.3 Hierarchical Layout Improvements

#### 6.6.3.1 Compound Graph Support

Compound support is a basic feature and CHISIO's main motivation. Because of that, compound support in layout styles is an important issue. But the original implementation of Sugiyama layout did not support the compound graphs. By making some extensions to the Sugiyama layout code of JGraph library, we have successfully added support for compound graphs.

Firstly, the main layout algorithm is changed into a recursive one so that it can support nested calls. By checking each node of the current graph if it is a compound node and calling the main layout algorithm for compound nodes' subgraphs in a recursive fashion, compound graph support is achieved.

For inter-graph edges, a pre-processing is implemented. They are carefully converted to intra-graph edges before layout algorithm is applied. In this way, the relation between different graphs (e.g. root graph and a compound node's child graph) is disconnected. Each graph can be laid out hierarchically as others do not exist. Thus, we can use recursive algorithm without difficulty.

Depth first manner is used while recursively processing nested graphs. The graph at the end of nesting is laid out first. Then, that graph's parent compound node behaves like a simple node for its parent graph with the new calculated dimensions. This goes on to the root graph recursively (Figure 6.15).

#### 6.6.3.2 Edges Between Nodes of Same Level

There was not any check for the edges between two nodes of the same level. To solve this problem, many solutions are proposed about layer assignment. Finally, *The Longest Path Layering* algorithm is used to assign the layers [5]. This algorithm runs in linear time and the height of the layering is minimal. Also it is guaranteed that there is no edges between two nodes of the same level.

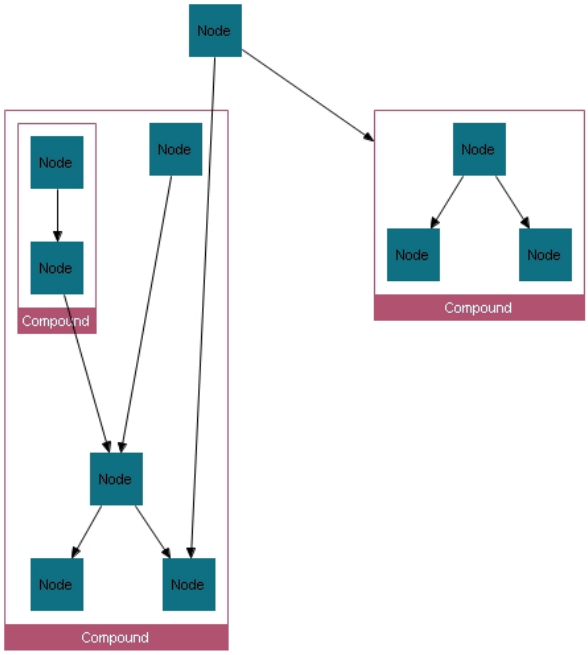


Figure 6.15: Hierarchical layout with compound support

But this layering approach may result in drawings with very long edges. To handle this problem, we have added an intermediate step to shorten the edge lengths via changing levels of some nodes by pushing up the nodes starting from the 2nd level (Figure 6.16).

#### 6.6.3.3 Non-Uniform Node Size Support

The original implementation of Sugiyama layout did not support non-uniform node sizes. When there are nodes with different sizes, overlapping nodes occur. To overcome this problem, maximum node dimensions are used in the calculation of levels and grids locations (Figure 6.17).

#### 6.6.3.4 Handling Cycles

When there are cycles, the original implementation was giving an error indicating that this layout can not be run for DAG (Directed Acyclic Graph). We have

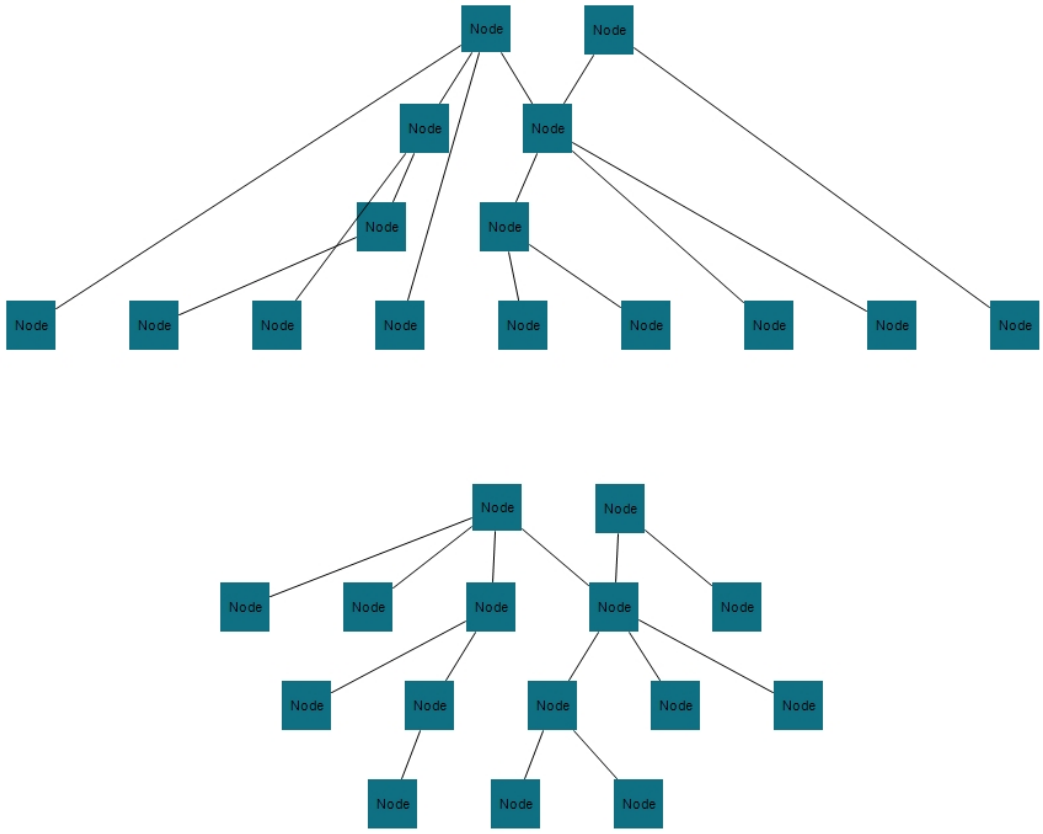


Figure 6.16: Same graph without (top) and with (bottom) improvement in Hierarchical layout by pushing up nodes

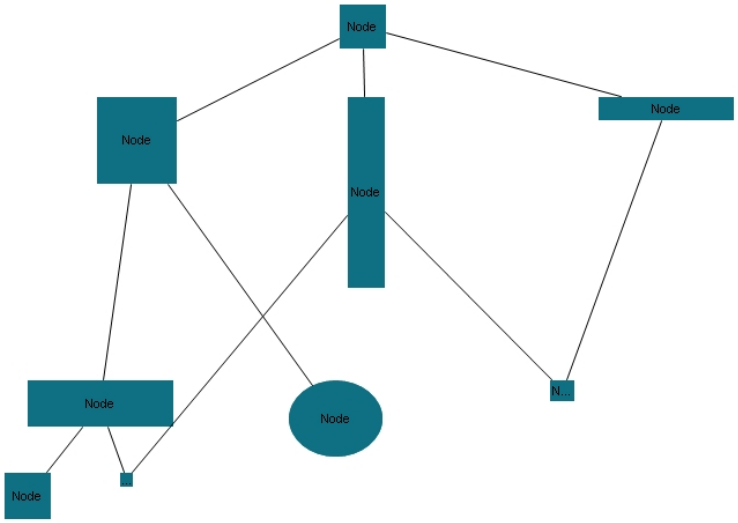


Figure 6.17: Non-uniform node size support in Hierarchical layout

handled cyclic graphs by reversing edges that create cycles as we traverse the graph to assign level numbers.

#### 6.6.3.5 Bendpoint Support

We have implemented the bendpoint support for multi-level edges which can be enabled from layout options dialog.

After assigning the levels for each node, multi-level edges are found. For each multi-level edge, dummy nodes are created amount of level differences between source and target nodes. Dummy nodes are located to the related levels. At the end of layout, each of created dummy nodes are replaced with a bendpoint (Figure 6.18).

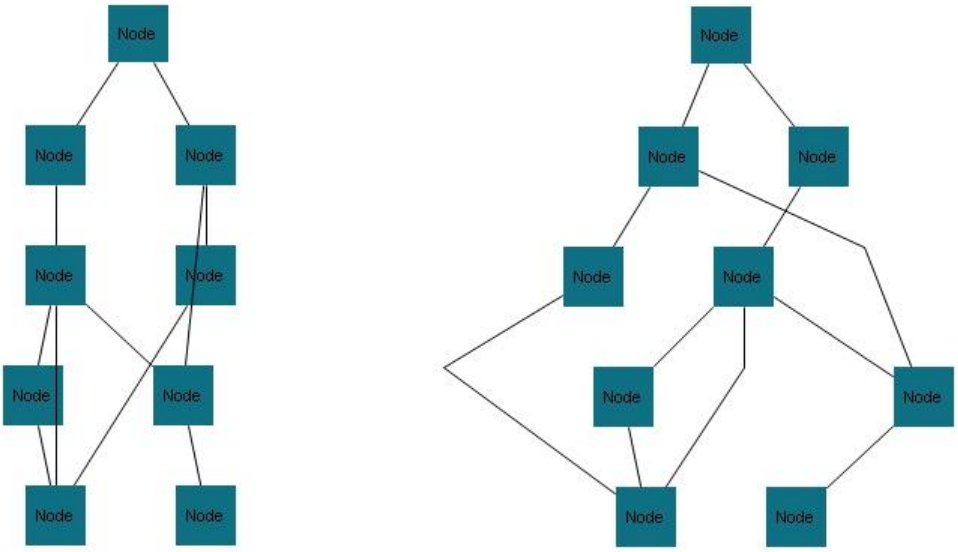


Figure 6.18: Same graph without (left) and with (right) bendpoints in Hierarchical layout

# Chapter 7

# Conclusion

In this thesis, a new open-source, free, easy-to-use compound graph editing and layout tool is introduced. This tool, named Chisio, can be used as a graph editor by end users and as a customizable graph editing tool by software developers or researchers for specific purposes. Chisio is specially designed for easily extending both user interface (UI) and layout algorithms.

Chisio has several layout algorithms which are already implemented. These styles are enough to optimize the drawings for most types of graphs. But one may want to add their own layout algorithm.

## 7.1 Contribution

There already exist many commercial and non-commercial graph visualization tools. Some of them, especially commercial ones, support compound graphs as well. One who wants to use a non-commercial freeware graph tool with compound support, can only find limited support at best. Chisio fills an important gap in this field with its full compound support. Besides, its layout capabilities for compound graphs is unique with an implementation of the latest algorithms developed in-house and improvements to popular implementations to support

compounds.

Chisio is an open source graph visualization tool. Its source code can be easily customized to add new facilities to Chisio. This gives the ability to create specialized graph visualization tools based on Chisio.

Developers can also use Chisio for their layout algorithm development. Animation ability makes tracking of the layout steps easily.

## 7.2 Future Work

The first version of Chisio has included many standard and advanced editing and layout features. However there is still a lot of room for improvement for future versions.

As it is easy to extend Chisio, number of node types and edge types can be increased to support wide range of graph drawings. Different layout algorithms like Radial and Orthogonal can be added. Chisio supports GraphML graph file format. Other formats such as GXL and GML can be supported as well. Expand/collapse support for compound nodes can be implemented to extend the compound support of Chisio. Do/redo mechanism must be developed for Chisio to make it a comprehensive editor and tolerate mistakes. Diagramming features like multiple object inspector, panning tool, alignment of multiple nodes, exposing positions in the inspectors for non-interactive modifications and moving selections slowly using SHIFT+CTRL or ALT keys can be added.

# Bibliography


[1] GINY Graph Library. http://csbi.sourceforge.net.

[2] JGraph. http://www.jgraph.com.

[3] U. Brandes, M. Eiglsperger, I. Herman, M. Himsolt, and M. S. Marshall. GraphML progress report. In *Graph Drawing*, pages 501–512, 2001.

[4] Chisio User's and Programmer's Guides. i-Vis Research Group, Bilkent University, Ankara, Turkey. http://www.cs.bilkent.edu.tr/ ivis/chisio.html.

[5] G. Di Battista, P. Eades, R. Tamassia, and I. G. Tollis. *Graph Drawing, Algorithms for the Visualization of Graphs*. Prentice-Hall, 1999.

[6] U. Dogrusoz, Q. Feng, B. Madden, M. Doorley, and A. Frick. Graph visualization toolkits. *IEEE Computer Graphics and Applications*, 22(1):30–37, January/February 2002.

[7] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir. A compound graph layout algorithm for biological pathways. In *Graph Drawing (Proc. GD '04)*, volume 3383 of *Lecture Notes in Computer Science*, pages 442–447. Springer-Verlag, 2004.

[8] E. Gamma, R. Helm, R. Johnson, and J. Vlissides. *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley Professional, 1995.

[9] Graphical Editing Framework version 3.1. The Eclipse Foundation, 2007. http://www.eclipse.org/gef.

[10] Graphlet, A toolkit for graph editors and graph algorithms. University of Passau, Passau, Germany. http://www.infosun.fmi.uni-passau.de/Graphlet.

[11] Graphviz - Graph Visualization Software. http://www.graphviz.org.

[12] H. He and O. Sykora. New circular drawing algorithms. In *Workshop on Information Technologies - Applications and Theory (ITAT04)*, 2004.

[13] JViews User's Guide. ILOG SA, 94253 Gentilly Cedex, France, 2002. http://www.ilog.com.

[14] T. Kamada and S. Kawai. An algorithm for drawing general undirected graphs. *Information Processing Letters*, 31(1):7–15, April 1989.

[15] M. Kaufmann and D. Wagner, editors. *Drawing Graphs: Methods and Models*, volume 2025 of *Lecture Notes in Computer Science*. Springer, 2001.

[16] K. Sugiyama, S. Tagawa, and M. Toda. Methods for visual understanding of hierarchical systems. *IEEE Transactions on Systems, Man, and Cybernetics*, 21(2):109–125, February 1981.

[17] Graph Layout Toolkit and Graph Editor Toolkit User's Guide and Reference Manual. Tom Sawyer Software, Oakland, CA, USA, 1992-2002. http://www.tomsawyer.com.

[18] uDraw(Graph), 2005. http://www.informatik.uni-bremen.de/uDrawGraph.

[19] VGJ, Visualizing Graphs with Java. Auburn University, Auburn, Alabama. http://www.infosun.fmi.uni-passau.de/Graphlet.

[20] yFiles User's Guide. yWorks GmbH, D-72076 Tbingen, Germany, 2002. http://www.yworks.com.