version

2.0

BILKENT UNIVERSITY

i-Vis Information Visualization Research Group

Chisio Layout Programmer's Guide

BILKENT I-VIS RESEARCH GROUP

Chisio Layout Programmer's Guide

Computer Engineering Department • Bilkent University
Ankara 06800, TURKEY
Phone +90 312.290.1401 • Fax +90 312.266.4047

Table of Contents

Chapter 1

Overview

In this chapter, we will introduce the Chisio Layout library and give an overview of its system architecture.

Chisio Layout is an open source library, which can be used by interactive graph editing tools to automatically layout compound, clustered or hierarchically structured graphs as well as simple, flat graphs. It can be used both locally and can be set up to be used remotely. Several popular layout algorithms are already implemented in this library, and may be modified as needed (see section on Existing Layout Styles on page 3). You may also add layout styles of your own (see section on Adding New Layout in page 12). The sources of Chisio Layout 2.0 can be found in sourceforge (`https://chilay.svn.sourceforge.net/svnroot/chilay`) under the directory named `chilay20`.

System Architecture

The core structure of the **layout level** (l-level) is shown in Figure 1. `LGraphObject` is the base class for all l-level graph objects, and it holds a reference to the **view level** (v-level) through a field named `vGraphObject`. Each graph consists of nodes and edges. Each node has a back reference to its owner graph. A node can own a graph, named its child graph, meaning this node is a compound one. Each edge has a source and a target node, and nodes hold their incident edges.

A graph manager is responsible for holding the entire topology, all the structure needed for the layout process. It maintains a collection of graphs, forming a compound graph structure through inclusion, and maintains the inter-graph edges. The `Layout` class is responsible for laying out the associated graph model (which is stored in a graph manager). The layout framework also lets the users associate each l-level node and edge with a v-level node and edge, respectively. It makes the necessary callbacks (to update methods) so that the results can be copied back to the associated v-level objects when layout is finished. v-level objects, which can be manipulated during layout, may implement the `Updatable` interface (in package `org.ivis.layout`). Users are also given an opportunity to perform any pre- and post-layout operations with respective methods. Finally, both layout and graph manager classes hold references to each other.

In Figure 1, it is also shown how a layout style (such as Compound Spring Embedder (CoSE) Layout) inherits from and extends the respective l-level classes. You will find more information about adding a new layout style in section titled Adding New Layout on page 12.

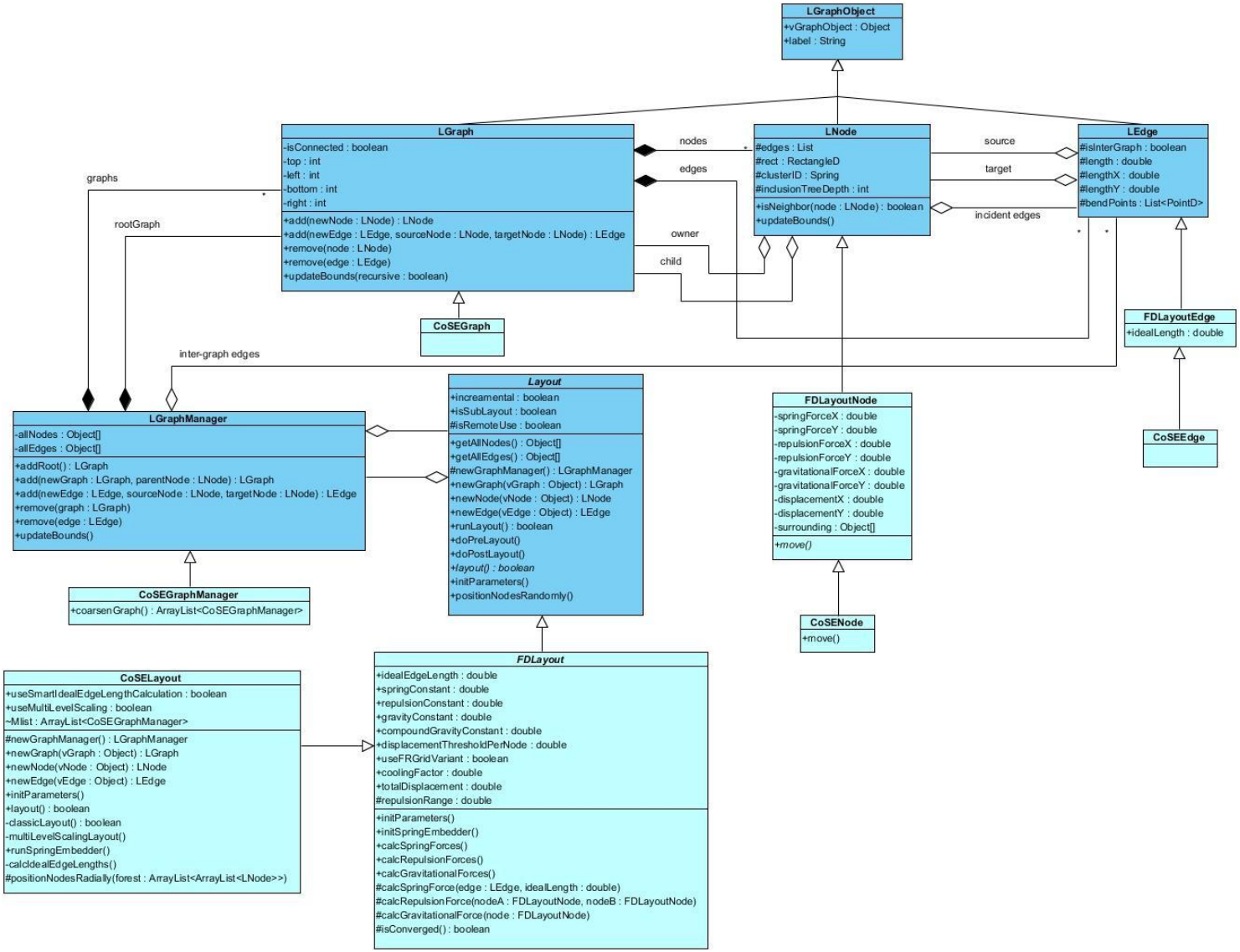

Figure 1 L-level architecture

Chapter 2

Layout & Customization

In this chapter, we introduce the existing layout styles and give instructions on how to add layout styles of your own.

Automatic layout process plays an important role on effective analysis of especially large amounts of relational information. Many methods have been proposed for automated layout of a graph representing relational information. We have implemented some of these within the scope of the Chisio Layout project. At the end of section, you will also find how to add a layout style of your own to the project.

Existing Layout Styles

Force Directed Layout

Our force-directed layout methods are based on that of Fruchterman and Reingold [1]. This core package does not constitute a layout style of its own but is solely used by other layout styles as a basis. In a force-directed layout, vertices behave like charged particles while edges behave like springs that connect these particles. All particles have the same sign so they only repel each other. This analogy is sufficient to provide the following conditions:

1. Neighbor vertices (which are connected by an edge) should be placed close to each other.

2. All vertices should be distributed well onto the screen.

Additionally, the gravitational force, exerted to all vertices, is introduced to keep the disconnected components

This physical system is implemented iteratively. All vertices are placed randomly at the beginning. Spring, repulsion and gravitational forces that are exerted to each vertex is calculated and applied at each iteration. Iterations continue until the system reaches the global minima (a stable state that all vertices move very slightly).

Force-directed placement schemes are easy to understand and implement, but it is hard to tune its parameters (spring, repulsion, constant, etc.). Constants with their default values can be found in "`src.org.ivis.layout.fd.FDLayoutConstant.java`" class. You can change these constants and experience how resulting layout changes. For example, increasing spring strength results relatively better layouts but consumes more time to be converged. Also increasing compound gravity strength (or decreasing compound gravity range factor) results relatively wider compound nodes (for non-empty ones).

Currently, two layout schemes are extended (implemented) from force directed layout: "Compound Spring Embedder" (CoSE) and "Circular Spring Embedder" (CiSE). You can find detailed information about those layouts in the following sections.

Compound Spring Embedder (CoSE) Layout

Most of the layout algorithms including [1] consider vertices as points on 2D or 3D space, and edges as straight lines between vertices. However, such an analogy is insufficient to represent complex relational real-life data. So, in [2], a layout algorithm that is based on a force-directed placement scheme is proposed with compound node support. The algorithm in [2] also allows inter-graph edges, and supports non-uniform node sizes.

CoSE constants with their default values can be found in "`src.org.ivis.layout.cose.CoSEConstant.java`" class. With the purpose of improving both visual quality and performance of [2], a multi-level scaling method [3] has been adopted. You can easily enable/disable multi-level scaling method by changing the corresponding variable in the `CoSEConstant` class.

Circular Spring Embedder (CiSE) Layout

CiSE [4] is used for layout of clustered graphs, where nodes in each cluster are drawn around a circle. CiSE is an extension to [1] and it allows moving and rotating nodes in the same cluster as a group. CiSE layout algorithm makes use of other layout algorithms like AVSDF Layout and Compound Spring Embedder (CoSE) Layout.

CoSE constants with their default values can be found in "`src.org.ivis.layout.cise.CiSEConstant.java`" class.

Cluster Layout

Similar to CiSE layout, Cluster layout style extends CoSE layout to implement a layout algorithm for clustered graphs. It arranges the nodes in the same cluster to be "close to each other" but doesn't explicitly place them around a particular shape like a circle.

AVSDF Layout

AVSDF layout class implements the circular drawing algorithm proposed by He and Sykora [5], and places all nodes of the input graph around a **single** circle, regardless of their clustering information.

AVSDF constants with their default values can be found in "`src.org.ivis.layout.avsdf.AVSDFConstant.java`" class.

Sugiyama Layout

Sugiyama layout class lays out the input graph using the hierarchical Sugiyama Layout Algorithm described in [6]. This algorithm is useful for rooted or hierarchically structured graphs (including trees).

Sugiyama constants with their default values can be found in "`src.org.ivis.layout.sgym.SGYMConstant.java`" class

Spring Layout

Spring layout class lays out the given input graph using the Spring Embedder taken from the GINY library [12].

Spring constants with their default values can be found in "`src.org.ivis.layout.spring.SpringConstant.java`" class

Adding New Layout

All of the existing layouts extend the `Layout` 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 `Layout` 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 Layout
{
    public RandomLayout()
    {
        super();
    }

    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 `initParameters` method, which is also in `Layout`.

In case we need layout-style specific nodes or edges for our layout algorithm, we should extend `LNode` or `LEdge` classes, respectively, and override the methods `newNode` or `newEdge` in `Layout` as shown below to make sure specialized node and edge instances are instantiated, respectively.

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

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

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

    ...

}

import org.ivis.layout.LEdge;
import org.ivis.layout.LNode;

public class RandomEdge extends LEdge
{
    public RandomEdge(LNode source, LNode target, Object vEdge)
    {
        super(source, target, vEdge);
    }
}
```

```
public class RandomLayout extends Layout
{
    public RandomLayout()
    {
        super();
    }

    public LNode newNode(Object vNode)
    {
        return new RandomNode(this.graphManager, vNode);
    }

    public LEdge newEdge(Object vEdge)
    {
        return new RandomEdge(null, null, vEdge);
    }

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

It is not necessary to extend all l-level classes in some cases. For example, if you think using l-level edge is sufficient to cover all edge properties and functionality for your new layout style, then you do not have to extend the l-level edge.

New Layout Options

`LayoutOptionsPack` is a class for maintaining the layout parameters that can be customized by the user (for example: by a graphical interface). 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(
            LayoutConstants.DEFAULT_ANIMATION_DURING_LAYOUT);
        general.setAnimationOnLayout(
            LayoutConstants.DEFAULT_ANIMATION_ON_LAYOUT);

        ...

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

    public General getGeneral()
    {
        return general;
    }

    ...

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

Chapter 3

Local vs. Remote Usage

In this chapter, you will find detailed information about configuring Chisio Layout for both local and remote usages.

Chisio Layout can be set up for local or remote usage. It can be imported in an existing graph editor as a library to provide the layout services. Alternatively, it can be deployed to a server to provide remote layout services.

Local Usage

As mentioned earlier, we assume two separate levels to represent the layout and view related information of a graph layout (l-level) and view (v-level). Each v-level graph object is associated with an l-level one. In order to use the implemented layout styles, you should create the corresponding l-level topology from your v-level structure. Assume that, class structure of your v-level graph objects is as shown in Figure 2.

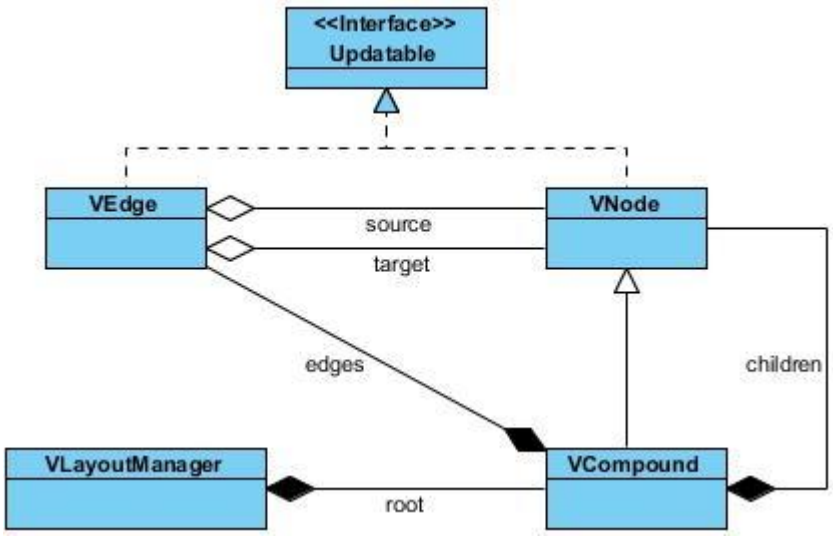

Figure 2 Sample v-level architecture

Figure 3 explains with an example how a compound graph is assumed to be represented at view (v-level) and how the corresponding layout (l-level) level should be formed for layout purposes.

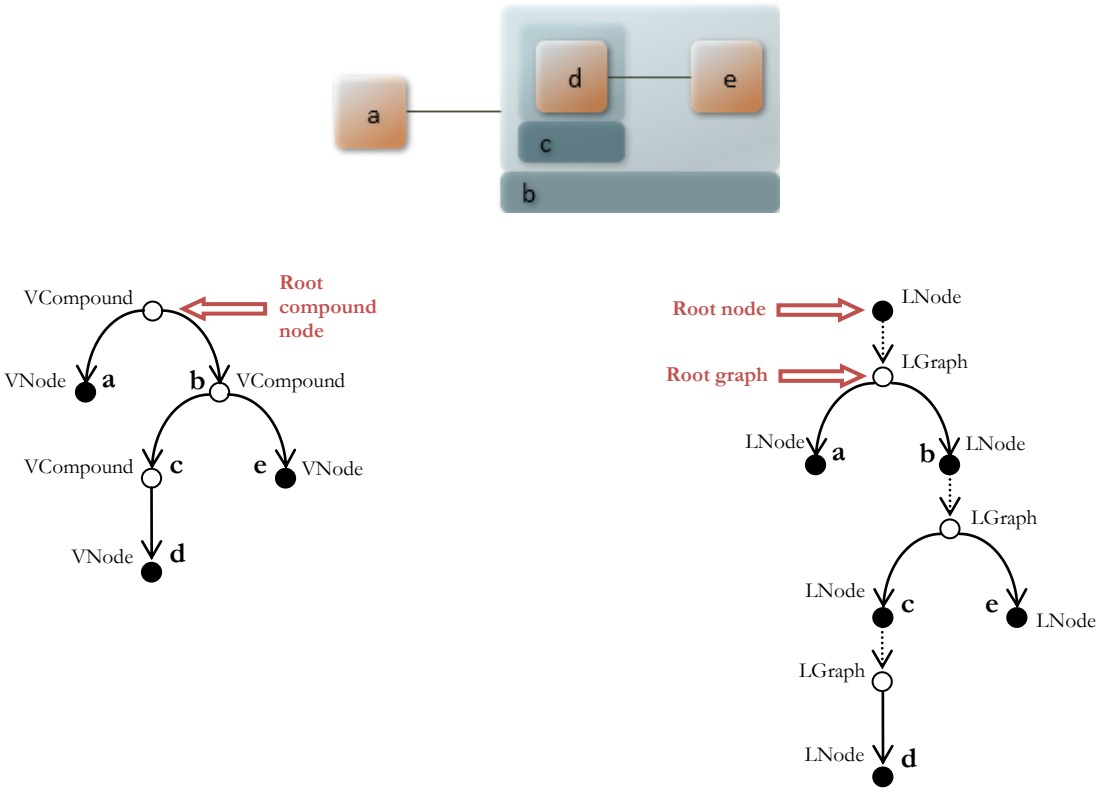

Figure 3 A compound graph **(top)** and its representation at view (v-level) **(bottom-left)** and layout (l-level) **(bottom-right)** levels.

Creating l-level from v-level

Sample code segment for creating the l-level topology from a v-level structured as in Figure 2 and Figure 3 is shown below.

```
import org.ivis.layout.*;

// v-level layout manager
public class VLayoutManager
{
    private VCompound root;
    private Layout layout;

    // mapping between view and layout level
    private HashMap<VNode, LNode> viewToLayout;
    private HashMap<LNode, VNode> layoutToView;
    ...

    public void createTopology()
    {
        // for this sample, we will use CoSE Layout style
        this.layout = new CoSELayout();

        LGraphManager graphMgr = this.layout.getGraphManager();
        LGraph lRoot = graphMgr.addRoot();

        for (VNode vNode: this.root.children)
```

```
        {
            this.createNode(vNode, null, this.layout);
        }

        for (VEdge vEdge: this.root.edges)
        {
            this.createEdge(vEdge, this.layout);
        }

        graphMgr.updateBounds();

        this.layout.runLayout();
    }

    private void createNode(VNode vNode, VNode parent, Layout layout)
    {
        LNode lNode = layout.newNode(vNode);
        LGraph rootLGraph = layout.getGraphManager().getRoot();

        this.viewToLayout.put(vNode, lNode);
        this.layoutToView.put(lNode, vNode);

        // if the vNode has a parent, add the lNode as a child of the parent l-node.
        // otherwise, add the node to the root graph.

        if (parent != null)
        {
            LNode parentLNode = this.viewToLayout.get(parent);
            parentLNode.getChild().add(lNode);
        }
        else
        {
            rootLGraph.add(lNode);
        }

        // copy location
        lNode.setLocation(vNode.x, vNode.y);

        // copy cluster ID (zero means unclustered)

        lNode.setClusterID(Integer.toString(vNode.clusterID));

        // if vNode is a compound, then recursively create child nodes

        if (vNode instanceof VCompound)
        {
            VCompound vCompound = (VCompound) vNode;

            // add new LGraph to the graph manager for the compound node
            layout.getGraphManager().add(layout.newGraph(null), lNode);

            // for each VNode in the node set create an LNode
            for (VNode vChildNode: this.root.children)
            {
                this.createNode(vChildNode, vCompound, layout);
            }

        }
        else
        {
            lNode.setWidth(vNode.width);
```

```
            lNode.setHeight(vNode.height);
        }
    }

    private void createEdge(VEdge vEdge, Layout layout)
    {
        LEdge lEdge = layout.newEdge(vEdge);

        LNode sourceLNode = this.viewToLayout.get(vEdge.source);
        LNode targetLNode = this.viewToLayout.get(vEdge.target);

        this.layout.getGraphManager().add(lEdge, sourceLNode, targetLNode);
    }
}
```

Updatable Interface

The `Updatable` interface forces classes to implement the `update` method which takes an `LGraphObject` as input. For our sample v-level model, `VNode` and `VEdge` classes may implement the `Updatable` method. And the implemented update methods may look like the following:

```
// v-level node class
public class VNode implements Updatable
{
    public int x, y; // location
    public int width, height; // size
    public int clusterID; // if graph is clustered
    ...

    public void update(LGraphObject lGraphObj)
    {
        // Since this is the update method of a v-level node, it is assumed
        // that the given LGraphObject is an instance of LNode. So, cast
        // operation is performed without type checking.
        LNode lNode = (LNode) lGraphObj;

        // This update operation should be performed only if the node is not
        // a compound, compound node class performs a different operation
        // in its own update method.
        this.x = lNode.getRect().x;
        this.y = lNode.getRect().y;
    }
}

// v-level edge class
public class VEdge implements Updatable
{
    private VNode source, target;
    private List<PointD> bendPoints;
    ...

    public void update(LGraphObject lGraphObj)
    {
        // Since this is the update method of a v-level edge, it is assumed
        // that the given LGraphObject is an instance of LEdge. So, cast
        // operation is performed without type checking.
        LEdge lEdge = (LEdge) lGraphObj;

        // Copy bend points
        for(PointD point: lEdge.getBendpoints())
        {
```

```
            this.bendPoints.add(point.getCopy());
        }
    }
}

// v-level compound node class
public class VCompound extends VNode
{
    public List<VNode> children;
    public List<VEdge> edges;
    ...

}
```

`Layout` class makes the necessary callbacks (calls `update` methods for relevant graph objects which implement `Updatable` interface) so that the results can be copied back to the associated v-level objects when layout is finished.

However, if you do not want callbacks to the v-level update methods through the `Updatable` interface, you must **not** associate v-level objects with l-level ones. To do so, you must call `newNode` and `newEdge` methods of `Layout` class with the `null` object parameter as shown below:

```
LNode lNode = layout.newNode(null);
LEdge lEdge = layout.newEdge(null);
```

In this case, you are to explicitly call the `update` methods of all graph objects such as `VNode` and `VEdge` in Figure 2 (or methods with similar content if v-level objects do not implement the `Updatable` interface) after layout has finished.

If the `Updatable` interface is implemented and the associated flag is enabled, our `Layout` class facilitates developers to see the layout progress through animation, which can be rather useful for tracing/debugging existing and added layout algorithms. It is also possible to change the animation period by modifying the relevant constants in `LayoutConstants` class under "`org.ivis.layout`" directory:

```
public static final boolean DEFAULT_ANIMATION_DURING_LAYOUT = true;
public static final int DEFAULT_ANIMATION_PERIOD = 50;
```

For the above example, the view will be updated once every 50 iterations.

Remote Usage

Chisio Layout library can be deployed to a server to provide remote layout services. In order to use Chisio Layout as a remote service properly, you should save the geometry of your graphs in XML format which conforms to our XML schema (Figure 4). After saving geometry, saved XML must be sent to the dedicated server via an HTTP request. In the following sections, you will find both how to deploy Chisio Layout to your dedicated server and how to use Chisio Layout as a remote service.

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<view>
  <node id="ID_0" clusterID="0">
    <bounds height="40" width="40" x="0" y="0"/>
  </node>
  <node id="ID_1" clusterID="0">
    <bounds height="40" width="40" x="0" y="0"/>
  </node>
  <node id="ID_2" clusterID="0">
    <bounds height="280" width="240" x="0" y="0"/>
    <children>
      <node id="ID_3" clusterID="0">
        <bounds height="40" width="40" x="0" y="0"/>
      </node>
      <node id="ID_4" clusterID="0">
        <bounds height="40" width="40" x="0" y="0"/>
      </node>
    </children>
  </node>
  <edge id="ID_0">
    <sourceNode id="ID_0"/>
    <targetNode id="ID_1"/>
  </edge>
  <edge id="ID_1">
    <sourceNode id="ID_1"/>
    <targetNode id="ID_2"/>
  </edge>
  <edge id="ID_2">
    <sourceNode id="ID_3"/>
    <targetNode id="ID_4"/>
  </edge>
  <edge id="ID_3">
    <sourceNode id="ID_0"/>
    <targetNode id="ID_4"/>
  </edge>
</view>
```

Figure 4 A sample XML file that conforms to our XML Schema

Deploying to a Dedicated Server

In this section, we describe how to deploy Chisio Layout to a Tomcat Server using Ant Builder and optionally Eclipse.

First of all, make sure that JDK 1.5 or higher installed, and `JAVA_HOME` environment variable is set to the JDK installation directory. Download and extract Eclipse [7] and install SVN [8] plug-in to Eclipse. Then, download

Ant [9], extract its contents to `{ANT_HOME}` directory and add `{ANT_HOME}\bin` directory to the `Path` system variable of Windows (See Figure 5).

Figure 5 Editing Path system variable, `{ANT_HOME} = "C:\Program Files\ant\"` for this sample

Download latest Tomcat 5.5 "core" zip distribution from [10], extract its contents to `{TOMCAT_HOME}`. If it does not exist, create the file named "crossdomain.xml" and put it to {`TOMCAT_HOME}\webapps\ROOT` directory. The content of the xml file must look like the following:

```
<?xml version="1.0"?>
<cross-domain-policy>
    <allow-access-from domain="*" to-ports="*" secure="false"/>
</cross-domain-policy>
```

First we have to check-out the Chisio Layout project. If you prefer to use Eclipse, from the "File" menu, select "Import", then select "Check Projects from SVN" and press the "Next" button. Then check the "create a new repository location" and add `https://chilay.svn.sourceforge.net/svnroot/chilay`. Under the newly created repository location, select the `chilay20` folder and press "Next". When prompted, create a new project in the directory `{CHILAY_HOME}` and use Java SE-1.6 or a higher runtime environment.

Open `build.properties` file under `{CHILAY_HOME}`, set `catalina.home` variable to `{TOMCAT_HOME}` and `catalina.build` variable to `{DEPLOYMENT_HOME}`. Please note that, `{DEPLOYMENT_HOME}` directory must be under `{TOMCAT_HOME}\webapps`. Right-click on build.xml, and select "Run as > Ant build". (Correct order to build the project is: clean, prepare and compile)

You do not have to use Eclipse for checking out and deploying the Chilay Layout project. You can check out the project via an SVN client program like tortoise SVN [13]. After installing tortoise SVN, right click in windows explorer while `C:\` is open. Click "SVN Checkout". Enter `https://chilay.svn.sourceforge.net/svnroot/chilay/chilay2x` as URL of repository. Set "Checkout directory" as `C:\chilay20`, "Checkout Depth" as *Fully recursive* and "Revision" as *HEAD revision*, then click "OK". Open `build.properties` file under `C:\chilay20`, set `catalina.home` variable to `{TOMCAT_HOME}` and `catalina.build` variable to `{DEPLOYMENT_HOME}`.

Open command prompt, and enter:

```
cd C:\chilay20
ant
```

Launch the tomcat server by clicking `{TOMCAT_HOME}\bin\startup.bat`. Whenever you encounter an exception or unexpected error, first try deleting all files under `{TOMCAT_HOME}\work\catalina` and restart Tomcat service.

Using Chisio Layout as a Remote Layout Service

In order to use Chisio Layout as a remote layout service properly, you must save the geometry of your graph in an XML file which conforms to our XML schema. After sending (requesting) saved XML file via HTTP, you also have to read and parse the responded XML file which contains final positions of your graph.

Here we provide a simple example web application in ActionScript language [11] to implement a very simple graph renderer, with which you can load graphs stored in this XML format and remotely request layout. In this section, you will find sample code segments for sending and parsing input and output XML files from our simple graph layout tool. The complete listing of the sources can be found under the `doc/layout_server/ActionScript` directory in the project sources.

Assume that, your graph layout tool will visualize the graph stored in the following XML file:

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<view>
    <node id="ID_0" label="A">
        <bounds width="40.0" height="40.0" x="200.0" y="200.0"/>
    </node>
    <node id="ID_1" label="B">
        <bounds width="40.0" height="40.0" x="350.0" y="350.0"/>
    </node>
    <node id="ID_2" label="C">
        <bounds width="240.0" height="280.0" x="600.0" y="200.0"/>
        <children>
            <node id="ID_3" label="D">
                <bounds width="40.0" height="40.0" x="650.0" y="390.0"/>
            </node>
            <node id="ID_4" label="E">
                <bounds width="40.0" height="40.0" x="750.0" y="250.0"/>
            </node>
        </children>
    </node>
    <edge id="ID_0">
        <sourceNode id="ID_0"/>
        <targetNode id="ID_1"/>
    </edge>
    <edge id="ID_1">
        <sourceNode id="ID_1"/>
        <targetNode id="ID_2"/>
    </edge>
    <edge id="ID_2">
        <sourceNode id="ID_3"/>
        <targetNode id="ID_4"/>
    </edge>
    <edge id="ID_3">
        <sourceNode id="ID_0"/>
        <targetNode id="ID_4"/>
    </edge>
</view>
```

After successfully opening the XML file above and storing the content of it in an XML object (XML class exists in default package), we can scan all nodes and edges and construct the graph model in memory:

```
package default
{
    public class Graph
    {
        public var rootNodes: Array = new Array; // nodes at the root level
        public var edges: Array = new Array;
        public var allNodes: Array = new Array; // all nodes in the graph
        ...

        public function fromXML(xml: XML): void
        {
            for each (var node: XML in xml.node)
            {
                this.rootNodes.push(this.nodeFromXML(node));
            }

            for each (var edge: XML in xml..edge)
            {
                var fromId: String = edge.sourceNode.@id;
                var toId: String = edge.targetNode.@id;

                if ((fromId != null) && (toId != null))
                {
                    this.addEdge(edge.@id, fromId, toId);
                }
            }
        }

        public function toXML(): XML
        {
            var result:String = '<?xml version="1.0" encoding="UTF-8"
                standalone="yes"?><view>';

            for each (var n: Node in nodes)
            {
                result += n.asXML();
            }

            for each (var e: Edge in edges)
            {
                result += e.asXML()
            }

            result += "</view>"

            return XML(result);
        }

        private function nodeFromXML(node: XML): Node
        {
            var isCompound: Boolean = (node.children.length() > 0);
            var n: Node = isCompound ? new CompoundNode(node.@id): new Node(node.@id);

            n.x = node.bounds.@x;
            n.y = node.bounds.@y;
            n.width = node.bounds.@width;
            n.height = node.bounds.@height;

            if (isCompound)
            {
                var childNode: XML;
```

```
                for each (childNode in node.children.node)
                {
                    (n as CompoundNode).addNode(this.nodeFromXML(childNode));
                }
            }

            return n;
        }

        public function addEdge(edgeId: String, fromId: String, toId: String): Edge
        {
            var source: Node = this.nodeFromId(fromId);
            var target: Node = this.nodeFromId(toId);

            var e: Edge = new Edge(edgeId, source, target)

            this.edges.push(e);

            return e;
        }

        public function nodeFromId(id: String): Node
        {
            this.prepareAllNodes(); // this method constructs allNodes variable

            for each (var node: Node in allNodes)
            {
                if (node.id == id)
                {
                    return node;
                }
            }

            return null;
        }
    }
}

package default
{
    public class Node
    {
        public var id: String = null;
        public var x: Number = 0;
        public var y: Number = 0;
        public var w: Number;
        public var h: Number;
        public var clusterIDs: Array = new Array;
        protected var parent: CompoundNode = null;

        public function Node(id: String)
        {
            this.id = id;
        }

        public function isCompound(): Boolean { return false }

        public function toRectangle(): Rectangle
        {
            return new Rectangle(this.x, this.y, this.width, this.height);
```

```
        }

        public function asXML(): XML
        {
            var res: String = '<node id="' + this.id + '">' +
                '<bounds height="' + this.height +
                '" width="' + this.width +
                '" x="' + this.x +
                '" y="' + this.y +
                '" />' + '<clusterIDs>';

            for each (var c: uint in _clusterIDs)
            {
                res += '<clusterID>' + c + '</clusterID>';
            }

            res += '</clusterIDs></node>';

            return XML(res);
        }
    }
}

package default
{
    public class CompoundNode extends Node
    {
        private var nodes: Array = new Array;

        public function CompoundNode(id: String)
        {
            super(id);
        }

        public function addNode(n: Node): void
        {
            this.nodes.push(n);
            n.parent = this;
        }

        public function isCompound(): Boolean { return true }

        override public function asXML(): XML
        {
            var res: String = '<node id="' + this.id + '">' +
                '<bounds height="' + this.height +
                '" width="' + this.width +
                '" x="' + this.x +
                '" y="' + this.y +
                '" />' +
                '<children>';

            for each (var n: Node in _nodes)
            {
                res += n.asXML();
            }

            res += '</children></clusterIDs>';

            for each (var c: uint in _clusterIDs)
            {
```

```
                res += '<clusterID>' + c + '</clusterID>';
            }

            res += '</clusterIDs></node>';

            return XML(res);
        }
    }
}


package default
{
    public class Edge
    {
        private var id: String;
        private var source: Node;
        private var target: Node;

        public function Edge(id: String, source: Node, target: Node)
        {
            this.id = id;
            this.source = source;
            this.target = target;
        }

        public function asXML(): XML
        {
            return XML('<edge id="' + this.id + '">' +
                '<sourceNode id="' + this.source.id + '"/>' +
                '<targetNode id="' + this.target.id + '"/></edge>');
        }
    }
}
```

When `fromXML` method of a `Graph` object is called, graph model specified in an XML file (like above) will be stored in the relevant `Graph` object. Once the topology intact in the memory, one can request remote layout on this graph model. Let us assume Compound Spring Embedder (CoSE) Layout style is to be used:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
        initialize="init()">

    <mx:Canvas id="drawingCanvas" width="1920" height="1080" x="0" y="0">
    </mx:Canvas>

    <mx:Script>
        <![CDATA[

            import ru.inspirit.net.MultipartURLLoader;

            private var loader: MultipartURLLoader = new MultipartURLLoader;
            private var graph: Graph = new Graph;
            ...

            public function init(): void
            {
                // When layout request successfully responed, onLayoutComplete method will
                // be called
                loader.addEventListener(Event.COMPLETE, onLayoutComplete);
```

```
            }

            // assume that, requestLayout function is called
            // via your interactive graph editor tool
            // after a graph model in XML format is loaded
            private function requestLayout(): void
            {
                var ba: ByteArray = new ByteArray;
                ba.writeUTFBytes(graph.toXML().toXMLString());
                loader.addFile(ba, "graph");

                // append general options
                loader.addVariable("layoutQuality", 1)
                loader.addVariable("animateOnLayout", false)
                loader.addVariable("incremental", false)

                // append CoSE options
                loader.addVariable("springStrength", 50)
                loader.addVariable("repulsionStrength", 50)
                loader.addVariable("gravityStrength", 50)
                loader.addVariable("compoundGravityStrength", 50)
                loader.addVariable("idealEdgeLength", 40)
                loader.addVariable("layoutStyle", "cose")

                // You should load the XML file to the dedicated server that you have
                // deployed Chisio Layout project
                loader.load("http://ip_number/deployment_directory/layout.jsp");
            }

            private function onLayoutComplete(e: Event): void
            {
                var response: XML = XML(loader.getResponse());
                graph.fromXML(response);
                drawGraph();
            }

        ]]>
    <mx:Script>
</mx:Application>
```

The `MultipartURLLoader` class can also be located under the sample project sources. Now we can visualize our graph as follows:

```
<?xml version="1.0" encoding="utf-8"?>
<mx:Application xmlns:mx="http://www.adobe.com/2006/mxml"
        initialize="init()">

    <mx:Canvas id="drawingCanvas" width="1920" height="1080" x="0" y="0">
    </mx:Canvas>

    <mx:Script>
        <![CDATA[

            ... // See the methods below

            private function drawGraph(): void
            {
                for each (var node: Node in graph.nodes)
                {
                    drawNode(node);
```

```
        }

        for each (var edge: Edge in graph.edges)
        {
            drawEdge(edge);
        }
    }

    private function drawNode(node: Node): void
    {
        if (node.isCompound())
        {
            drawCompound((node as CompoundNode));
        }
        else
        {
            drawingCanvas.graphics.beginFill(0xAA0000, 75);
            drawingCanvas.graphics.lineStyle(3, 0x000000);
            drawingCanvas.graphics.drawRoundRect(node.x,
                node.y,
                node.width,
                node.height,
                10);
            drawingCanvas.graphics.endFill();
        }
    }

    private function drawEdge(edge: Edge): void
    {
        var clippingPoints: Array = new Array();

        /* getIntersection method calculates the intersection (clipping) points of
         * the two input rectangles with line segment defined by the centers of
         * these two rectangles. The clipping points are saved in the input double
         * array and whether or not the two rectangles overlap is returned.
         */
        Utils.getIntersection(edge.source.toRectangle(),
            edge.target.toRectangle(),
            clippingPoints);

        drawingCanvas.graphics.beginFill(0x000000, 100);
        drawingCanvas.graphics.lineStyle(2, 0x000000);
        drawingCanvas.graphics.moveTo(clippingPoints[0], clippingPoints[1]);
        drawingCanvas.graphics.lineTo(clippingPoints[2], clippingPoints[3]);
        drawingCanvas.graphics.endFill();
    }

    private function drawCompound(compound: CompoundNode): void
    {
        drawingCanvas.graphics.beginFill(0x00AA00, 10);
        drawingCanvas.graphics.lineStyle(2, 0x000000);
        drawingCanvas.graphics.drawRect(compound.x,
            compound.y,
            compound.width,
            compound.height);
        drawingCanvas.graphics.endFill();

        var childNode: Node;

        for each (childNode in compound.nodes)
        {
```

```
                    drawNode(childNode);
                }
            }
        ]]>
    <mx:Script>
</mx:Application>
```

Layout information calculated for the graph described by the XML file in Figure 4 is integrated into the XML file (Figure 7). Figure 6 shows a drawing of the layout calculated.

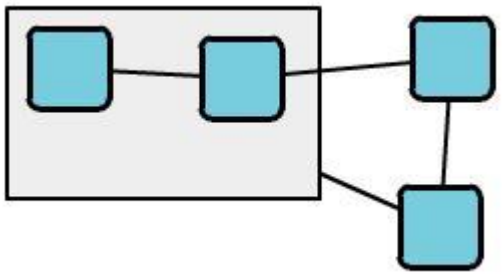

Figure 6 Final Layout

```
<?xml version="1.0" encoding="UTF-8" standalone="yes"?>
<view>
  <node id="ID_0" clusterID="0">
    <bounds height="40" width="40" x="212" y="15"/>
  </node>
  <node id="ID_1" clusterID="0">
    <bounds height="40" width="40" x="206" y="99"/>
  </node>
  <node id="ID_2" clusterID="0">
    <bounds height="95" width="156" x="10" y="10"/>
    <children>
      <node id="ID_3" clusterID="0">
        <bounds height="40" width="40" x="21" y="20"/>
      </node>
      <node id="ID_4" clusterID="0">
        <bounds height="40" width="40" x="107" y="25"/>
      </node>
    </children>
  </node>
  <edge id="ID_0">
    <sourceNode id="ID_0"/>
    <targetNode id="ID_1"/>
  </edge>
  <edge id="ID_1">
    <sourceNode id="ID_1"/>
    <targetNode id="ID_2"/>
  </edge>
  <edge id="ID_2">
    <sourceNode id="ID_3"/>
    <targetNode id="ID_4"/>
  </edge>
  <edge id="ID_3">
    <sourceNode id="ID_0"/>
    <targetNode id="ID_4"/>
  </edge>
</view>
```

Figure 7 A sample final layout responded from Chisio Layout Service

Chapter 4

Third-Party Software License Agreements

In this chapter, we list any third-party software license agreements.

Chisio Layout is distributed under Eclipse Public License, version 1.0. Notwithstanding the terms and conditions of this license and any agreement you have with i-Vis Research Group of Bilkent University, the third-party software code, products, or files identified below are "Excluded Components" and are subject to the following terms and conditions.

Component	Version	By	License
JAXB	-	Oracle	Common Development and Distribution License, Version 1.1
JUnit	4.6	Object Mentor	Common Public License - v 1.0
Commons	1.x	The Apache Software Foundation	Apache License Version 2.0

References

[1] T.M.J. Fruchterman, E.M. Reingold, Graph drawing by force-directed placement, Software Practice and Experience 21 (11) (1991) 1129–1164.

[2] U. Dogrusoz, E. Giral, A. Cetintas, A. Civril, and E. Demir, "A Layout Algorithm For Undirected Compound Graphs", Information Sciences, 179, pp. 980–994, 2009.

[3] C. Walshaw. *A Multilevel Algorithm for Force-directed Graph Drawing*. (Proc. Graph Drawing) LNCS, 1984:171-182, 2001.

[4] U. Dogrusoz, M.E. Belviranli, and A. Dilek, "CiSE: A Circular Spring Embedder Layout Algorithm", under review.

[5] H. He and O. Sýkora, "New circular drawing algorithms", In Proc. Workshop on Information Technologies - Applications and Theory (ITAT'04), 2004.

[6] K. Sugiyama, S. Tagawa, and M. Toda: "Methods for visual understanding of hierarchical system structures", IEEE Transactions on Systems, Man, and Cybernetics SMC-11(2): 109-125, February 1981.

[7] http://eclipse.org/downloads/

[8] http://subclipse.tigris.org/

[9] http://ant.apache.org/bindownload.cgi

[10] http://tomcat.apache.org/download-55.cgi

[11] http://www.actionscript.org/

[12] http://csbi.sourceforge.net/

[13] http://tortoisesvn.net/downloads.html