



IBM ILOG Views
Graph Layout V5.3
User's Manual

June 2009

© **Copyright International Business Machines Corporation 1987, 2009.**

US Government Users Restricted Rights – Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Copyright notice

© Copyright International Business Machines Corporation 1987, 2009.

US Government Users Restricted Rights - Use, duplication or disclosure restricted by GSA ADP Schedule Contract with IBM Corp.

Trademarks

IBM, the IBM logo, ibm.com, Websphere, ILOG, the ILOG design, and CPLEX are trademarks or registered trademarks of International Business Machines Corp., registered in many jurisdictions worldwide. Other product and service names might be trademarks of IBM or other companies. A current list of IBM trademarks is available on the Web at "Copyright and trademark information" at <http://www.ibm.com/legal/copytrade.shtml>

Adobe, the Adobe logo, PostScript, and the PostScript logo are either registered trademarks or trademarks of Adobe Systems Incorporated in the United States, and/or other countries.

Linux is a registered trademark of Linus Torvalds in the United States, other countries, or both.

Microsoft, Windows, Windows NT, and the Windows logo are trademarks of Microsoft Corporation in the United States, other countries, or both.

Java and all Java-based trademarks and logos are trademarks of Sun Microsystems, Inc. in the United States, other countries, or both.

Other company, product, or service names may be trademarks or service marks of others.

Notices

For further information see *<installdir>/license/notices.txt* in the installed product.

Table of Contents

Preface	About This Manual	9
	What You Need to Know	9
	Manual Organization	10
	Notation	10
	Typographic Conventions	10
	Naming Conventions	10
	A Note on Examples	11
	Related Documentation	11
	Books	11
	Bibliographies	12
	Journals	12
	Conferences	13
Chapter 1	Introducing the Graph Layout Package	15
	What is the Graph Layout Package of IBM ILOG Views?	15
	Features of IBM ILOG Views Graph Layout	17
	IBM ILOG Views Graph Layout in User Interface Applications	21
Chapter 2	Basic Concepts	25
	Graph Layout: A Brief Introduction	25

	What is a Good Layout?	26
	Methods for Using Layout Algorithms	27
	Graph Layout in IBM ILOG Views	27
	The Base Class: IlvGraphLayout	28
	Basic Operations with IlvGraphLayout	29
	Instantiating a Subclass of IlvGraphLayout	29
	Attaching a Grapher	29
	Performing a Layout	30
	Detaching a Grapher	31
	Layout Parameters in IlvGraphLayout	31
	Allowed Time	31
	Animation	32
	Layout Region	33
	Preserve Fixed Links	34
	Preserve Fixed Nodes	35
	Random Generator Seed Value	35
	Use Default Parameters	37
Chapter 3	Getting Started with Graph Layout	39
	Basic Steps for Using Layout Algorithms: A Summary	39
	Sample Application	40
	Launching IBM ILOG Views Studio with the Graph Layout Extension	41
	A Quick Look at the Interface	42
Chapter 4	Layout Algorithms	45
	Determining the Appropriate Layout Algorithm	45
	Generic Parameters Support	48
	Layout Characteristics	49
	Tree Layout	49
	Samples	50
	What Types of Graphs?	51
	Application Domains	51

Features	52
Limitations	52
Brief Description of the Algorithm	53
Code Sample	55
Parameters	55
Generic Parameters	56
Specific Parameters (All Tree Layout Modes)	56
Free Layout Mode	59
Level Layout Mode	70
Radial Layout Mode	72
Tip-Over Layout Modes	77
For Experts: Further Tips and Tricks	79
Hierarchical Layout	84
Samples	84
What Types of Graphs?	86
Application Domains	86
Features	86
Limitations	87
Brief Description of the Algorithm	87
Code Sample	89
Parameters	89
Generic Parameters	89
Specific Parameters	90
Sequences of Graph Layout	102
Orthogonal Link Layout	106
Samples	106
What Types of Graphs?	108
Application Domains	108
Features	108
Limitations	109
Brief Description of the Algorithm	109

Code Sample	110
Parameters	110
Generic Parameters	110
Specific Parameters	111
Random Layout	115
Sample	115
What Types of Graphs?	115
Features	116
Limitations	116
Brief Description of the Algorithm	116
Code Sample	116
Parameters	117
Generic Parameters	117
Specific Parameters	118
Bus Layout	119
Sample	119
What Types of Graphs?	119
Application Domains	119
Features	119
Brief Description of the Algorithm	120
Code Sample	120
Parameters	121
Generic Parameters	121
Specific Parameters	122

Chapter 5	Using Advanced Features	133
	Using a Layout Report	133
	Layout Report Classes	134
	Creating a Layout Report	134
	Reading a Layout Report	135
	Information Stored in a Layout Report	135
	Using Layout Event Listeners	135

Using the Graph Model	136
The Graph Model Concept	137
The IlvGraphModel Class	138
Using the IlvGrapherAdapter	141
Laying Out a Non-Views Grapher	141
Using the Filtering Features to Lay Out a Part of an IlvGrapher	142
Filtering by Layers	143
Filtering by Graphic Objects	144
Laying Out Graphs with Nonzoomable Graphic Objects	144
A Special Case: Nonzoomable Graphic Objects	145
Reference Transformers	145
How a Reference Transformer is Used	146
Reference Views	146
Specifying a Reference Transformer	147
Defining a New Type of Layout	148
Questions and Answers about Using the Layout Algorithms	152
Index	163

About This Manual

The IBM® ILOG® Views Component Suite provides special support for applications that need to display graphs (networks) of nodes and links. Any graphic object can be defined to behave like a node and be connected to other nodes via links, which themselves can have many different forms. The Graph Layout package provides high-level, ready-to-use graph drawing services that allow you to obtain readable representations easily.

This *User's Manual* explains how to use the C++ API and grammar that are detailed in the IBM ILOG Views *Graph Layout Reference Manual*.

What You Need to Know

This manual assumes that you are familiar with the PC or UNIX® environment in which you are going to use IBM® ILOG® Views, including its particular windowing system. Since IBM ILOG Views is written for C++ developers, the documentation also assumes that you can write C++ code and that you are familiar with your C++ development environment so as to manipulate files and directories, use a text editor, and compile and run C++ programs.

Manual Organization

This manual provides information for developing applications that incorporate the IBM ILOG Views Graph Layout package. It describes the fundamentals that underlie the graph algorithms and shows how to create customized graph layouts.

This manual contains the following chapters:

- ◆ Chapter 1, *Introducing the Graph Layout Package* describes the Graph Layout package of IBM ILOG Views and its features.
- ◆ Chapter 2, *Basic Concepts* provides background information and basic concepts for using Graph Layout.
- ◆ Chapter 3, *Getting Started with Graph Layout* provides information to get started quickly using Graph Layout.
- ◆ Chapter 4, *Layout Algorithms* describes the layout algorithms provided with the Graph Layout package.
- ◆ Chapter 5, *Using Advanced Features* provides information on using a layout report, using layout event listeners, using a graph model, laying out a non-Views grapher, laying out a portion of a graph, laying out graphs with nonzoomable objects, and defining new types of layouts.

Notation

Typographic Conventions

The following typographic conventions apply throughout this manual:

- ◆ Code extracts and file names are written in a "code" typeface.
- ◆ Entries to be made by the user, such as in dialog boxes, are written in a "code" typeface.
- ◆ Command variables to be supplied by the user are written in *italics*.
- ◆ Some words in *italics*, when seen for the first time, may be found in the glossary.

Naming Conventions

Throughout the documentation, the following naming conventions apply to the API.

- ◆ The names of types, classes, functions, and macros defined in the IBM ILOG Views Foundation library begin with `Ilv`, for example `IlvGraphic`.

- ◆ The names of types and macros not specific to IBM ILOG Views begin with `Il`, for example `IlBoolean`.
- ◆ The names of classes as well as global functions are written as concatenated words with each initial letter capitalized.

```
class IlvDrawingView;
```

- ◆ The names of virtual and regular methods begin with a lowercase letter; the names of static methods start with an uppercase letter. For example:

```
virtual IlvClassInfo* getClassInfo () const;  
static IlvClassInfo* ClassInfo* () const;
```

A Note on Examples

The documentation offers examples and explanations of how to use IBM ILOG Views effectively. Moreover, some examples are extracted from the source code delivered with IBM ILOG Views, which is in the `samples` directory, just below the directory where IBM ILOG Views is installed.

Related Documentation

The following documentation may provide helpful information when using IBM ILOG Views Graph Layout.

Books

The first book dedicated to graph layout has been published:

Di Battista, Giuseppe, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. *Graph Drawing: Algorithms for the Visualization of Graphs*. Prentice Hall, 1999 (see <http://www.cs.brown.edu/people/rt/gdbook.html> or http://www.prenhall.com/books/esm_0133016153.html).

Graph layout is closely related to graph theory, for which extensive literature exists. See:

Clark, John and Derek Allan Holton. *A First Look at Graph Theory*. World Scientific Publishing Company, 1991.

For a mathematics-oriented introduction to graph theory, see:

Diestel, Reinhard. *Graph Theory*. 2nd ed. Springer-Verlag, 2000.

A more algorithmic approach may be found in:

Gibbons, Alan. *Algorithmic Graph Theory*. Cambridge University Press, 1985.

Gondran, Michel and Michel Minoux. *Graphes et algorithmes*, 3rd ed. Eyrolles, Paris, 1995 (in French).

Bibliographies

A comprehensive bibliographic database of papers in computational geometry (including graph layout) can be found in:

The Geometry Literature Database

(<http://compgeom.cs.uiuc.edu/~jeffe/compgeom/biblios.html>)

A recommended bibliographic survey paper is the following:

Di Battista, Giuseppe, Peter Eades, Roberto Tamassia, and Ioannis G. Tollis. "Algorithms for Drawing Graphs: an Annotated Bibliography." *Computational Geometry: Theory and Applications* 4 (1994): 235-282 (also available at <http://www.cs.brown.edu/people/rt/gd-biblio.html>).

Journals

The following is an electronic journal:

Journal of Graph Algorithms and Applications

(<http://www.cs.brown.edu/publications/jgaa>)

Special issues of the following journals are dedicated to graph drawing:

Algorithmica

(<http://link.springer-ny.com/link/service/journals/00453/>)

Computational Geometry: Theory and Applications

(<http://www.elsevier.nl/inca/publications/store/5/0/5/6/2/9/>)

Journal of Visual Languages and Computing

(<http://www.academicpress.com/jvlc>)

The following journals occasionally publish papers on graph layout:

Information Processing Letters

(<http://www.elsevier.nl/inca/publications/store/5/0/5/6/1/2/>)

Computer-aided Design

(<http://www.elsevier.nl/inca/publications/store/3/0/4/0/2/>)

IEEE Transactions on Software Engineering

(<http://www.computer.org/tse/>)

Many papers are presented at many conferences in Combinatorics and Computer Science.

Conferences

An annual Symposium on Graph Drawing has been held since 1992. The proceedings are published by Springer-Verlag in the *Lecture Notes in Computer Science* series (for 1999 see, for example, <http://link.springer.de/link/service/series/0558/tocs/t1731.htm>). The 2000 Symposium is being held in Colonial Williamsburg, Virginia (see <http://www.cs.virginia.edu/~gd2000/>).

Introducing the Graph Layout Package

This chapter introduces you to the Graph Layout package of IBM® ILOG® Views. The following topics are covered:

- ◆ *What is the Graph Layout Package of IBM ILOG Views?*
- ◆ *Features of IBM ILOG Views Graph Layout*
- ◆ *IBM ILOG Views Graph Layout in User Interface Applications*

What is the Graph Layout Package of IBM ILOG Views?

Many types of complex business data can be best visualized as a set of nodes and interconnecting links, more commonly called a graph or a network. Examples of graphs include business organizational charts, work flow diagrams, telecom network displays, and genealogical trees. Whenever these graphs become large or heavily interconnected, it becomes difficult to see the relationships between the various nodes and links (also called “edges”). This is where IBM ILOG Views Graph Layout algorithms help.

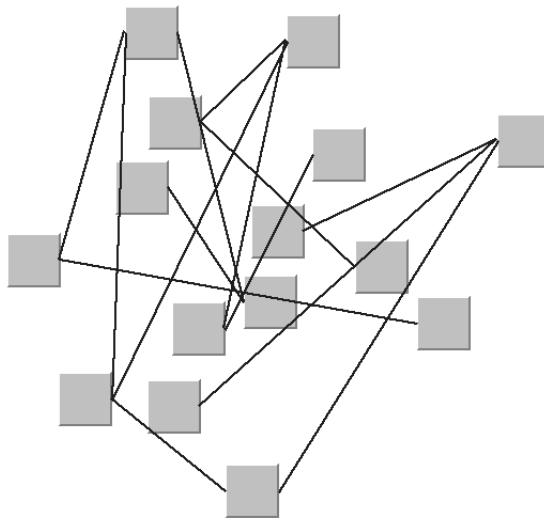
IBM ILOG Views provides special support for applications that need to display graphs. Any graphic object can be defined to behave like a node and can be connected to other nodes via links, which themselves can have many different forms.

The Graph Layout package provides high-level, ready-to-use relationship visualization services. It allows you to take any “messy” graph and apply a sophisticated graph layout algorithm to rearrange the positions of the nodes and links. The result is a more readable and understandable picture.

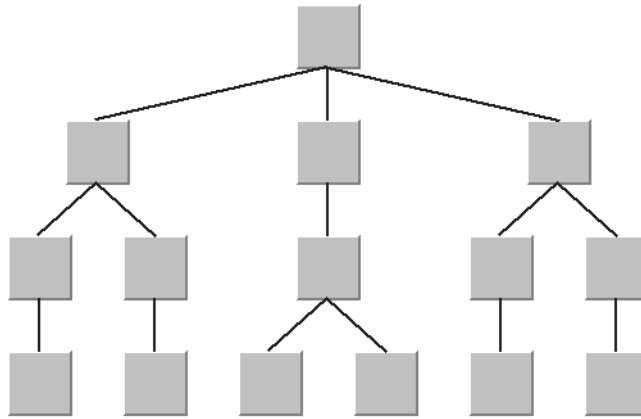
The Graph Layout package is composed of the following modules:

- ◆ `ilvlayout` A high-level, generic framework for the graph layout services provided by IBM ILOG Views.
- ◆ `ilvbus` A layout algorithm designed to display bus network topologies (that is, a set of nodes connected to a bus node).
- ◆ `ilvhierarchical` A layout algorithm that arranges nodes in horizontal or vertical levels such that the links flow in a uniform direction.
- ◆ `ilvorthlink` A layout algorithm that reshapes the links of a graph without moving the nodes.
- ◆ `ilvrandom` A layout algorithm that moves the nodes of the graph at randomly computed positions inside a user-defined region.
- ◆ `ilvtree` A layout algorithm that arranges the nodes of a tree horizontally, vertically, or in circular layers, starting from the root of the tree.

Before getting started with the Graph Layout package, take a look at two sample drawings of the same graph. For the first one, the nodes were placed randomly when the graph was drawn.



Using the Tree Layout algorithms provided in IBM ILOG Views, the following drawing was obtained:



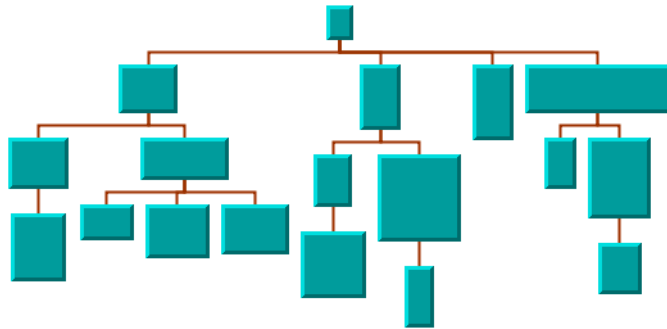
In the second drawing, the layout algorithm has distributed the nodes with the tree structure exposed, avoiding overlapping nodes and showing the symmetries of the graph. This drawing presents a much more readable layout than the first drawing does.

Features of IBM ILOG Views Graph Layout

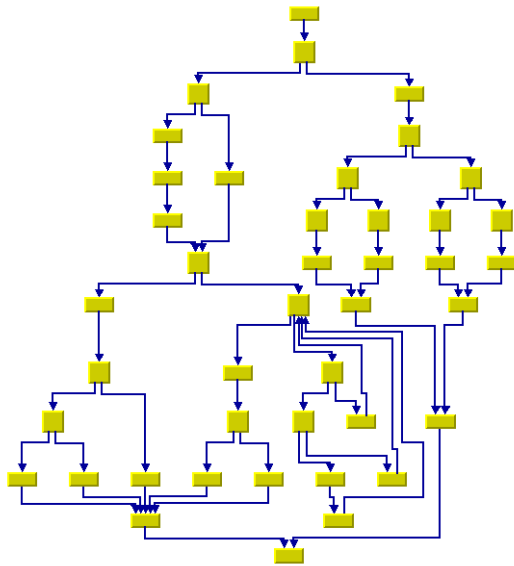
The Graph Layout package of IBM ILOG Views provides several ready-to-use layout algorithms. In addition, new layout algorithms can be developed using the generic layout framework of the Graph Layout package.

The Graph Layout package provides the following ready-to-use layout algorithms:

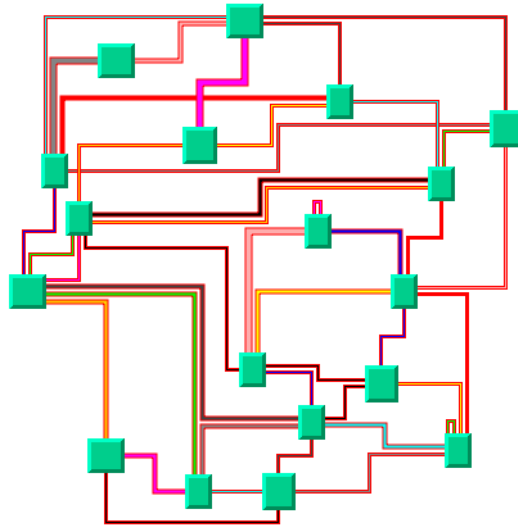
- ◆ Tree Layout



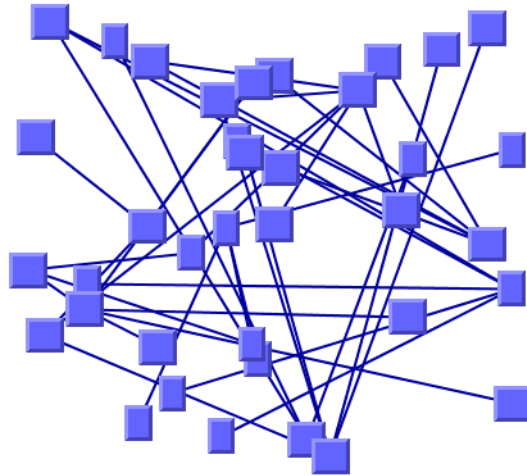
◆ Hierarchical Layout



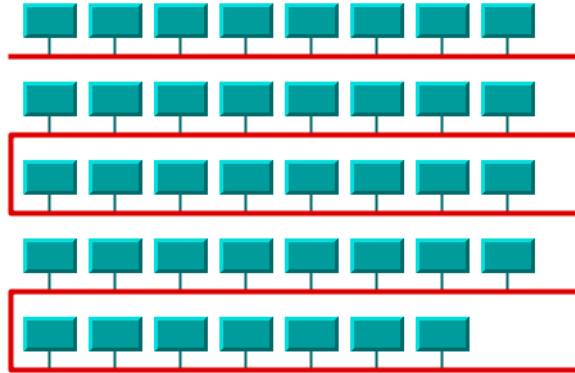
◆ Orthogonal Link Layout



◆ Random Layout



◆ Bus Layout



The Graph Layout package provides the following features for using the layout algorithms. (Note that some of these features are not supported by all the algorithms provided with IBM ILOG Views.)

- ◆ Capability to fit the layout into a manager view or a user-defined rectangle
- ◆ Capability to take into account the size of the nodes when performing the layout to avoid overlapping nodes
- ◆ Capability to perform the layout using only the nodes and links that are on user-defined layers of the graph
- ◆ Capability to perform the layout only on those parts of the graph that meet user-defined conditions
- ◆ Capability to use non-Views graphers
- ◆ Layout reports providing information concerning the behavior of the layout algorithm
- ◆ Layout-event listeners that can receive and report information during the graph layout
- ◆ A generic framework for customizing the layout algorithms. The following generic parameters are defined. (Note that not all the layout algorithms provided with IBM ILOG Views support all these parameters. Whether a generic parameter is supported depends on the particular layout algorithm.)
 - Allowed Time

This parameter allows the layout algorithm to stop computation when a user-defined time specification is exceeded.
 - Animation

This parameter allows the layout algorithm to redraw the graph after each iteration or step.

- **Preserve Fixed Nodes**

This parameter allows the layout algorithm to preserve the location of selected nodes. Certain nodes can be specified as fixed and will not be moved when the layout is performed. The layout algorithm can “pin” specified nodes in place.
- **Preserve Fixed Links**

This parameter allows the layout algorithm to preserve the shape of selected links. Certain links can be specified as fixed and will not be reshaped when the layout is performed. The layout algorithm can “pin” specified links in place.
- **Filtering**

The layout algorithms are able to perform the layout using only the nodes and links that are on user-defined layers of the grapher (see *Filtering by Layers*) or to exclude nodes and links on an individual basis (see *Filtering by Graphic Objects*).
- **Layout Region**

This parameter allows the layout algorithm to control the size of the graph drawing.
- **Random Generator Seed Value**

This parameter allows the layout algorithm to use randomly-generated numbers that can be initialized with a user-defined seed value. These seed values are then used during layout computations to produce different layouts of the graph.
- **Use Default Parameters**

This parameter allows the layout algorithm to return to using default parameter settings after the default settings have been modified.

IBM ILOG Views Graph Layout in User Interface Applications

Many fields use graph drawings and graph layouts in user interface applications. Therefore, the IBM ILOG Views Graph Layout package is particularly well-suited for these kinds of applications. The following is a list of some of the fields where the graph layout capabilities of the Graph Layout package can be used:

- ◆ **Telecom and Networking**
 - LAN Diagrams (*Bus Layout*)
- ◆ **Electrical Engineering**
 - Logic Diagrams (*Hierarchical Layout*)
 - Circuit Block Diagrams (*Hierarchical Layout, Orthogonal Link Layout, Bus Layout*)
- ◆ **Industrial Engineering**

- Industrial Process Charts (*Hierarchical Layout*)
- Schematic Design Diagrams (*Orthogonal Link Layout, Hierarchical Layout*)
- Equipment/Resource Control Charts (*Bus Layout, Orthogonal Link Layout*)
- ◆ **Business Processing**
 - Workflow Diagrams (*Hierarchical Layout*)
 - Process Flow Diagrams (*Hierarchical Layout*)
 - Organization Charts (*Tree Layout*)
 - Entity Relation Diagrams (*Orthogonal Link Layout*)
 - PERT Charts (*Hierarchical Layout*)
- ◆ **Software Management/Software (Re-)Engineering**
 - UML Diagrams (*Hierarchical Layout, Tree Layout*)
 - Flow Charts (*Hierarchical Layout*)
 - Data Inspector Diagrams (*Orthogonal Link Layout, Hierarchical Layout*)
 - Call Graphs (*Hierarchical Layout, Tree Layout*)
- ◆ **CASE Tools**
 - Design Diagrams (*Orthogonal Link Layout, Hierarchical Layout*)
- ◆ **Data Base and Knowledge Engineering**
 - Decision Trees (*Tree Layout*)
 - Database Query Graphs (*Hierarchical Layout*)
- ◆ **The World Wide Web**
 - Web Site Maps (*Tree Layout*)

Basic Concepts

In this chapter, you will learn about some basic concepts and background information that will help you when using the IBM® ILOG® Views Graph Layout package. The following topics are covered:

- ◆ *Graph Layout: A Brief Introduction*
- ◆ *Graph Layout in IBM ILOG Views*
- ◆ *The Base Class: IlvGraphLayout*
- ◆ *Basic Operations with IlvGraphLayout*
- ◆ *Layout Parameters in IlvGraphLayout*

Graph Layout: A Brief Introduction

This section provides some background information about graph layout in general, not specifically related to the IBM ILOG Views Graph Layout package.

Simply speaking, a graph is a data structure which represents a set of entities, called nodes, connected by a set of links. (A node can also be referred to as a vertex. A link can also be referred to as an edge or a connection.) In practical applications, graphs are frequently used to model a very wide range of things: computer networks, software program structures, project management diagrams, and so on. Graphs are powerful models because they permit

applications to benefit from the results of graph theory research. For instance, efficient methods are available for finding the shortest path between two nodes, the minimum cost path, and so on.

Graph layout is used in graphical user interfaces of applications that need to display graph models. To lay out a graph means to draw the graph so that an appropriate, readable representation is produced. Essentially, this involves determining the location of the nodes and the shape of the links. For some applications, the location of the nodes may be already known (based on the geographical positions of the nodes, for example). However, for other applications, the location is not known (a pure “logical” graph) or the known location, if used, would produce an unreadable drawing of the graph. In these cases, the location of the nodes must be computed.

But what is meant by an “appropriate” drawing of a graph? In practical applications, it is often necessary for the graph drawing to respect certain quality criteria. These criteria may vary depending on the application field or on a given standard of representation. It is often difficult to speak about what a good layout consists of. Each end user may have different, subjective criteria for qualifying a layout as “good”. However, one common goal exists behind all the criteria and standards: the drawing must be easy to understand and provide easy navigation through the complex structure of the graph.

What is a Good Layout?

To deal with the various needs of different applications, many classes of graph layout algorithms have been developed. A layout algorithm addresses one or more quality criteria, depending on the type of graph and the features of the algorithm, when laying out a graph. The most common criteria are:

- ◆ Minimizing the number of link crossings
- ◆ Minimizing the total **area** of the drawing
- ◆ Minimizing the number of **bends** (in orthogonal drawings)
- ◆ Maximizing the smallest **angle** formed by consecutive incident links
- ◆ Maximizing the display of **symmetries**

How can a layout algorithm meet each of these quality criteria and standards of representation? If you look at each individual criteria, some can be met quite easily, at least for some classes of graphs. For other classes, it may be quite difficult to produce a drawing that meets the criteria. For example, minimizing the number of link crossings is relatively simple for trees (that is, graphs without cycles). However, for general graphs, minimizing the number of link crossings is a mathematical NP-complete problem (that is, with all known algorithms, the time required to perform the layout grows very fast with the size of the graph.)

Moreover, if you want to meet several criteria at the same time, an optimal solution simply may not exist with respect to each individual criteria because many of the criteria are mutually contradictory. Time-consuming trade-offs may be necessary. In addition, it is not a trivial task to assign weights to each criteria. Multicriteria optimization is, in most cases, too complex to implement and much too time-consuming. For these reasons, layout algorithms are often based on heuristics and may provide less than optimal solutions with respect to one or more of the criteria. Fortunately, in practical terms, the layout algorithms will still often provide reasonably readable drawings.

Methods for Using Layout Algorithms

Layout algorithms can be employed in a variety of ways in the various applications in which they are used. The most common ways of using an algorithm are the following:

◆ **Automatic layout**

The layout algorithm does everything without any user intervention, except perhaps the choice of the layout algorithm to be used. Sometimes a set of rules can be coded to choose automatically (and dynamically) the most appropriate layout algorithm for the particular type of graph being laid out.

◆ **Semi-automatic layout**

The end user is free to improve the result of the automatic layout procedure by hand. At times the end user can move and “pin” nodes at a desired location and perform the layout again. In other cases, a part of the graph is automatically set as “read-only” and the end user can modify the rest of the layout.

◆ **Static layout**

The layout algorithm is completely redone (“from scratch”) each time the graph is changed.

◆ **Incremental layout**

When the layout algorithm is performed a second time on a modified graph, it tries to preserve the stability of the layout as much as possible. The layout is not performed again from scratch. The layout algorithm also tries to economize CPU time by using the previous layout as an initial solution. Some layout algorithms and layout styles are incremental by nature. For others, incremental layout may be impossible.

Graph Layout in IBM ILOG Views

In IBM ILOG Views, graphs are instances of the class `IlvGrapher`. We call these instances *graphers*. The nodes, which are instances of `IlvGraphic`, and the links, which are instances of `IlvLinkImage`, “know” how to draw themselves. The nodes can have arbitrary

coordinates, or they can be placed interactively or by code. All that needs to be done to lay out the grapher to obtain a readable drawing is to compute and assign appropriate coordinates for the nodes. In some cases, the shape of the links may also need to be modified. The main task of the Graph Layout package is to provide support for the operation of laying out a grapher—that is, drawing the graph.

The Graph Layout package of IBM ILOG Views benefits from its integration with the graph visualization and graph manipulation features of the IBM ILOG Views 2D Graphics Professional product. This core library provides a wide range of very useful features to build powerful graphic interfaces easily:

- ◆ Predefined, extensible types of graphic objects for nodes and links
- ◆ A customizable mechanism to choose the contact points between links and nodes
- ◆ Grapher interactor classes
- ◆ Multiple views of the same grapher
- ◆ Management of multiple layers
- ◆ Selections management
- ◆ Events management
- ◆ Listeners on the contents of the grapher and/or on the views
- ◆ Printing facilities
- ◆ User properties on nodes and links
- ◆ Input/output operations

For details on these features, see the IBM ILOG Views *Foundation* and *2D Graphics Professional User's Manuals*.

Note: The Graph Layout package allows you to add layout capabilities to applications that do not use the IBM ILOG Views grapher. For details, see *Laying Out a Non-Views Grapher*.

The Base Class: `IlvGraphLayout`

The `IlvGraphLayout` class is the base class for all layout algorithms. This class is an abstract class and cannot be used directly. You must use one of its subclasses (`IlvTreeLayout`, `IlvHierarchicalLayout`, `IlvOrthogonalLinkLayout`, `IlvRandomLayout`, or `IlvBusLayout`). You can also create your own subclasses to implement other layout algorithms (see *Defining a New Type of Layout*).

Despite the fact that only subclasses of `IlvGraphLayout` are directly used to obtain the layouts, it is still necessary to learn about this class because it contains methods that are inherited (or overridden) by the subclasses. And, of course, you will need to understand it if you subclass it yourself.

You can find more information about the class `IlvGraphLayout` in the sections:

- ◆ *Basic Operations with IlvGraphLayout* tells you about the basic methods you need using the subclasses of `IlvGraphLayout`.
- ◆ *Layout Parameters in IlvGraphLayout* contains the methods that are related to the customization of the layout algorithms.
- ◆ *Using Layout Event Listeners* is an advanced feature that tells you about the layout event listener mechanism.
- ◆ *Defining a New Type of Layout* is an advanced feature that tells you how to implement new subclasses.

Basic Operations with IlvGraphLayout

When subclassing `IlvGraphLayout`, you will normally use the basic methods described in this section.

Instantiating a Subclass of IlvGraphLayout

The class `IlvGraphLayout` is an abstract class. You will instantiate a subclass as shown in the following example:

```
IlvOrthogonalLinkLayout* layout = new IlvOrthogonalLinkLayout();
```

If you want to use the layout report that is returned by the layout algorithm, you need to declare a handle for the appropriate layout report class, as in this example:

```
IlvGraphLayoutReport* layoutReport;
```

For more information on the layout report, see *Using a Layout Report*.

Attaching a Grapher

The `IlvGraphLayout::attach` method of the `IlvGraphLayout` class allows you to specify the grapher you want to lay out:

```
void attach(IlvGrapher* grapher)
```

For example:

```
IlvGrapher* grapher = new IlvGrapher(display);  
// Add nodes and links to the grapher here  
layout->attach(grapher);
```

You must attach the grapher before performing the layout. The method `IlvGraphLayout::attach` first detaches the grapher that is already attached, if any. You can obtain the attached grapher using the method `IlvGraphLayout::getGrapher`.

Performing a Layout

The `IlvGraphLayout::performLayout` method starts the layout algorithm using the currently attached grapher and the current settings for the layout parameters (see *Layout Parameters in IlvGraphLayout*).

```
IlvGraphLayoutReport* performLayout()
```

The layout algorithm first verifies whether it is necessary to perform the layout. It checks internal flags to see whether the grapher or any of the parameters have been changed since the last time the layout was successfully performed. A “change” can be any of the following:

- ◆ Nodes or links were added or removed.
- ◆ Nodes or links were moved or reshaped.
- ◆ The value of a layout parameter was modified.
- ◆ The size of a manager view into which the layout must fit changed. (The layout region mechanism is explained in *Layout Region*.)

Users often do not want the layout to be computed again if no changes occurred. If there were no changes, the method `IlvGraphLayout::performLayout` returns without performing the layout.

The protected abstract method `IlvGraphLayout::layout` is then called. This means that the control is passed to the subclasses that are implementing this method. The implementation computes the layout and moves the nodes to new positions.

The `IlvGraphLayout::performLayout` method returns an instance of `IlvGraphLayoutReport` (or of a subclass) that contains information about the behavior of the layout algorithm. It tells you whether the algorithm performed normally, or whether a particular, predefined case occurred. (For a more detailed description of the layout report, see *Using a Layout Report*.)

Note that the layout report that is returned can be an instance of a subclass of `IlvGraphLayoutReport` depending on the particular subclass of `IlvGraphLayout` you are using. Subclasses of `IlvGraphLayoutReport` are used to store layout algorithm-dependent information.

Detaching a Grapher

You call the `IlvGraphLayout::detach` method when you no longer need the layout instance. If the `IlvGraphLayout::detach` method is not called, some objects may not be deleted. This method also performs cleaning operations on the grapher (properties are removed that may have been added by the layout algorithm on the grapher's objects) and reinitializes parameters as mentioned in *Attaching a Grapher*.

```
void detach()
```

Layout Parameters in IlvGraphLayout

The `IlvGraphLayout` class defines a number of generic parameters. These parameters can be used to customize the layout algorithms.

Although the `IlvGraphLayout` class defines the generic parameters, it does not control how these parameters are used by its subclasses. Each layout algorithm (that is, each subclass of `IlvGraphLayout`) supports a subset of the generic parameters and determines the way in which it uses the parameters. When you create your own layout algorithm by subclassing `IlvGraphLayout`, you decide whether to use the parameters and the way in which you are going to use them.

The `IlvGraphLayout` class defines the following generic parameters:

- ◆ *Allowed Time*
- ◆ *Animation*
- ◆ *Layout Region*
- ◆ *Preserve Fixed Links*
- ◆ *Preserve Fixed Nodes*
- ◆ *Random Generator Seed Value*
- ◆ *Use Default Parameters*

Table 4.2 provides a summary of the generic parameters supported by each layout algorithm. If you are using one of the subclasses provided with IBM ILOG Views, check the documentation for that subclass to know whether it supports a given parameter and whether it interprets the parameter in a particular way.

Allowed Time

Several layout algorithms can be designed to stop computation when a user-defined time specification is exceeded. This may be done for different reasons: as a security measure to

avoid a long computation time on very large graphs or as an upper limit for algorithms that iteratively improve a current solution and have no other criteria to stop the computation.

IBM ILOG Views allows you to specify the allowed time:

```
void IlvGraphLayout::setAllowedTime(IlvRuntimeType time)
```

To obtain the current value, use the method:

```
IlvRuntimeType IlvGraphLayout::getAllowedTime() const
```

If you subclass `IlvGraphLayout`, use the following call to know whether the specified time was exceeded:

```
IlBoolean IlvGraphLayout::isLayoutTimeElapsed() const
```

The time is in seconds. The default value is 32.0 seconds.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, the following method is provided:

```
IlBoolean IlvGraphLayout::supportsAllowedTime() const
```

The default implementation returns `IlFalse`. A subclass can override this method to return `IlTrue` to indicate that this mechanism is supported.

Animation

Some iterative layout algorithms can optionally redraw the graph after each iteration or step. This may create a pleasant animation effect and may be used to keep the user aware of the evolution of the layout computation by showing intermediate results (as a kind of progress bar). However, this increases the duration of the layout because additional redrawing operations need to be performed.

IBM ILOG Views allows you to specify that a redraw of the grapher must be performed after each iteration (or step):

```
void IlvGraphLayout::setAnimate(IlBoolean option)
```

To obtain the current value, use the following method:

```
IlBoolean IlvGraphLayout::isAnimate()
```

The default value is `IlFalse`.

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, the following method is provided:

```
IlBoolean IlvGraphLayout::supportsAnimation()
```

The default implementation returns `IlFalse`. A subclass can override this method to return `IlTrue` to indicate that this mechanism is supported.

Layout Region

Some layout algorithms can control the size of the graph drawing and can take into account a user-defined layout region. For example, you can specify that the layout should fit a given rectangle within a manager view or that it should fit the entire manager view.

IBM ILOG Views allows you to specify a layout region in three different ways:

- ◆ You can indicate that the size of the drawing must fit (exactly or approximately) the size of a given manager view:

```
void IlvGraphLayout::setLayoutRegion(IlvView* view)
```

- ◆ You can specify a region (the rectangle `rect`) that the drawing must fit (exactly or approximately) with the dimensions of the rectangle being given in the manager view coordinates:

```
void IlvGraphLayout::setLayoutRegion(IlvView* view, const
IlvRect& rect)
```

- ◆ You can specify a region (the rectangle `rect`) that the drawing must fit (exactly or approximately) with the dimensions of the rectangle being given in manager (or grapher) coordinates:

```
void IlvGraphLayout::setLayoutRegion(const IlvRect& rect)
```

You must use the last method if you want to perform the layout with no manager view attached to the grapher or if you want to define the layout region in manager coordinates.

You can obtain the rectangle that defines the current layout region using the method:

```
void IlvGraphLayout::getLayoutRegion (IlvRect& rect) const
```

This method returns a copy of the rectangle that the graph drawing must fit (exactly or approximately). The dimensions of the rectangle are in the manager (grapher) coordinates. Depending on the last method you called, one of the following cases can occur:

- ◆ If `IlvGraphLayout::setLayoutRegion(const IlvRect&)` was the last method called, it returns the rectangle with no transformation.
- ◆ If `IlvGraphLayout::setLayoutRegion(IlvView*, const IlvRect&)` was the last method called, it returns the rectangle transformed to the manager coordinates (using the transformer of the view).
- ◆ If `IlvGraphLayout::setLayoutRegion(IlvView*)` was the last method called, it returns a rectangle with the attributes `x=0, y=0`. The attributes, width and height, are equal to the current width and height of the view transformed to the manager coordinates (using the transformer of the view).

- ◆ None of the methods was called. (This is the default behavior.) If at least one manager view is attached to the grapher, it returns a rectangle with the attributes $x=0$, $y=0$. The width and height are equal to the current width and height of the first attached view, transformed to the manager coordinates (using the transformer of the view). If no view is attached, it returns an empty rectangle.

To indicate whether a subclass of `IlvGraphLayout` supports the layout region mechanism, the following method is provided:

```
IlBoolean IlvGraphLayout::supportsLayoutRegion() const
```

The default implementation returns `IlFalse`. A subclass can override this method in order to return `IlTrue` to indicate that this mechanism is supported.

Note that if you are performing the layout using the default settings, there must be at least one manager view (an instance of `IlvView*`) attached to the grapher.

Preserve Fixed Links

At times, you may want some links of the graph to be “pinned” (that is, to stay in their current shape when the layout is performed). You need a way to indicate the links that the layout algorithm cannot reshape. This makes sense especially when using a Semi-automatic layout (the method where the end user fine tunes the layout by hand after the layout is completed) or when using an Incremental layout (the method where the graph and/or the shape of the links is modified after the layout has been performed and then the layout is performed again).

IBM ILOG Views allows you to specify that a link is fixed using the method:

```
void IlvGraphLayout::setFixed(IlAny link, IlBoolean fixed)
```

If `fixed` is `IlTrue`, it means that the link is fixed. To obtain the current setting for a link:

```
IlBoolean IlvGraphLayout::isFixed(IlAny link) const
```

The default value is `IlFalse`.

To remove the fixed attribute from all links in the grapher, use the method:

```
void IlvGraphLayout::unfixAllLinks()
```

Note: *The fixed attributes you may have set will be taken into consideration only if you call the method `void IlvGraphLayout::setPreserveFixedLinks(IlBoolean option)` with an `IlTrue` argument.*

You can read the current option using the method:

```
IlBoolean IlvGraphLayout::isPreserveFixedLinks() const
```

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, the following method is provided:

```
IlBoolean IlvGraphLayout::supportsPreserveFixedLinks() const
```

The default implementation returns `IlFalse`. A subclass can override this method in order to return `IlTrue` to indicate that this mechanism is supported.

Preserve Fixed Nodes

At times, you may want some nodes of the graph to be “pinned” (that is, to stay in their current position when the layout is performed). You need a way to indicate the nodes that the layout algorithm cannot move. This makes sense especially when using a Semi-automatic layout (the method where the user fine tunes the layout by hand after the layout is completed) or when using an Incremental layout (the method where the graph and/or the position of the nodes is modified after the layout has been performed and then the layout is performed again).

IBM ILOG Views allows you to specify that a node is fixed using the method:

```
void IlvGraphLayout::setFixed(IlAny node, IlBoolean fixed)
```

If `fixed` is `IlTrue`, it means that the node is fixed. To obtain the current setting for a node:

```
IlBoolean IlvGraphLayout::isFixed(IlAny node) const
```

The default value is `IlFalse`.

To remove the fixed attribute from all nodes in the grapher, use the method:

```
void IlvGraphLayout::unfixAllNodes()
```

Note: *The fixed attributes you may have set will be taken into consideration only if you call the method `void IlvGraphLayout::setPreserveFixedNodes(IlBoolean option)` with an `IlTrue` argument.*

You can read the current option using the method:

```
IlBoolean IlvGraphLayout::isPreserveFixedNodes() const
```

To indicate whether a subclass of `IlvGraphLayout` supports this mechanism, the following method is provided:

```
IlBoolean IlvGraphLayout::supportsPreserveFixedNodes() const
```

The default implementation returns `IlFalse`. A subclass can override this method in order to return `IlTrue` to indicate that this mechanism is supported.

Random Generator Seed Value

Some layout algorithms use random numbers (or randomly chosen parameters) for which they accept a user-defined seed value. For example, the Random Layout uses the random generator to compute the coordinates of the nodes.

Subclasses of `IlvGraphLayout` that are designed to support this mechanism allow the user to choose one of three ways of initializing the random generator:

- ◆ With a default value that is always the same.
- ◆ With a user-defined seed value that can be changed when re-performing the layout.
- ◆ With an arbitrary seed value, which is different each time. In this case, the random generator is initialized based on the system time.

The user chooses the initialization option depending on what happens when the layout algorithm is performed again on the same graph. If the same seed value is used, the same layout is produced, which may be the desired result. In other situations, the user may want to produce different layouts in order to select the best one. This can be achieved by performing the layout several times using different seed values.

Here is an example of how this parameter can be used in combination with the `IlvRandomLayout` class in your implementation of the method

```
IlvGraphLayout::layout():
```

```
IlvRandomLayout* layout = new IlvRandomLayout();
IlvRandom* random = (layout->isUseSeedValueForRandomGenerator()) ?
    new IlvRandom(layout->getSeedValueForRandomGenerator()) :
    new IlvRandom();
```

To specify the seed value, use the method:

```
void IlvGraphLayout::setSeedValueForRandomGenerator (IlUShort
seed)
```

and to obtain the current value:

```
IlUShort IlvGraphLayout::getSeedValueForRandomGenerator() const
```

The default value is 0.

Note: *The user-defined seed value is used only if you call the method `void IlvGraphLayout::setUseSeedValueForRandomGenerator (IlBoolean option)` with an `IlTrue` argument.*

You can read the current option using the method:

```
IlBoolean IlvGraphLayout::isUseSeedValueForRandomGenerator()
const
```

To indicate whether a subclass of `IlvGraphLayout` supports this parameter, the following method is provided:

```
IlBoolean IlvGraphLayout::supportsRandomGenerator() const
```

The default implementation returns `IlFalse`. A subclass can override this method in order to return `IlTrue` to indicate that this parameter is supported.

Use Default Parameters

All the generic parameters have a default value. After modifying parameters, you may want the layout algorithm to use the default values. Then, you may want to return to your customized values. IBM® ILOG® Views keeps the previous settings when selecting the default values mode. You can switch between the default values mode and the mode for your own settings using the method:

```
void IlvGraphLayout::setUseDefaultParameters(IlBoolean option)
```

To obtain the current value:

```
IlBoolean IlvGraphLayout::isUseDefaultParameters() const
```

The default value is `IlFalse`. This means that any setting you make will be taken into consideration and the parameters that have not been specified will have their default values.

Getting Started with Graph Layout

This chapter provides information to get started using the Graph Layout package of IBM® ILOG® Views. The following topics are covered:

- ◆ *Basic Steps for Using Layout Algorithms: A Summary*
- ◆ *Sample Application*
- ◆ *Launching IBM ILOG Views Studio with the Graph Layout Extension*

Basic Steps for Using Layout Algorithms: A Summary

To use the layout algorithms provided by the Graph Layout package of IBM ILOG Views, you will usually perform the following steps:

1. Create a grapher object (`IlvGrapher`) and fill it with nodes and links.
2. Create an instance of the layout algorithm (any subclass of `IlvGraphLayout`). For details, see *Instantiating a Subclass of `IlvGraphLayout`*.
3. Attach the grapher to the layout instance. See *Attaching a Grapher*.
4. Modify the default settings for the layout parameters, if needed. See *Layout Parameters in `IlvGraphLayout`*.
5. Call the `performLayout` method. See *Performing a Layout*.

6. Read and display information from the layout report. The layout report is an object in which the layout algorithm stores information about its behavior. For details, see *Using a Layout Report*.
7. When the layout instance is no longer needed, detach the grapher from the layout instance. See *Detaching a Grapher*.

A sample application that illustrates these steps is provided with the release. The section Sample Application tells you how to compile and run the application and provides the sample code. You can use this application as an example to get started with the layout algorithms of the Graph Layout package.

Sample Application

The basic steps for using the layout algorithms are illustrated in the sample application provided with this release. The sample uses the Orthogonal Link Layout, but the principles are similar for any of the other layouts.

The source code of the application is named `layoutsample1.cpp` and can be found at the location:

```
<installdir>/samples/layout/userman/layoutsample1.cpp
```

To compile and run the sample, do the following:

1. Go to the `<installdir>/samples/layout/userman/<system>` directory.
2. On UNIX, set the variable that stores the path of dynamic libraries, as explained in the `<installdir>/readme.htm` file delivered with the product.
3. Compile the application:

On UNIX:

```
make
```

On Windows:

```
nmake (in a DOS Console).  
or use the Microsoft Studio Workspace userman.dsw file.
```

4. Run the application:

```
layoutsample1
```

The `layoutsampl1.cpp` contains the following code:

```
// Declare a handle for the layout instance
IlvOrthogonalLinkLayout* layout = new IlvOrthogonalLinkLayout();

// Attach the grapher to the layout instance
layout->attach(grapher);

// Perform the layout and get the layout report
IlvGraphLayoutReport* layoutReport = layout->performLayout();

// Print information from the layout report (optional)
IlvPrint("layout done in %.8g sec., code = %d",
        layoutReport->getLayoutTime(),
        layoutReport->getCode());
// Detach the grapher from the layout instance
layout->detach();

// Delete the layout instance
delete layout;
```

The sample application produces the following graph:

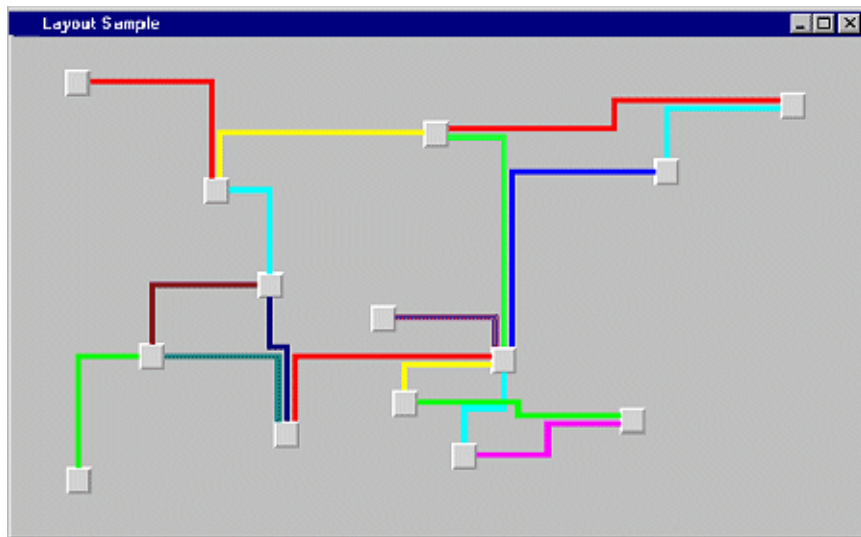


Figure 3.1 Output from Sample Application


Launching IBM ILOG Views Studio with the Graph Layout Extension

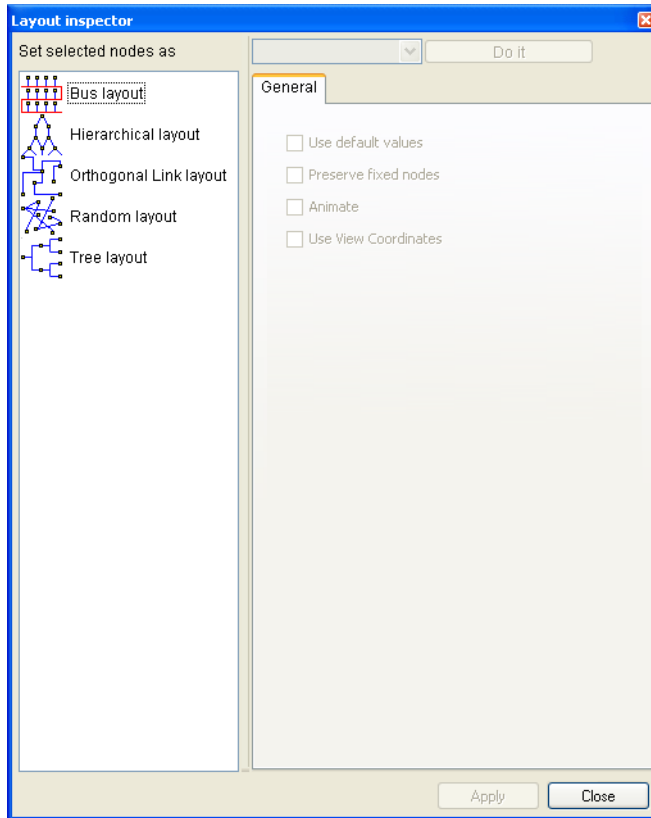
To launch IBM® ILOG® Views Studio with the Graphic Layout extension, do the following:

1. Go to the directory \$ILVHOME/studio/<system> of the IIBM ILOG Views distribution.
2. Type ivfstudio -selectPlugins.
3. A window appears listing the available plug-ins. For Graph Layout, the various layouts are listed.
4. Check the layout(s) and any other plug-ins you may want to use.
5. Click OK to validate and launch IBM ILOG Views Studio with the plug-ins you have selected.

A Quick Look at the Interface

When you launch IBM ILOG Views Studio with the Graph Layout extension, the Main window with the Palettes panel appears on your screen.

1. To use the Graph Layout extension, you must create a grapher buffer. Click File, New, Grapher.
2. On the Buffer toolbar, click  to open the Layout inspector. (Or select the Layout inspector command from the Tools menu.)



3. Populate the grapher buffer with nodes and links.
4. Select a layout from the list of layouts.
5. Click the apply button.
The selected layout is applied.

Layout Algorithms

This chapter describes the layout algorithms of the IBM ILOG Views Graph Layout package. The following topics are covered:

- ◆ *Determining the Appropriate Layout Algorithm*
- ◆ *Tree Layout*
- ◆ *Hierarchical Layout*
- ◆ *Orthogonal Link Layout*
- ◆ *Random Layout*
- ◆ *Bus Layout*

Determining the Appropriate Layout Algorithm


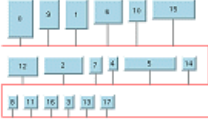
When using the Graph Layout package, you need to determine which of the ready-to-use layout algorithms is appropriate for your particular needs. Some layout algorithms can handle a wide range of graphs. Others are designed for particular classes of graphs and will give poor results or will reject graphs that do not belong to these classes. For example, a Tree Layout algorithm is designed for tree graphs, but not cyclic graphs. Therefore, it is important to lay out a graph using the appropriate layout algorithm.

Table 4.1 can help you determine which of the layout algorithms is best suited for a particular type of graph. Across the top of the table are various classifications of different types of graphs. The layout algorithms appear on the left side of the table. By identifying some the general characteristics of the graph you want to lay out, you can see from the table whether a layout algorithm is suited for that particular type of graph. The illustrations in the table cells provide an example of the drawing produced when a layout algorithm is applied to a particular type of graph. For example, if you know that the structure of the graph is a tree, you can look at the Domain-Independent Graphs/Trees column to see which layout algorithms are appropriate. The Tree Layout, Hierarchical Layout, and Orthogonal Link Layout could all be used. Use the illustrations in the table cells to help you further narrow your choice.

Table 4.1 Layout Algorithms and Common Types of Graphs

Layout	Domain-Independent Graphs				Telecom-Oriented Representations
	Trees	Cyclic Graphs	Combination of Cycles and Trees	Any Graph	
Tree Layout					
Hierarchical Layout					

Table 4.1 Layout Algorithms and Common Types of Graphs (Continued)

Layout	Domain-Independent Graphs				Telecom-Oriented Representations
	Trees	Cyclic Graphs	Combination of Cycles and Trees	Any Graph	
Orthogonal Link Layout					
Bus Layout					 For bus topologies

Generic Parameters Support

The generic parameters of the Graph Layout package allow you to customize the behavior of the layout algorithms to meet specific needs. Table 4.2 indicates the generic parameters that are supported by each layout algorithm. These parameters are defined in the base class of all layout algorithms, `IlvGraphLayout`.

Table 4.2 Generic Parameters Supported by Layout Algorithms

Layout Algorithm	Allowed Time	Animation	Fixed Links	Fixed Nodes	Layout Region	Random Generator Seed Value
Tree Layout	Yes		Yes	Yes		
Hierarchical Layout	Yes		Yes	Yes		
Orthogonal Link Layout	Yes	Yes	Yes	N/A		
Random Layout				Yes	Yes	Yes
Bus Layout				Yes	Yes	

Layout Characteristics

It is often useful to know how certain settings will affect the resulting layout of the graph after the layout algorithm has been applied. Table 4.3 provides additional information about the behavior of the layout algorithms.

Table 4.3 *Layout Characteristics of Layout Algorithms*

Layout Algorithm	Do the initial positions of the nodes affect the layout? ¹	How do I get a different layout of the same graph when I perform the layout a second time?
Tree Layout	Yes	You can completely change the layout by moving nodes or selecting a different root node. To change the dimensions of the graph, use the various offset parameters, or, in some layout modes, the aspect ratio parameter.
Hierarchical Layout	No	You can use specified node level indices to change the level structure. You can use specified node position indices to change the node order within the levels. You can change the layout by changing the link priorities. To change only the dimensions of the graph, use the various offset parameters.
Orthogonal Link Layout	No	You can completely change the layout by changing the link connection policy. You can change the dimensions of the graph by using the link offset and final segment length parameters.
Random Layout	No	This is the default behavior when using the default parameter settings (the random generator is initialized differently each time).
Bus Layout	No	You change the dimensions of the graph by using the various dimensional parameters.

¹ All of the layout classes provided in IBM ILOG Views (except the Orthogonal Link Layout) support the fixed nodes mechanism. This means that you can specify nodes that cannot be moved during the layout.

Tree Layout

In this section, you will learn about the *Tree Layout* algorithm provided with the IBM® ILOG® Views Graph Layout package (class `IlvTreeLayout` from the library `ilvtree`).

Samples

Here are some sample drawings produced with the Tree Layout:

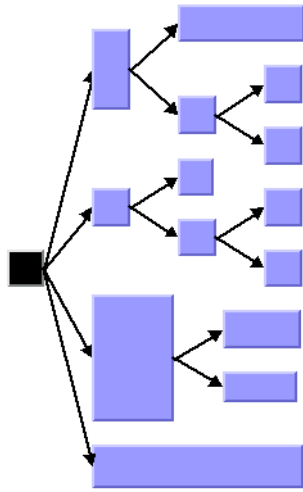


Figure 4.1 Sample Tree Layout in Free Layout Mode with Center Alignment and Flow Direction to the Right

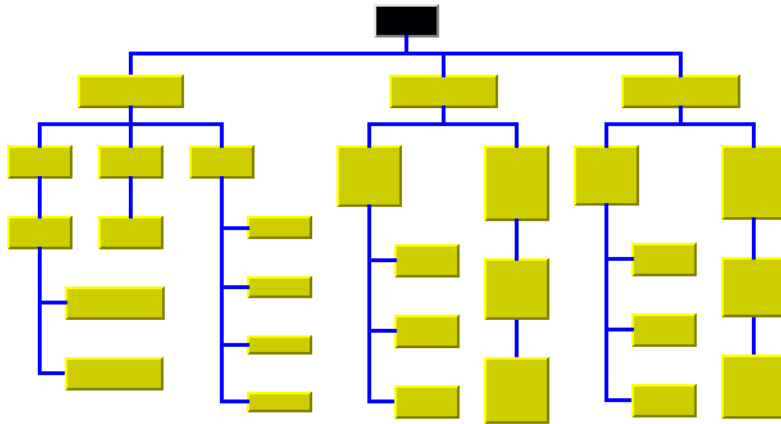


Figure 4.2 Sample Tree Layout with Flow Direction to the Bottom, Orthogonal Link Style, and Tip-Over Alignment at Some Leaf Nodes

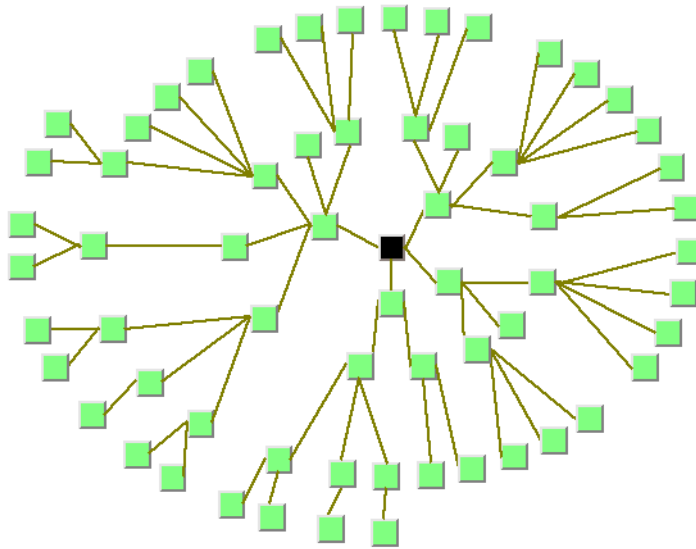


Figure 4.3 Sample Tree Layout in Radial Layout Mode with Aspect Ratio 1.3

What Types of Graphs?

- ◆ Tree layout is primarily designed for pure trees. It can also be used for non-trees, that is, for cyclic graphs. In this case, the algorithm computes and uses a spanning tree of the graph, ignoring all links that do not belong to the spanning tree.
- ◆ Directed and undirected trees. If the links are directed, the algorithm automatically chooses the canonical root node. If the links are undirected, you can choose a root node.
- ◆ Connected and disconnected graphs. If the graph is not connected, the layout algorithm treats each connected component separately. Each component has exactly one root node. In this case, a forest of trees is laid out.

Application Domains

Application domains of the Tree Layout include:

- ◆ Business processing (organizational charts)
- ◆ Software management/software (re-)engineering (UML diagrams, call graphs)
- ◆ Database and knowledge engineering (decision trees)
- ◆ The World Wide Web (Web site maps)

Features

- ◆ Takes into account the size of the nodes so that no overlapping occurs.
- ◆ Optionally, reshapes the links to give them an orthogonal form (alternating horizontal and vertical line segments).
- ◆ Various layout modes: *free*, in *levels*, *radial*, or *automatic tip-over*.
- ◆ In the free layout mode: arranges the children of each node, starting recursively from the root, so that the links flow uniformly in the same direction.
- ◆ In the level layout mode: partitions the nodes into levels, and arranges the levels horizontally or vertically.
- ◆ In radial layout mode: partitions the nodes into levels, and arranges the levels in circles or ellipses around the root.
- ◆ In the tip-over modes: arranges the nodes in a similar way to the free layout mode, but tries to tip over children automatically to better fit the layout to the given aspect ratio.
- ◆ Provides several alignment and offset options.
- ◆ Allows specifying nodes that must be directly neighbored.
- ◆ Takes the old position of nodes into account. Positions the nodes without changing the relative order of the nodes in the tree, so that the layout is stable on incremental changes of the graph.
- ◆ Very efficient, scalable algorithm. Produces a nice layout quickly even if the number of nodes is huge.

Limitations

- ◆ If “orthogonal” is not specified as the link style (see Link Style), some links may overlap nodes, depending on the size of the nodes and the alignment and offset parameters.
- ◆ The layout algorithm first determines a spanning tree of this graph. If the graph is not a pure tree, some links will not become part of the spanning tree. These links are ignored. Hence, they may cross other links or overlap nodes in the final layout.
- ◆ The algorithm tries to preserve the relative order of the children of each node for incremental stability. It uses a heuristic to calculate the relative order from the old positions. The heuristic may fail if children overlap on their old positions or are neither horizontally nor vertically aligned.
- ◆ Despite preserving the relative order of the children, in rare cases the layout is not perfectly stable in the radial modes. Subsequent layouts may rotate the nodes around the root, although the relative circular order of the nodes within their circular levels is still preserved.

- ◆ The tip-over layout modes perform several tries with different tip-over alignment options according to various heuristics. From these, the algorithm picks the layout that best fits the given aspect ratio. This may not be the optimal layout for the aspect ratio, but it is the best layout among the performed tries. To calculate the absolutely best fitting layout is computationally infeasible (it is generally an NP-complete problem).

Brief Description of the Algorithm

For the Tree Layout, the core algorithm for the layout modes *free*, *level*, and *radial* works in just two steps and is very fast.

- ◆ Step 1: Calculation of the spanning tree
- ◆ Step 2: Calculation of node positions and link shapes

The variations for the layout mode *tip-over* perform the second step several times and pick the result that best fits the given aspect ratio (the ratio between width and height of the drawing area). Therefore, the tip-over layout modes are slower.

Step 1: Calculation of the spanning tree

If the graph is disconnected, the layout algorithm chooses a *root node* for each connected component. Starting from the root node, it traverses the graph to choose the links of the *spanning tree*. If the graph is a pure tree, all links will be chosen. If the graph has cycles, some links will not become part of the spanning tree. These links are called *non-tree links*, while the links of the spanning tree are called *tree links*. The non-tree links are ignored in step 2 of the algorithm.

The *root* is the black node in Figure 4.1, Figure 4.2 and Figure 4.3. In the spanning tree, each node except the root has a *parent* node. All nodes that have the same parent are called *children* with respect to the parent and *siblings* among themselves. Nodes without children are called *leaves*. Each child at a node starts a subtree (also called a *branch* of the tree). Figure 4.4 illustrates a spanning tree.

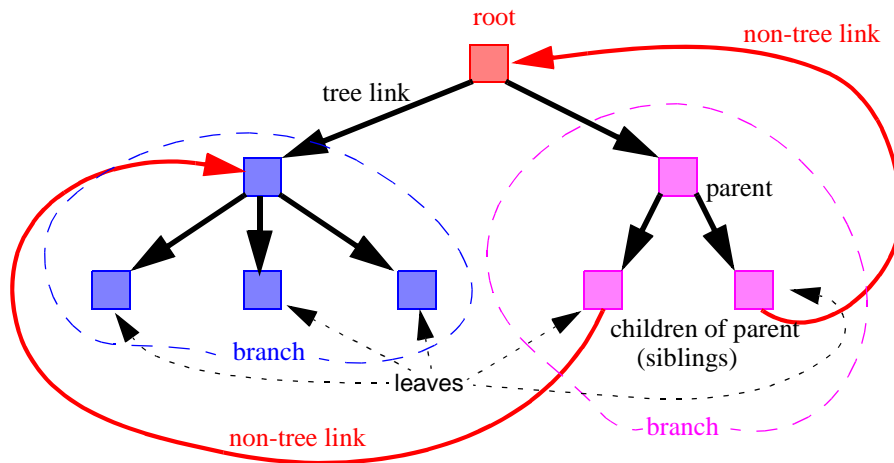


Figure 4.4 Sketch of Spanning Tree

Step 2: Calculation of node positions and link shapes

The layout algorithm arranges the nodes according to the layout mode and the offset and alignment options. In free and level modes, the nodes are arranged horizontally or vertically so that all tree links flow roughly in the same direction. In the radial modes, the nodes are arranged in circles or ellipses around the root so that all tree links flow radially away from the root. Finally, the link shapes are calculated according to the link style and alignment options.

Code Sample

Below is a code sample using the `IlvTreeLayout` class:

```
// ...

IlvGrapher* grapher = new IlvGrapher(display);
IlvView* view = new IlvView(display, "", "", IlvRect(0, 0, 100, 100));
grapher->addView(view);
view->show();

// ... Fill in the grapher with nodes and links here
// ... Suppose we have added rootNode as a node in the grapher
IlvGraphic* rootNode = 0;

IlvTreeLayout* layout = new IlvTreeLayout();
layout->attach(grapher);

// Specify the root node, orientation and alignment
layout->setRoot(rootNode);
layout->setFlowDirection(IlvRight);
layout->setGlobalAlignment(IlvLayoutCenterAlignment);

// Perform the layout
IlvGraphLayoutReport* layoutReport = layout->performLayout();
if (layoutReport->getCode() != IlvLayoutReportLayoutDone)
    IlvWarning("Layout not done. Error code = %d\n", layoutReport->getCode());

// If this grapher is not anymore subject of layout:
layout->detach();

// Once the layout algorithm is not anymore needed:
delete layout;
```

Parameters

The Tree Layout uses generic parameters, common to other graph layouts, and specific parameters applicable in tree layouts only. Refer to the following sections for general information on parameters among the graph layouts:

- ◆ *Generic Parameters Support*
- ◆ *Layout Characteristics*

The Tree Layout parameters are described in detail in this topic under:

- ◆ *Generic Parameters*
- ◆ *Specific Parameters (All Tree Layout Modes)*

Generic Parameters

The `IlvTreeLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class:

- ◆ *Allowed Time*
- ◆ *Preserve Fixed Links*
- ◆ *Preserve Fixed Nodes*

The following paragraphs describe the particular way in which these parameters are used by this subclass.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see [Allowed Time](#).) If the layout stops early because the allowed time has elapsed, the nodes and links are not moved from their positions before the layout call.

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (For more information on link parameters in the `IlvGraphLayout` class, see [Preserve Fixed Links and Link Style](#).)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (For more information on node parameters in the `IlvGraphLayout` class, see [Preserve Fixed Nodes](#).) Moreover, the layout algorithm ignores fixed nodes completely and also does not route the links that are incident to the fixed nodes. This can result in undesired overlapping nodes and link crossings. However, this feature is useful for individual, disconnected components that can be laid out independently.

Specific Parameters (All Tree Layout Modes)

The following parameters are specific to the `IlvTreeLayout` class. They apply to all layout modes.

- ◆ *Root Node*
- ◆ *For Experts: Further Options for Root Nodes*
- ◆ *Position*
- ◆ *Compass Directions*
- ◆ *Layout Mode*

Root Node

The final layout is influenced mainly by the choice of the root node.

The root node is placed in a prominent position. For instance, in a top-down drawing with free layout mode it is placed at the top of the tree; with radial mode it is placed at the center of the tree.

The spanning tree is calculated starting from the root node. If the graph is disconnected, the layout algorithm needs one root node for each connected component.

The layout algorithm automatically selects a root node when needed. It uses a heuristic that calculates preferences for all nodes to become a root. It chooses the node with the highest preference. The heuristic gives nodes without incoming links the highest preference, and leaf nodes without outgoing links the lowest preference. Hence, in a directed tree, the canonical root is always chosen automatically.

It is possible to influence the choice of the root node. To select a node explicitly as the root, use the method:

```
void IlvTreeLayout::setRoot(IlAny node);
```

This gives the node the maximal preference to become the root during layout. If only one node is specified this way, the algorithm selects this node. If several nodes of the same connected component are specified this way, the layout algorithm chooses one of them as the root.

For Experts: Further Options for Root Nodes

The layout algorithm manages a list of the root nodes that have been specified by the `IlvTreeLayout::setRoot` method. To obtain this list, use the method:

```
const IlList* IlvTreeLayout::getSpecRoots() const;
```

After layout, you can also retrieve the list of root nodes that were actually used by the algorithm. This list is not necessarily the same as the list of specified roots. For instance, it contains the chosen root nodes if none were specified, or if too many were specified. To obtain the list of root nodes that were used by the algorithm, call the method:

```
const IlList* IlvTreeLayout::getCalcRoots() const;
```

Note that the returned lists are constant. You should not change them directly. However, you can iterate over the lists and retrieve the root nodes, as in the following example, to perform special root operations:

```
IlvLink* link = layout->getCalcRoots()->getFirst();
while (link) {
    root = (IlvGraphic*)link->getValue();
    link = link->getNext();
    // ... perform special operation with root
}
```


To directly manipulate the root node preference value of an individual node, you can use the method:

```
IlvTreeLayout::setRootPreference(IlAny node, IlInt preference);
```

In this case, the layout uses the specified value instead of the heuristically calculated preference for this node. The normal preference value should be between 0 and 10000. Specifying a root node explicitly corresponds to setting the preference value to 10000. If you want to prohibit a node from becoming the root, specify a preference value of zero (0).

A negative preference value indicates that the layout algorithm should recalculate the root node preference, using the heuristic. You can clear the root node setting as follows: If a root was specified by the `IlvTreeLayout::setRoot` method but this node should no longer be the root in subsequent layouts, then call:

```
layout->setPreference(node, -1);
```

This also removes the node from the list of specified roots.

Position

The position of the *top left corner* of the layout can be set to (10, 10) in the following way:

```
IlvPoint point(10, 10);  
layout->IlvTreeLayout::setPosition(point, IlTrue);
```

If the graph consists of only a single tree, it is often more convenient to set the position of the root node instead. This can be done by the same method, passing `IlFalse` instead of `IlTrue`:

```
layout->IlvTreeLayout::setPosition(point, IlFalse);
```

If no position is specified, the layout keeps the root node at its previous position.

To obtain the current position, use:

```
const IlvPoint* IlvTreeLayout::getPosition() const;
```

This method returns a null pointer if no position was specified. If it returns a point, you can query whether the specified position is either the top left corner or the position of the first root node:

```
IlBoolean IlvTreeLayout::isRootPosition() const;
```

Compass Directions

To simplify the explanations of the layout parameters, we use the compass directions *north*, *south*, *east* and *west*. The center of the root node of a tree is considered the north pole.

In the nonradial layout modes, the link flow direction always corresponds to the south direction. If the root node is placed at the top of the drawing, then north is at the top, south at the bottom, east on the right, and west on the left side. If the root node is placed at the left border of the drawing, then north is on the left, south on the right, east at the top, and west at the bottom.

In the radial layout modes, the root node is placed in the center of the drawing, hence the meaning of north and south depends on the position relative to the root: the north side of the node is the side closer to the root, and the south side is the side that is further away from the root. The east direction is counterclockwise around the root, and the west direction is clockwise around the root. This is similar to a cartographic map of a real globe that shows the area of the north pole as if you were looking down at the top of the globe.

Compass directions are used to give uniform names to certain layout options. They occur in the alignment options, the level justification option and the east-west neighboring feature, which are explained later. In Figure 4.5 and Figure 4.15, a compass icon shows the compass directions in these drawings.

Layout Mode

The tree layout algorithm has various layout modes. To select a layout mode, use the method:

```
void IlvTreeLayout::setLayoutMode(IlvTreeLayoutMode mode);
```

To obtain the current layout mode, call:

```
IlvTreeLayoutMode IlvTreeLayout::getLayoutMode() const;
```

The type `IlvTreeLayoutMode` is an `IIList` type defined in the file `ilviews/layout/tree.h`. The choices of layout modes are shown in Table 4.4:

Table 4.4 *Tree Layout Modes*

IIList Type	See Section
<code>IlvTreeLayoutFreeMode</code>	<i>Free Layout Mode</i>
<code>IlvTreeLayoutLevelMode</code>	<i>Level Layout Mode</i>
<code>IlvTreeLayoutRadialMode</code>	<i>Radial Layout Mode</i>
<code>IlvTreeLayoutAlternatingRadialMode</code>	<i>Alternating Radial Mode</i>
	<i>Tip-Over Layout Modes</i>
<code>IlvTreeLayoutTipOverMode</code>	<i>Tip Over Fast</i>
<code>IlvTreeLayoutTipRootsOverMode</code>	<i>Tip Roots Over</i>
<code>IlvTreeLayoutTipLeavesOverMode</code>	<i>Tip Leaves Over</i>
<code>IlvTreeLayoutTipRootsAndLeavesOverMode</code>	<i>Tip Roots and Leaves Over</i>

Free Layout Mode

The free layout mode arranges the children of each node starting recursively from the root so that the links flow roughly in the same direction. For instance, if the link flow direction is

top-down, the root node is placed at the top of the drawing. Siblings (that is, nodes with the same parent) are justified at their top borders, but nodes of different tree branches (that is, nodes with different parents) are not justified.

The following statement sets the free layout mode:

```
layout->IlvTreeLayout::setLayoutMode(IlvTreeLayoutFreeMode);
```

Additional parameter information for free tree layout is as follows:

- ◆ *Flow Direction*
- ◆ *Alignment*
 - *Global Alignment*
 - *Alignment of Individual Nodes*
 - *Tip-over Alignment*
- ◆ *Link Style*
 - *Global Link Style*
 - *Individual Link Style*
- ◆ *Spacing Parameters*
- ◆ *For Experts: Further Spacing Parameters*

Flow Direction

The flow direction parameter specifies the direction of the tree links. The compass icons show the compass directions in these layouts. This is illustrated in Figure 4.5.

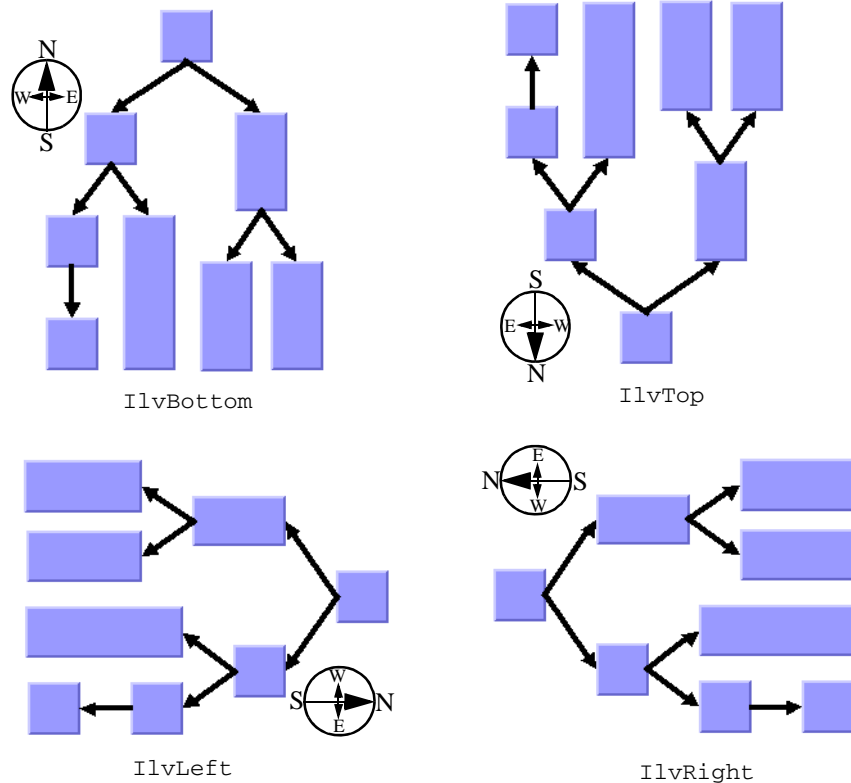


Figure 4.5 Flow Directions

If the flow direction is to the bottom, the root node is placed topmost. Each parent node is placed above its children, which are normally arranged horizontally.

Note: The alignment option specifies how children are arranged. Normal alignment here means any alignment option except the tip-over alignment option.

If the flow direction is to the right, the root node is placed leftmost. Each parent node is placed to the left of its children, which are normally arranged vertically. To specify the flow direction, use the following method:

```
void IlvTreeLayout::setFlowDirection(IlvDirection direction);
```

The valid values for `direction` are:

- ◆ `IlvRight` (the default)
- ◆ `IlvLeft`
- ◆ `IlvBottom`

◆ IlvTop

To obtain the current choice, use the following method:

```
IlvDirection IlvTreeLayout::getFlowDirection() const;
```

Alignment

The alignment option controls how a parent is placed relative to its children. The alignment can be set globally, in which case all nodes are aligned in the same way, or locally on each node, with the result that different alignments occur in the same drawing.

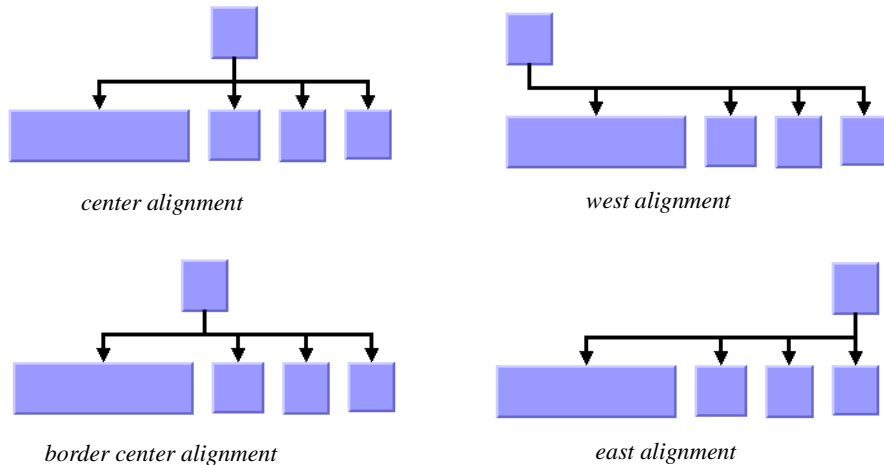


Figure 4.6 Alignment Options

Global Alignment

To set the global alignment, use the following method:

```
void IlvTreeLayout::setGlobalAlignment(IlvLayoutAlignment alignment);
```

The valid values for alignment are:

◆ IlvLayoutCenterAlignment (the default)

The parent is centered over its children, taking the center of the children into account.

◆ IlvLayoutBorderCenterAlignment

The parent is centered over its children, taking the border of the children into account. If the size of the first and the last child varies, the border center alignment places the parent closer to the larger child than the default center alignment.

◆ IlvLayoutEastAlignment

The parent is aligned with the border of its easternmost child. For instance, if the flow direction is to the bottom, east is the direction to the right. If the flow direction is to the top, east is the direction to the left. See *Compass Directions* for details.

◆ `IlvLayoutWestAlignment`

The parent is aligned with the border of its westernmost child. For instance, if the flow direction is to the bottom, west is the direction to the left. If the flow direction is to the right, west is the direction to the bottom. See *Compass Directions* for details.

◆ `IlvLayoutTipOverAlignment`

The children are arranged sequentially instead of in parallel, and the parent node is placed with an offset to the children. For details see *Tip-over Alignment*.

◆ `IlvLayoutMixedAlignment`

Each parent node can have a different alignment. In this case the alignment of each individual node can be set, with the result that different alignments can occur in the same graph.

To obtain the current choice, use the following method:

```
IlvLayoutAlignment IlvTreeLayout::getGlobalAlignment() const;
```

Alignment of Individual Nodes

All nodes have the same alignment unless the global alignment is set to `IlvLayoutMixedAlignment`. Only when the global alignment is set to “mixed” can each node have an individual alignment style.

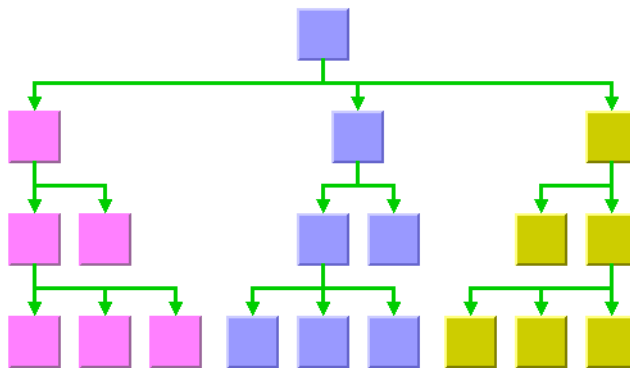


Figure 4.7 *Different Alignments Mixed in the Same Drawing*

To set and retrieve the alignment of an individual node, use the following methods:

```
void IlvTreeLayout::setAlignment(IlAny node, IlvLayoutAlignment alignment);
```

```
IlvLayoutAlignment IlvTreeLayout::getAlignment(IlAny node) const;
```

The valid values for the individual alignments of a node are:

- ◆ IlvLayoutCenterAlignment (the default)
- ◆ IlvLayoutBorderCenterAlignment
- ◆ IlvLayoutEastAlignment
- ◆ IlvLayoutWestAlignment
- ◆ IlvLayoutTipOverAlignment

Tip-over Alignment

Normally the children of a node are placed in a *parallel* arrangement with siblings directly neighbored to each other. Tip-over alignment means a *sequential* arrangement of the children instead.

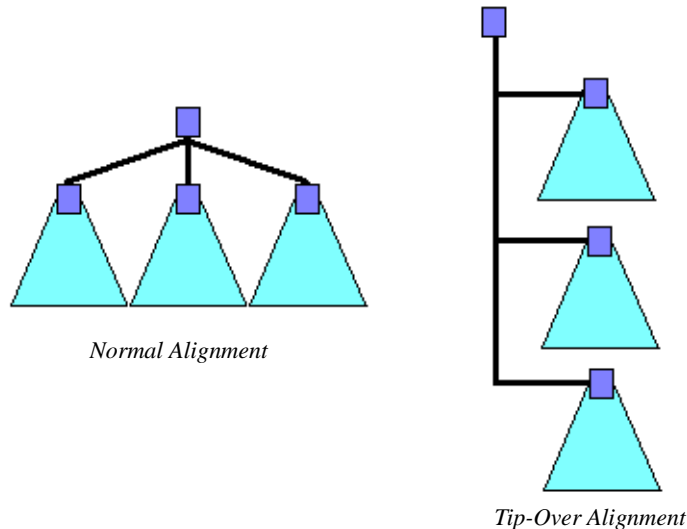


Figure 4.8 Normal Alignment and Tip-over Alignment

Tip-over alignment is useful if the tree has many leaves. With normal alignment, this would result in the layout being very wide. If the global alignment style is tip-over, the drawing is very high instead. In order to balance the width and height of the drawing, you can set the global alignment to “mixed”:

```
layout-  
>IlvTreeLayout::setGlobalAlignment(IlvLayoutMixedAlignment);
```

and the individual alignment to “tip-over” for some parents with a high number of children:

```
layout->IlvTreeLayout::setAlignment(parent,
IlvLayoutTipOverAlignment);
```

Tip-over alignment can be specified explicitly for some (or all) nodes. Furthermore, the Tree Layout offers layout modes that automatically determine when to tip over, yielding a drawing fit to a given aspect ratio. These layout modes are described in Tip-Over Layout Modes.

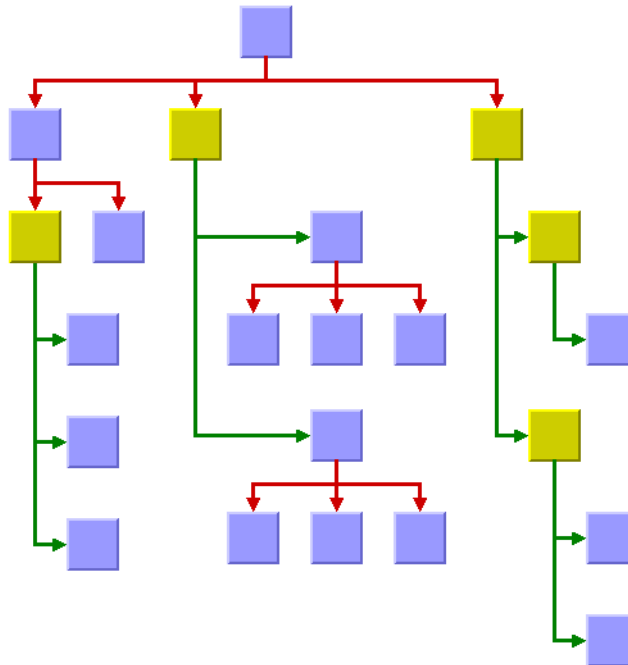


Figure 4.9 Tip-over Alignment

Tip-over alignment works very well with the Orthogonal Link Layout.

Link Style

When the layout algorithm moves the nodes, straight-line links—such as instances of `IlvLinkImage`—will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvDoubleSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any). Also, you can specify that the links be reshaped into an “orthogonal” form. You can set the link style

globally, in which case all links have the same kind of shape, or locally on each link, in which case different link shapes occur in the same drawing.

Note: *The layout algorithm calls the method*

IlvGrapherAdapter::ensureReshapeableLinks on the attached graph model to ensure that all the links can be reshaped as needed. With an IlvGrapher, this method may replace links with the appropriate type of link and install the appropriate link connector on the nodes. For details on the graph model, see Using the Graph Model.

Global Link Style

To set the global link style, the following method is provided:

```
void IlvTreeLayout::setGlobalLinkStyle(IlvLayoutLinkStyle style);
```

The valid values for `style` are:

◆ `IlvLayoutNoReshapeLinkStyle`

None of the links is reshaped in any manner.

◆ `IlvLayoutStraightLineLinkStyle`

All the intermediate points of the links (if any) are removed. This is the default value. See Figure 4.1 and Figure 4.3 as examples.

◆ `IlvLayoutOrthogonalLinkStyle`

The links are reshaped in an orthogonal form (alternating horizontal and vertical segments). See Figure 4.2 and Figure 4.9 as examples.

◆ `IlvLayoutMixedLinkStyle`

Each link can have a different link style. In this case, the style of each individual link can be set to have different link shapes occurring on the same graph.

To obtain the current choice, use the following method:

```
IlvLayoutLinkStyle getLinkStyle() const;
```

Individual Link Style

All links have the same style of shape unless the global link style is `IlvLayoutMixedStyle`. Only when the global link style is set to “mixed” can each link have an individual link style.

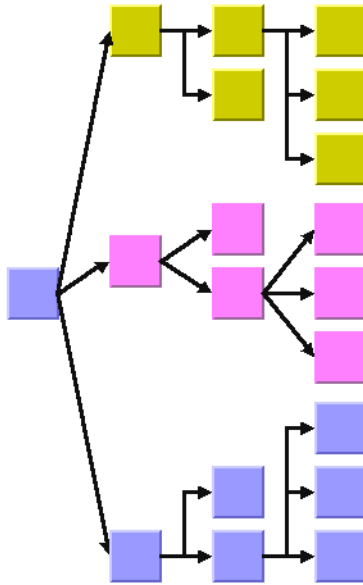


Figure 4.10 *Different Link Styles Mixed in the Same Drawing*

To set and retrieve the style of an individual link, use the following methods:

```
void IlvTreeLayout::setLinkStyle(IlAny link, IlvLayoutLinkStyle
style);
IlvLayoutLinkStyle IlvTreeLayout::getLinkStyle(IlAny link) const;
```

The valid values for the individual alignments of a node are:

- ◆ IlvLayoutStraightLineLinkStyle (the default)
- ◆ IlvLayoutNoReshapeLinkStyle
- ◆ IlvLayoutOrthogonalLinkStyle

Note: *The link style of a Tree Layout graph requires links in an IlvGrapher that can be reshaped. Links of type IlvLinkImage, IlvOneLinkImage, IlvDoubleLinkImage, IlvOneSplineLinkImage, and IlvDoubleSplineLinkImage cannot be reshaped. You can use the class IlvPolylineLinkImage instead.*

Spacing Parameters

The spacing of the layout is controlled mainly by three spacing parameters: the distance between a parent and its children, the minimal distance between siblings, and the minimal distance between nodes of different branches. For instance, if the flow direction is to the top or bottom, the offset between parent and children is vertical, while the sibling offset and the branch offset are horizontal.

For tip-over alignment, there is a fourth spacing parameter: the minimal distance between branches starting at a node with tip-over alignment. This offset is always orthogonal to the normal branch offset, that is, if the flow direction is to the top or bottom, the tip-over branch offset is vertical.

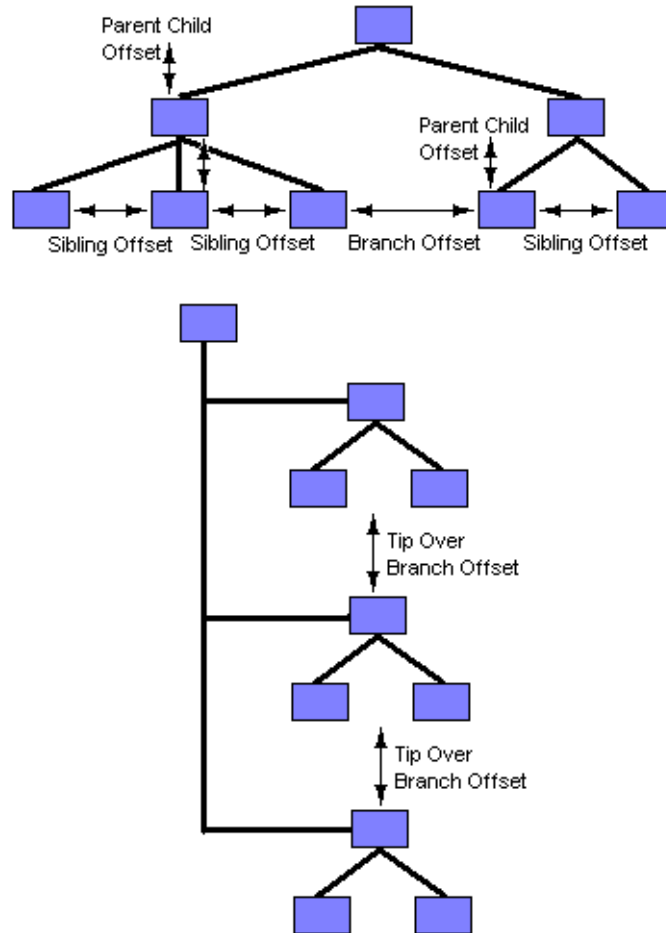


Figure 4.11 Spacing Parameters

The spacing parameters can be set by the following methods:

```
void IlvTreeLayout::setParentChildOffset(IlvPos offset);
void IlvTreeLayout::setSiblingOffset(IlvPos offset);
void IlvTreeLayout::setBranchOffset(IlvPos offset);
void IlvTreeLayout::setTipOverBranchOffset(IlvPos offset);
```

They can be obtained by the corresponding methods:

```
IlvPos IlvTreeLayout::getParentChildOffset() const;
IlvPos IlvTreeLayout::getSiblingOffset() const;
IlvPos IlvTreeLayout::getBranchOffset() const;
IlvPos IlvTreeLayout::getTipOverBranchOffset() const;
```

For Experts: Further Spacing Parameters

If the link style is orthogonal, the shape of the links from the parent to its children looks like a fork (see Figure 4.7). The position of the bend points in this shape can be influenced by the *orthogonal fork percentage*, a value between 0 and 100. This is a percentage of the parent child offset. If the orthogonal fork percentage is 0, the link shape forks directly at the parent node. If the percentage is 100, the link shape forks at the child node. A good choice is between 25 and 75. This percentage can be set and obtained by the following methods:

```
void IlvTreeLayout::setOrthForkPercentage(IlInt percentage);
IlInt IlvTreeLayout::getOrthForkPercentage() const;
```

If the link style is not orthogonal, links may overlap neighboring nodes. This happens only in very few cases, for instance, if a link starts at a very small node that is neighbored by a huge node. This deficiency can be fixed by increasing the branch offset. However, this influences the layout globally, affecting nodes without that deficiency. To avoid a global change, you can change the *overlap percentage* instead, which is a value between 0 and 100. This value is used by an internal heuristic of the layout algorithm that considers a node to be smaller by this percentage. The default percentage is 30. This usually results in a better usage of the space. However, if very small nodes are neighbored to huge nodes, it is recommended decreasing the overlap percentage or setting it to 0 to switch this heuristic off to avoid links overlapping nodes. The overlap percentage can be set and obtained by the following methods:

```
void IlvTreeLayout::setOverlapPercentage(IlInt percentage);
IlInt IlvTreeLayout::getOverlapPercentage() const;
```

Tip: It is recommended that you always set the *orthogonal fork percentage* to a larger value than the *overlap percentage*.

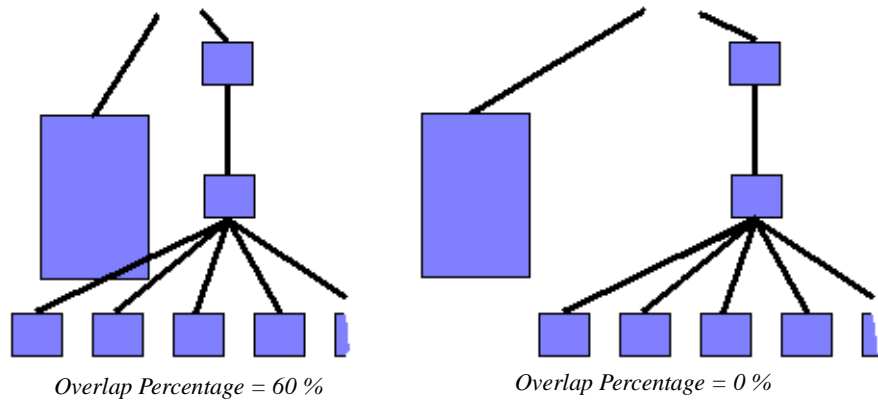


Figure 4.12 The Effect of the Overlap Percentage

Level Layout Mode

The level layout mode partitions the node into levels and arranges the levels horizontally or vertically. The root is placed at level 0, its children at level 1, the children of those children at level 2, and so on. In contrast to the free layout mode, in level layout mode the nodes of the same level are justified with each other even if they are not siblings (that is, they do not have the same parent). Figure 4.13 shows the same graph in free layout mode and in level layout mode.

The following statement sets the level layout mode:

```
layout->IlvTreeLayout::setLayoutMode(IlvTreeLayoutLevelMode);
```

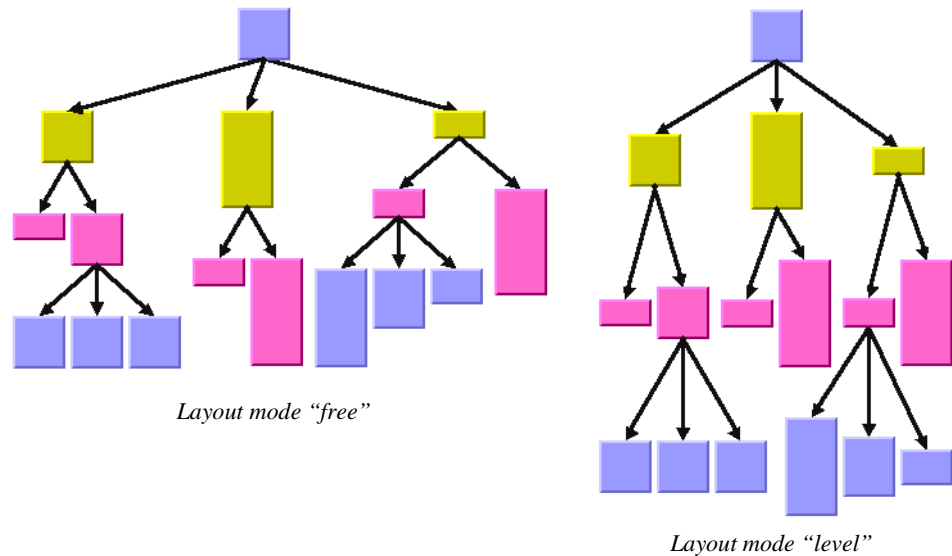


Figure 4.13 Free Layout Mode and Level Layout Mode

Additional parameter information for level tree layout is as follows:

- ◆ *General Parameters*
- ◆ *Level Justification*

General Parameters

Most layout parameters that work for the free layout mode work as well for the level layout mode. You can set the flow direction, the spacing offsets, the global or individual link style, and the global or individual alignment. See Free Layout Mode for details.

The differences from the free layout mode are:

- ◆ The tip-over alignment does not work in level layout mode.
- ◆ The parent-child offset parameter controls the spacing between the levels. In level layout mode, it is the minimal distance between parent and its children, while in free layout mode, it is the exact distance between parent and its children.
- ◆ The overlap percentage has no effect in level layout mode.

Level Justification

In level layout mode with flow direction to the top or bottom, the nodes are organized in horizontal levels such that the nodes of the same level are placed approximately at the same y-coordinate. The nodes can be justified, depending on whether the top border, the bottom border, or the center of all nodes of the same level should have the same y-coordinate.

In flow direction to the left or right, the nodes are organized in vertical levels approximately at the same x-coordinate. The nodes of the same level can be justified at the left border, at the right border, or at the center.

To distinguish the justification independently from the flow direction, we use the directions north and south (see section Compass Directions). The north border of a node is the border that is closer to the level where its parent is placed, and the south border of a node is the border that is closer to the level where its children are placed. If the flow direction is to the bottom, the level justification north means that the nodes are justified at the top border, and south means that the nodes are justified at the bottom border. If the flow direction is to the top, it is converse: north means the bottom border and south means the top border. If the flow direction is to the right, then north means the left border and south means the right border.

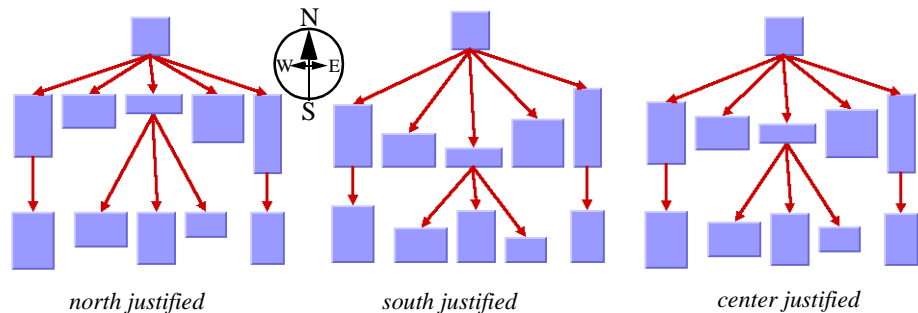


Figure 4.14 Level Justification

To specify the level justification, use the following method:

```
void IlvTreeLayout::setLevelJustification(IlvLayoutAlignment
justification);
```

To obtain the current value, use the method:

```
IlvLayoutAlignment IlvTreeLayout::getLevelJustification() const;
```

The valid choices for the justification are:

- ◆ `IlvLayoutCenterAlignment` (the default)
- ◆ `IlvLayoutNorthAlignment`
- ◆ `IlvLayoutSouthAlignment`

Radial Layout Mode

The radial layout mode partitions the node into levels and arranges the levels in circles around the root node. Figure 4.15 shows an example of the radial layout mode. The compass icons show the compass directions in this drawing.

The following statement sets the radial layout mode:

```
layout->IlvTreeLayout::setLayoutMode(IlvTreeLayoutRadialMode);
```

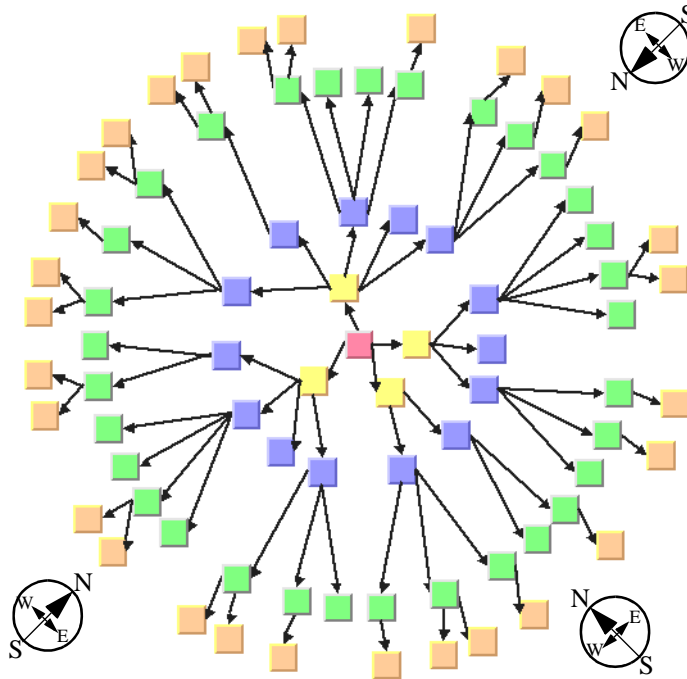


Figure 4.15 Radial Layout Mode

The following statement sets the radial layout mode:

```
layout->IlvTreeLayout::setLayoutMode(IlvTreeLayoutRadialMode);
```

Additional parameter information for radial tree layout is as follows:

- ◆ *General Parameters*
- ◆ *Alternating Radial Mode*
- ◆ *Aspect Ratio*
- ◆ *Spacing Parameters*
- ◆ *For Experts: Further Parameters*

General Parameters

Most layout parameters that work for the free and level layout mode work as well for the radial layout mode. You can set the spacing offsets, the level justification, the global or individual link style, and the global or individual alignment. See Free Layout Mode and Level Layout Mode for details.

Here is a list of differences from the other modes:

- ◆ The tip-over alignment does not work in radial layout mode.
- ◆ The orthogonal link style does not work in radial mode.
- ◆ The parent-child offset parameter controls the minimal distance between the circular levels. However, it is sometimes necessary to increase the offset between circular levels to obtain enough space on the circle to place all nodes of a level.
- ◆ The level justification *north* means justification at the inner border of the circular level (that is, towards the root), and the level justification *south* means justification at the outer border of the circular level (that is, away from the root).
- ◆ The level justifications *north* and *south* sometimes result in node overlapping.
- ◆ The overlap percentage has no effect in radial layout mode.

Alternating Radial Mode

If levels contain many nodes, it is sometimes necessary to increase the radius of the circular level to provide enough space on the circumference of the circle for all the nodes. This may result in a considerable distance from the previous level. To avoid this, there is an *alternating* radial mode. This mode places the nodes of a level alternating onto two circles instead of one circle, resulting in a better space usage of the layout.

The alternating radial mode uses two circles only when necessary. For many small and light trees, there will be no difference from the normal radial mode. Only for large graphs with a high degree of children will the alternating radial mode have an effect.

The following statement sets the alternating radial layout mode:

```
layout->c( IlvTreeLayoutAlternatingRadialMode );
```

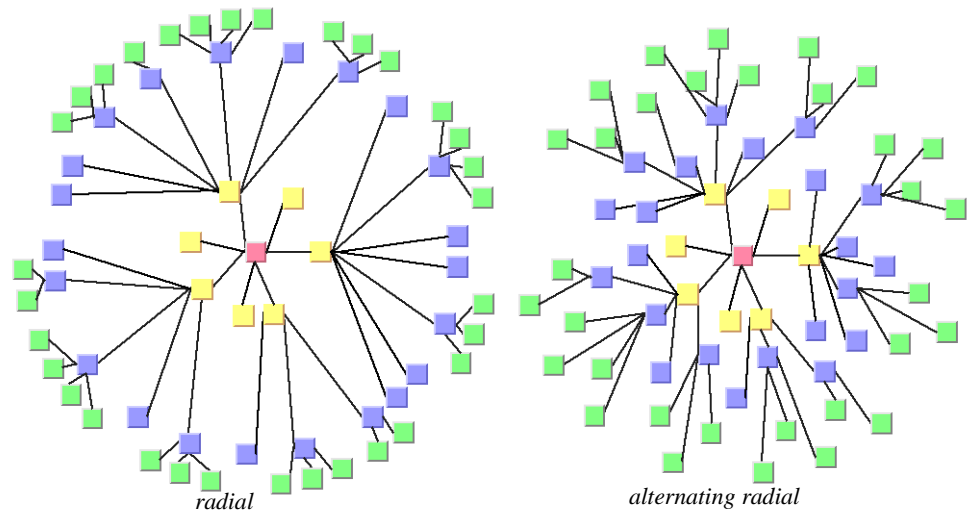


Figure 4.16 Radial and Alternating Radial Layout Mode

Aspect Ratio

If the drawing area is not a square, arranging levels as circles is not always the best choice. You can specify the aspect ratio of the drawing area to better fit the layout to the area. In this case, the algorithm uses ellipses instead of circles. See Figure 4.3 for an example.

If the drawing area is a view (a subclass of `IlvAbstractView`), you can use this method:

```
void IlvTreeLayout::setAspectRatio(const IlvAbstractView* view);
```

If the drawing area is given only as a rectangle, use:

```
void IlvTreeLayout::setAspectRatio(const IlvRect& rect);
```

If neither a view nor a rectangle is given, you can calculate the aspect ratio from the width and height of the drawing area as `aspectRatio = width/height` and use this method:

```
void IlvTreeLayout::setAspectRatio(IlDouble aspectRatio);
```

To obtain the current aspect ratio, call:

```
IlDouble IlvTreeLayout::getAspectRatio() const;
```

Spacing Parameters

The spacing parameters of the radial layout mode are controlled by the same methods as for the free or level layout mode:

```
void IlvTreeLayout::setParentChildOffset(IlvPos offset);
```

```
void IlvTreeLayout::setSiblingOffset(IlvPos offset);
```

```
void IlvTreeLayout::setBranchOffset(IlvPos offset);
```

Note that the sibling and branch offsets are minimal distances tangential to the circles or ellipses, while the parent-child offset is a minimal distance radial to the circles or ellipses. Figure 4.17 shows the spacing parameters in radial mode.

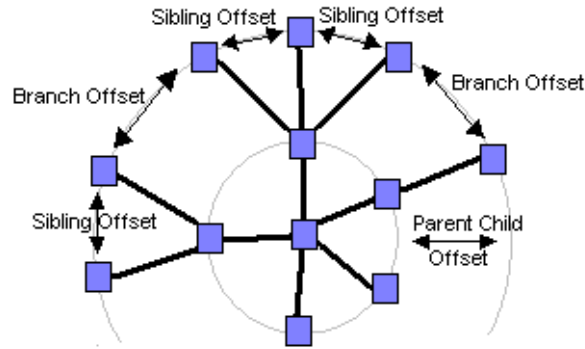


Figure 4.17 Spacing Parameters in Radial Layout Mode

For Experts: Further Parameters

If a node has many children, they may cover a major part of the circle, and hence are placed nearly 360 degree around the node. This can result in links overlapping other nodes. The deficiency can be fixed by increasing the offset between parent and children. However, this influences the layout globally, also affecting nodes without the deficiency. To avoid a global change, you can limit the maximal angle between the two rays from the parent (if it is not the root) to its two outermost children. This increases the offset between parent and children only where necessary. Use the following method to set and obtain the angle in degrees:

```
void IlvTreeLayout::setMaxChildrenAngle(IIInt angle);
IIInt IlvTreeLayout::getMaxChildrenAngle() const;
```

Recommended values are between 30 and 180. Setting the value to 0 means the angle is unrestricted. The calculation of the angle is not very precise above 180 degrees, or if the aspect ratio is not 1.0.

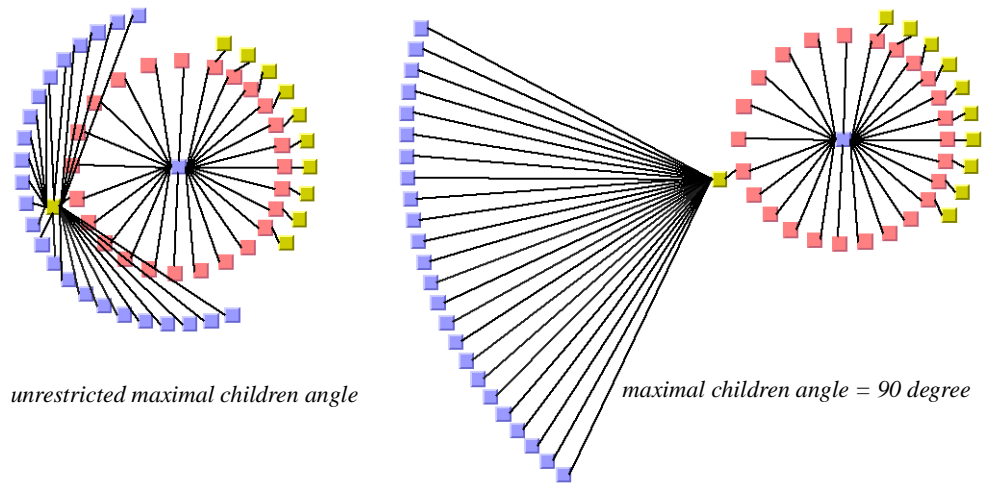


Figure 4.18 *Maximal Children Angle*

Tip-Over Layout Modes

As in radial layout, drawing in free layout mode can be adjusted to the aspect ratio of the area. Free layout mode can also use tip-over alignment to balance the drawing between height and depth.

While tip-over alignment can be specified explicitly for individual nodes, the Tree Layout Algorithm also has layout modes that automatically use tip-over alignment when needed. These are the tip-over layout modes.

The tip-over layout modes work as follows: Several tries are performed in free layout mode. For each try, some tip-over alignments are set for individual nodes, while the specified alignment of all other nodes is preserved. The algorithm picks the try that best fits the specified aspect ratio of the drawing area.

The aspect ratio can be set by one of the following methods (see Aspect Ratio in the Radial Layout Mode):

```
void IlvTreeLayout::setAspectRatio(const IlvAbstractView* view);
void IlvTreeLayout::setAspectRatio(const IlvRect& rect);
void IlvTreeLayout::setAspectRatio(IlDouble aspectRatio);
```

The tip-over modes are slightly more time-consuming than the other layout modes. For very large trees, it is recommended that you set the allowed layout time to a high value (that is, 60 seconds) when using the tip-over modes, by:

```
layout->IlvGraphLayout::setAllowedTime(60);
```

This avoids running short of time for sufficient iterations of the layout algorithm. Because it would be too time-consuming to check all possibilities of tip-over alignment use, there are heuristics that check only certain tries, according to four different strategies:

- ◆ *Tip Leaves Over*
- ◆ *Tip Roots Over*
- ◆ *Tip Roots and Leaves Over*
- ◆ *Tip Over Fast*

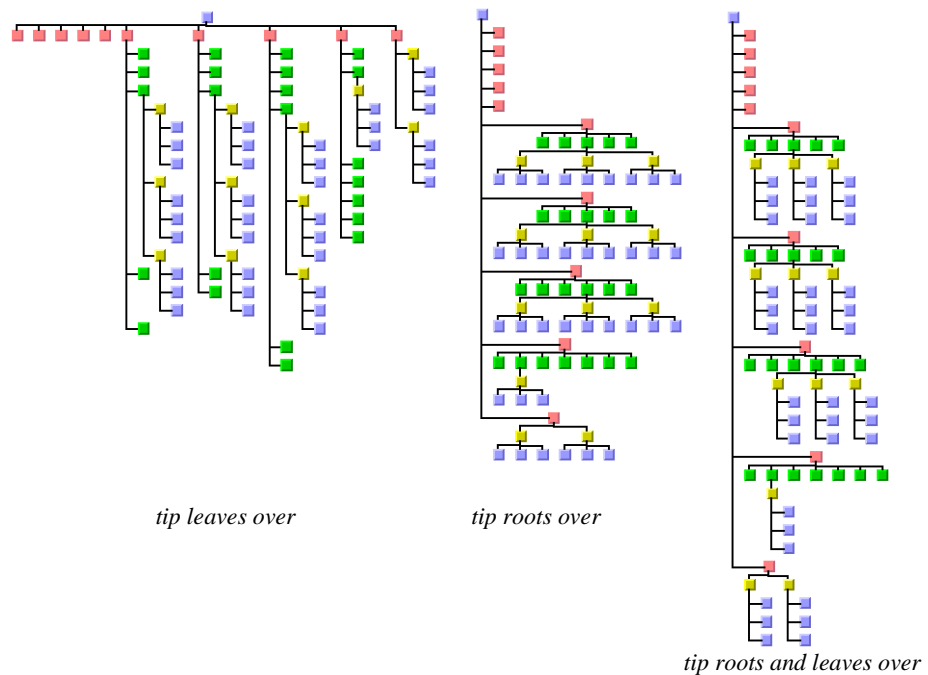


Figure 4.19 *Tip-Over Strategies*

Tip Leaves Over

To use this tip-over strategy, set the layout mode in this way:

```
layout-
>IlvTreeLayout::setLayoutMode(IlvTreeLayoutTipLeavesOverMode);
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the leaves, then the leaves and their parents, then additionally the parents of these parents, and so on. As a result, the nodes closer to the root use normal alignment, and the nodes closer to the leaves use tip-over alignment.

Tip Roots Over

To use this tip-over strategy, set:

```
layout-
>IlvTreeLayout::setLayoutMode(IlvTreeLayoutTipRootsOverMode);
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the root node, then the root and its children, then additionally the children of these children, and so on. As a result, the nodes closer to the leaves use normal alignment, and the nodes closer to the root use tip-over alignment.

Tip Roots and Leaves Over

To use this tip-over strategy, set:

```
layout-
>IlvTreeLayout::setLayoutMode(IlvTreeLayoutTipRootsAndLeavesOverM
ode);
```

The heuristic first tries the layout without any additional tip-over options. Then it tries to tip over the root node and the leaves simultaneously; then the root and its children, and the leaves and its parent; then additionally the children of these children and the parents of these parents, and so on. As result, the nodes in the middle of the tree use normal alignment, and the nodes closer to the root or leaves use the tip-over alignment.

This is the slowest strategy because it includes all tries of the strategy “*tip leaves over*” as well as all tries of the strategy “*tip roots over*.”

Tip Over Fast

The fast tip-over is a compromise between all other strategies. To use this strategy, set the layout mode in this way:

```
layout->IlvTreeLayout::setLayoutMode(IlvTreeLayoutTipOverMode);
```

The heuristic tries a small selection of the other strategies, not all possibilities. Therefore, it is the fastest strategy for large graphs.

It is possible that all four strategies yield the same result, because the strategies are not disjoint, that is, certain tries are performed with all four strategies. Furthermore, the tip-over modes do not necessarily produce the optimal layout that gives the best possible fit to the aspect ratio. The reason is that certain unusual configurations of tip-over alignment are never tried, because otherwise the running time would be too high.

For Experts: Further Tips and Tricks

Here are more featured items for using Tree Layout:

- ◆ *East-West Neighbors*
- ◆ *Retrieving Link Categories*

- ◆ *Sequences of Layouts with Incremental Changes*
- ◆ *Interactive Editing*
- ◆ *Specifying the Order of Children*

East-West Neighbors

You can specify that two unrelated nodes must be directly neighbored in a direction perpendicular to the flow direction. In level or radial layout mode, both nodes are placed in the same level next to each other. In free layout or tip-over mode, both nodes are placed aligned at the north border. Such nodes are called *east-west neighbors*, because one node is placed as the direct neighbor on the east side of the other node (and the other node becomes the direct neighbor on the west side of the first node). See also *Compass Directions*).

Technically, both nodes are treated as parent and child, even if there may be no link in between. Therefore, one of the two nodes can have a real parent, but the other node should not, because its virtual parent is its *east-west neighbor*.

This feature can be used, for example, for annotating nodes in a typed syntax tree occurring in compiler construction. Figure 4.20 illustrates the usage.

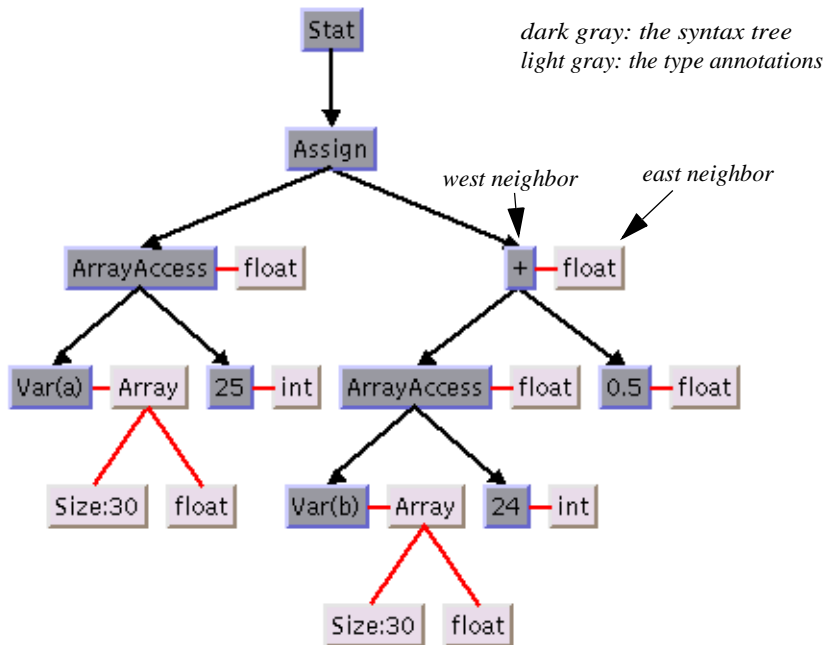


Figure 4.20 Annotated Syntax Tree of Statement `a[25] = b[24] + 0.5;`

Use the following method to specify that two nodes are *east-west neighbors*:

```
void IlvTreeLayout::setEastWestNeighboring(IlAny eastNode, IlAny westNode);
```

You can also use the following method, which is identical except for the reversed parameter order:

```
void IlvTreeLayout::setWestEastNeighboring(IlAny westNode, IlAny eastNode);
```

If the flow direction is to the bottom, the latter method may be easier to remember, because in this case west is to the left of east in the layout, similar to the text flow of the parameters.

To obtain which node is the east or west neighbor of a node, call:

```
IlAny IlvTreeLayout::getEastNeighbor(IlAny node);
IlAny IlvTreeLayout::getWestNeighbor(IlAny node);
```

Note that each node can have maximally one east neighbor and one west neighbor, because they are *directly* neighbored. If more than one direct neighbor is specified, it will be partially ignored. Cyclic specifications can cause conflict as well. For instance, if node B is the east neighbor of node A and node C is the east neighbor of B, then node A cannot be the east neighbor of C. (Strictly speaking, such cycles could be technically possible in some situations in the radial layout mode, but nonetheless they are not allowed in any layout mode.)

If B is the east neighbor of A, then A is automatically the west neighbor of B. On the other hand, the east neighbor of A can itself have another east neighbor. This allows creating chains of east-west neighbors, which is a common way to visualize lists of trees. Two examples are shown in Figure 4.21.

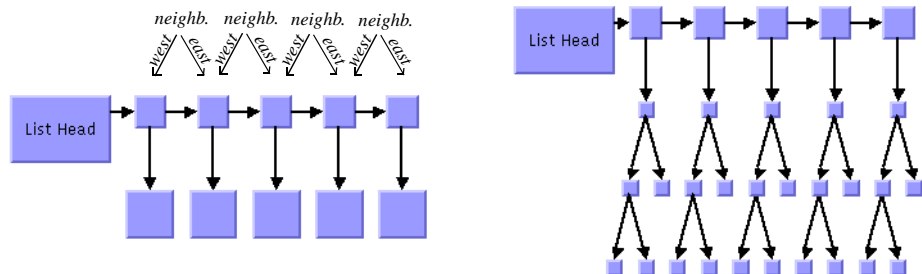


Figure 4.21 Chains of East-West Neighbors to Visualize Lists of Trees

Retrieving Link Categories

The Tree Layout Algorithm works on a spanning tree, as mentioned in a Brief Description of the Algorithm. If the graph to be laid out is not a pure tree, the algorithm ignores some links. In order to perform a special treatment of such links, it is possible to obtain the list of non-tree links.

There are parents and children in the spanning tree. We distinguish the following link categories:

- ◆ *Forward tree link*: a link from a parent to its child.
- ◆ *Backward tree link*: a link from a child to its parent. If the link is drawn as a directed arrow, the arrow will point converse to the flow direction.
- ◆ *Non-tree link*: a link between two unrelated nodes; neither one is a child of the other.

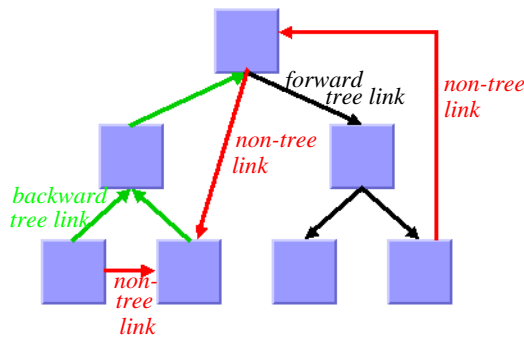


Figure 4.22 Link Categories

The layout algorithm uses these link categories internally but does not store them permanently for the sake of time and memory efficiency. If you want to perform a special treatment on some link categories (for example, to call the Orthogonal Link Layout on the non-tree links), you must specify *before layout* that you want to access the link categories *after layout*. In order to do this, use the method

`IlvTreeLayout::setCategorizingLinks` in the following way:

```
layout->setCategorizingLinks( IlTrue );
// now perform a layout
layout->performLayout();
// now you can access the link categories
```

After layout, the link categories are available in three lists that can be obtained by these methods:

```
const IlList* IlvTreeLayout::getCalcForwardTreeLinks() const;
const IlList* IlvTreeLayout::getCalcBackwardTreeLinks() const;
const IlList* IlvTreeLayout::getCalcNonTreeLinks() const;
```

These lists get filled each time the layout is called, unless you switch *categorizing links* back to `IlFalse`.

Note that the returned lists are constant. You should not change them directly. However, you can iterate over the lists and retrieve the links of each category, for example, to perform a special operation for non-tree links:

```
IlvLink* link = layout->getCalcNonTreeLinks()->getFirst();
while (link) {
    linkimage = (IlvLinkImage*)link->getValue();
    link = link->getNext();
    // ... perform special operation with linkimage
}
```

Sequences of Layouts with Incremental Changes

You can work with trees that become out-of-date from time to time, for example, those that need to be extended with more children. If you perform a layout after an extension, you probably want to identify the parts that were already laid out in the original graph. The Tree Layout Algorithm supports such incremental changes because it takes the previous positions of the nodes into account. It preserves the relative order of the children in the next layout.

Interactive Editing

The fact that the relative order is preserved is particularly useful during interactive editing. It allows you to easily correct the layout. For instance, if the first layout places a node A left to its sibling B but you need it in reverse order, you can simply move A to the right of B and start a new layout to clean up the drawing. In this second layout, A will stay to the right of B, and the subtree of A will “follow” node A.

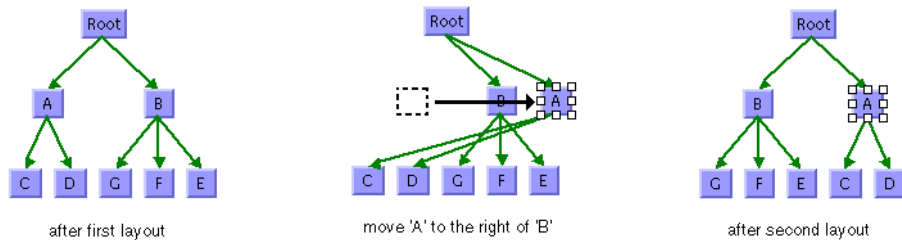


Figure 4.23 Interactive Editing to Achieve Specific Order of Children

Specifying the Order of Children

Some applications require a specific relative order of children in the tree. That means, for instance—with flow direction to the bottom—which child must be placed to the left of another child. Even if the graph was never laid out, you can use the coordinates to specify a certain order of the children at a node in the following way:

- ◆ In free or level layout mode with flow direction to the bottom or top, determine the maximal width “ W ” of all nodes. Simply move the child that should be leftmost to the coordinate $(0, 0)$, and the child that should get the i th relative position (in order from left to right) to coordinate $((W+1)*i, 0)$.
- ◆ If the flow direction is to the left or to the right, determine the maximal height H of all nodes. Move the child that should be topmost to the coordinate $(0, 0)$, and the child that should get the i th relative position (in the order from top to bottom) to coordinate $(0, (H+1)*i)$.
- ◆ In the radial layout modes, determine the maximal diagonal $D = W/2 + H$ of all nodes. If the position of the parent is (x, y) before layout, then move the child that should be the first in the circular order to the coordinate $(x, y+D)$, and the child that should get the i th relative position in the circular order to coordinate $(x+D*i, y+D)$.

If you want to specify a relative order for all nodes in radial mode, you must do this for the parents before you do it for the children. In this case, the movements can be easily performed during a depth-first traversal from the root to the leaves.

The layout that is performed after the movement arranges the children with the desired relative order.

Hierarchical Layout

In this section, you will learn about the *Hierarchical Layout* algorithm provided with the IBM ILOG Views Graph Layout package (class `IlvHierarchicalLayout` from the library `ilvhierarchical`).

Samples

Here are some sample drawings produced with the Hierarchical Layout:

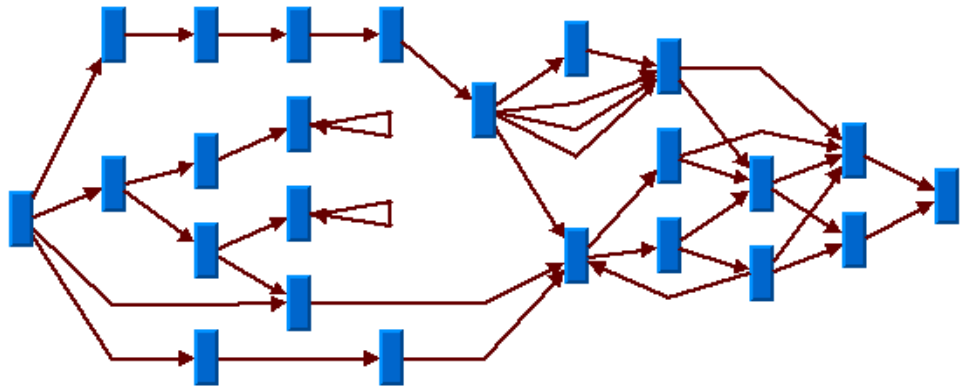


Figure 4.24 Sample Layout with Self-Loops, Multiple Links, and Cycles

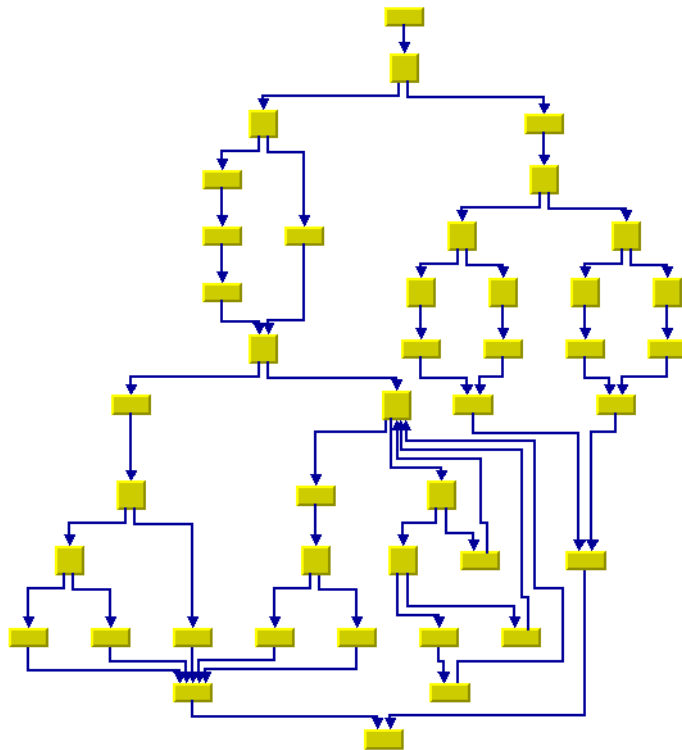


Figure 4.25 Flowchart with Orthogonal Link Style

What Types of Graphs?

Any type of graph:

- ◆ Preferably graphs with directed links (The algorithm takes the link direction into account.)
- ◆ Connected and disconnected graphs
- ◆ Planar and nonplanar graphs

Application Domains

Application domains of the Hierarchical Layout include:

- ◆ Electrical engineering (logic diagrams, circuit block diagrams)
- ◆ Industrial engineering (industrial process diagrams, schematic design diagrams)
- ◆ Business processing (workflow diagrams, process flow diagrams, PERT charts)
- ◆ Software management/software (re-)engineering (UML diagrams, flowcharts, data inspector diagrams, call graphs)
- ◆ Database and knowledge engineering (database query graphs)
- ◆ CASE tools (designs diagrams)

Features

- ◆ Organizes nodes without overlaps in horizontal or vertical levels.
- ◆ Arranges the graph such that the majority of links are short and flow uniformly in the same direction (from left to right, from top to bottom, and so on).
- ◆ Reduces the number of link crossings. Most of the time, produces drawings with no crossings or only a small number of crossings.
- ◆ Often produces balanced drawings that emphasize the symmetries in the graph.
- ◆ Supports self-links (that is, links with the same origin and destination node), multiple links between the same pair of nodes, and cycles.
- ◆ Efficient, scalable algorithm. Produces a nice layout relatively quickly for most sparse and medium-dense graphs, even if the number of nodes is very large.
- ◆ Provides several alignment and offset options.
- ◆ The computation time depends on the number of nodes, the number of levels, and the number of links that cross several levels. Most of the time, the links are placed between adjacent levels, which keeps the computation time small.

Limitations

- ◆ The algorithm tries to minimize the number of link crossings (which is generally an NP-complete problem). It is mathematically impossible to solve this problem quickly for any graph size. Therefore, the algorithm uses a very fast heuristic that obtains a good layout, but not always with the theoretical minimum number of link crossings.
- ◆ The algorithm tries to place the nodes such that all links point uniformly in the same direction. It is impossible to place cycles of links in this way. For this reason, it sometimes produces a graph where a small number of links are reversed to point in the opposite direction. The algorithm tries to minimize the number of reversed links (which, again, is an NP-complete problem). Therefore, the algorithm uses a very fast heuristic resulting in a good layout, but not always with the theoretical minimum number of reversed links.
- ◆ The computation time required to obtain an appropriate drawing depends most significantly on the number of bends in the links. Since the algorithm places one bend whenever a link crosses a level, the number of bends can grow relatively quickly if the layout requires many long links that span several levels. Therefore, the layout process may become very time-consuming for dense graphs (the number of links is relatively high compared to the number of nodes) or for graphs that require a large number of node levels.

Brief Description of the Algorithm

This algorithm works in four steps.

Step 1: Leveling

The nodes are partitioned into groups. Each group of nodes forms a level. The objective is to group the nodes in such a way that the links always point from a level with a smaller index to a level with a larger index.

Step 2: Crossing Reduction

The nodes are sorted within each level. The algorithm tries to keep the number of link crossings small when, for each level, the nodes are placed in this order on a line (see Figure 4.26). This ordering results in the relative position index of each node within its level.

Step 3: Node Positioning

From the level indices and position indices, balanced coordinates for the nodes are calculated. For instance, for a layout where the link flow is from top to bottom, the nodes are placed along horizontal lines such that all nodes belonging to the same level have (approximately) the same y-coordinate. The nodes of a level with a smaller index have a smaller y-coordinate than the nodes of a level with a higher index. Within a level, the nodes with a smaller position index have a smaller x-coordinate than the nodes with a higher position index.

Step 4: Link Routing

The shapes of the links are calculated such that the links bypass the nodes at the level lines. In many cases, this requires that a bend point be created whenever a link needs to cross a level line. In a top-to-bottom layout, these bend points have the same y-coordinate as the level line they cross. (Note that these bend points also obtain a position index).

Figure 4.26 shows how the Hierarchical Layout algorithm uses the level and position indices to draw the graph.

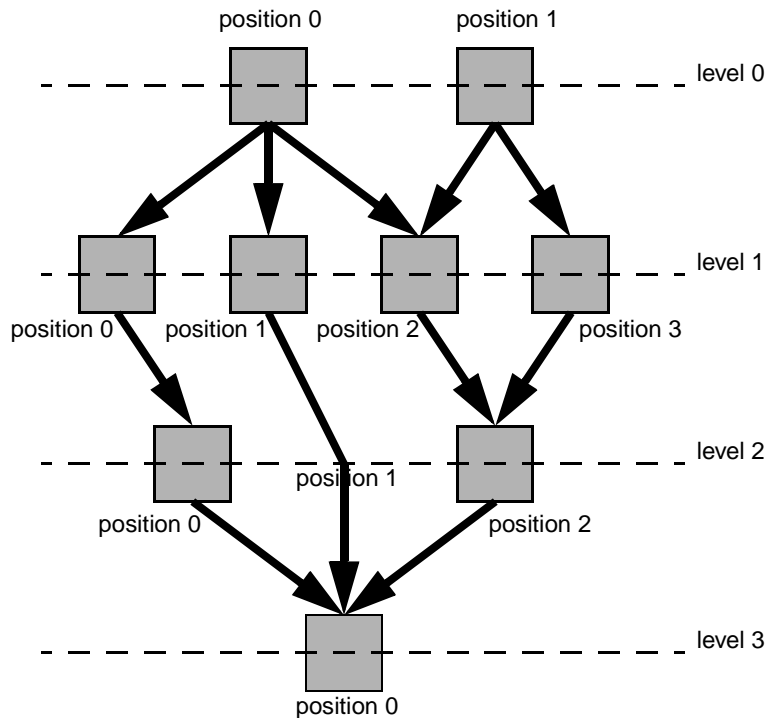


Figure 4.26 *Level and Position Indices*

The steps of the layout algorithm can be affected in several ways. For instance, you can specify the desired level index that the algorithm should choose for a node in Step 1 or the desired relative node position within the level in Step 2. You can also specify the justification of the nodes within a level and the style of the links shapes.

Code Sample

Below is a code sample using the `IlvHierarchicalLayout` class:

```
// ...
IlvGrapher* grapher = new IlvGrapher(display);

// ... Fill in the grapher with nodes and links here

IlvHierarchicalLayout* layout = new IlvHierarchicalLayout();
layout->attach(grapher);

// Set the layout parameters, e.g., flow to bottom:
layout->setFlowDirection(IlvBottom);

// ...
// Perform the layout
IlvGraphLayoutReport* layoutReport = layout->performLayout();
if (layoutReport->getCode() != IlvLayoutReportLayoutDone)
    IlvWarning("Layout not done, Error code = %d\n", layoutReport->getCode());

// ...
// If this grapher is not anymore subject of layout:
layout->detach();

// ...
// Once the layout algorithm is not anymore needed:
delete layout;
```

Parameters

The Hierarchical Layout uses generic parameters, common to other graph layouts, and specific parameters applicable in hierarchical layouts only. Refer to the following sections for general information on parameters among the graph layouts:

- ◆ *Generic Parameters Support*
- ◆ *Layout Characteristics*

The Hierarchical Layout parameters are described in detail in this topic under:

- ◆ *Generic Parameters*
- ◆ *Specific Parameters*

Generic Parameters

The `IlvHierarchicalLayout` class supports the following generic parameters defined in the `IlvGraphLayout` class:

- ◆ *Allowed Time*

- ◆ *Preserve Fixed Links*
- ◆ *Preserve Fixed Nodes*

The following paragraphs describe the particular way in which these parameters are used by this subclass.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see *Allowed Time*.) If the layout stops early because the allowed time elapsed, the nodes and links are not moved at all and remain in the same position they had before the call of layout.

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (For more information on link parameters in the `IlvGraphLayout` class, see *Preserve Fixed Links and Link Style*.)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (For more information on node parameters in the `IlvGraphLayout` class, see *Preserve Fixed Nodes*.) Moreover, the layout algorithm ignores fixed nodes completely and also does not route the links that are incident to the fixed nodes. This can result in undesired overlapping nodes and link crossings. However, this feature is useful for individual, disconnected components that can be laid out independently.

Specific Parameters

The following parameters are specific to the `IlvHierarchicalLayout` class:

- ◆ *Flow Direction*
- ◆ *Level Justification*
- ◆ *Link Style*
- ◆ *Connector Style*
- ◆ *Link Priority*
- ◆ *Level Index*
- ◆ *Position Index*
- ◆ *Spacing Parameters*

Flow Direction

The flow direction parameter specifies the direction in which the majority of the links should point. If the flow direction is to the top or to the bottom, the node levels are oriented

horizontally and the links mostly vertically. If the flow direction is to the left or to the right, the node levels are oriented vertically and the links mostly horizontally.

If the flow direction is to the bottom, the nodes of the level with index 0 are placed at the top border of the drawing. The nodes with level index 0 are usually the root nodes of the drawing (that is, the nodes without incoming links). If the flow direction is to the top, the nodes with level index 0 are placed at the bottom border of the drawing. If the flow direction is to the right, the nodes are placed at the left border of the drawing.

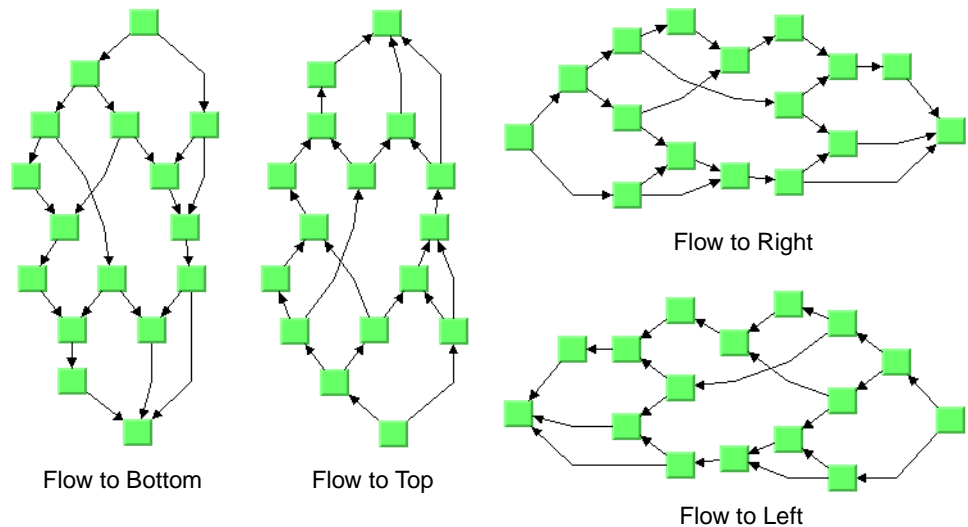


Figure 4.27 *Flow Directions*

To specify the flow direction, use the following method:

```
void IlvHierarchicalLayout::setFlowDirection(IlvDirection
direction);
```

The valid values for *direction* are:

- ◆ IlvRight (the default)
- ◆ IlvLeft
- ◆ IlvBottom
- ◆ IlvTop

To obtain the current choice, use the following method:

```
IlvDirection IlvHierarchicalLayout::getFlowDirection() const;
```

Level Justification

If the layout uses horizontal levels, the nodes of the same level are placed approximately at the same y-coordinate. The nodes can be justified, depending on whether the top border, or the bottom border, or the center of all nodes of the same level should have the same y-coordinate.

If the layout uses vertical levels, the nodes of the same level are placed approximately at the same x-coordinate. In this case, the nodes can be justified to be aligned at the left border, at the right border, or at the center of the nodes that belong to the same level.

To specify the level justification, use the following method:

```
void IlvHierarchicalLayout::setLevelJustification(IlvDirection justification);
```

If the flow direction is to the top or to the bottom, the valid values for justification are:

- ◆ IlvTop
- ◆ IlvBottom
- ◆ IlvCenter (the default)

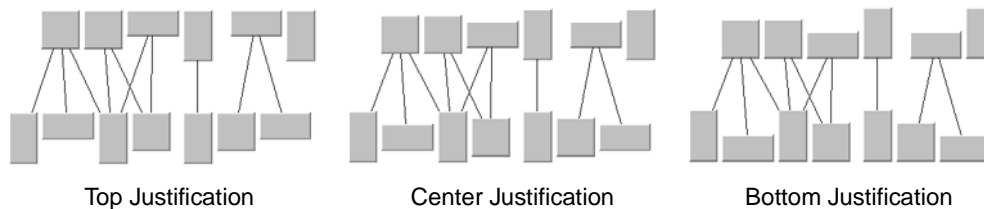


Figure 4.28 *Level Justification for Horizontal Levels*

If the flow direction is to the left or to the right, the valid values for justification are:

- ◆ IlvLeft
- ◆ IlvRight
- ◆ IlvCenter (the default)

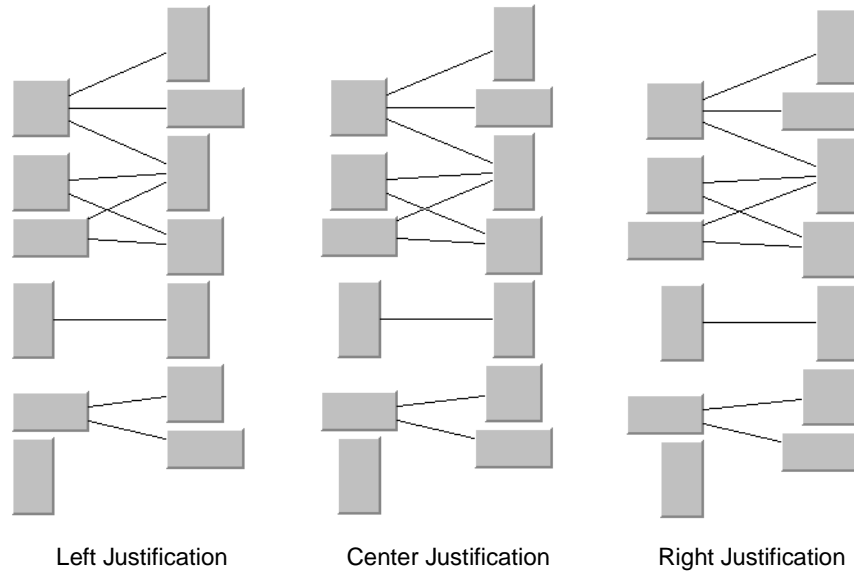


Figure 4.29 *Level Justification for Vertical Levels*

To obtain the current value, use the following method:

```
IlvDirection IlvHierarchicalLayout::getLevelJustification()
const;
```

Link Style

The layout algorithm positions the nodes and routes the links. To avoid overlapping nodes and links, it creates bend points for the shapes of links. The link style parameter controls the position and number of bend points. The link style can be set globally, in which case all links have the same kind of shape, or locally on each link such that different link shapes occur in the same drawing.

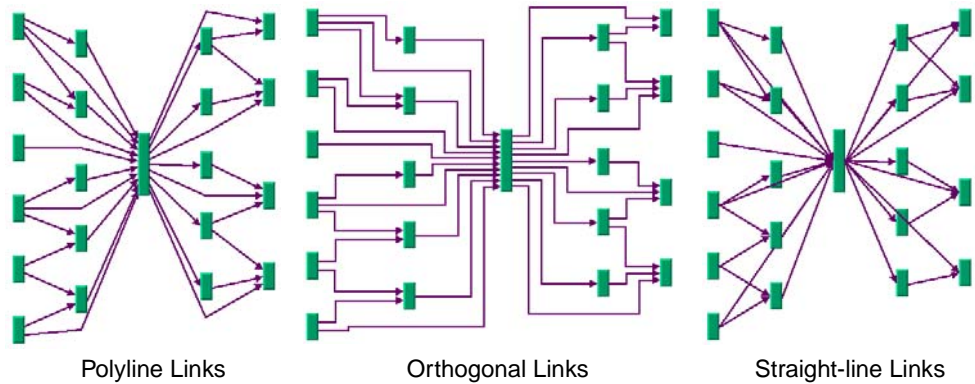


Figure 4.29 Link Styles

Note: The layout algorithm calls the method

`IlvGraphModel::ensureReshapeableLinks` on the attached graph model to ensure that all the links can be reshaped as needed. With an `IlvGrapher`, this method may replace links with the appropriate type of link and install the appropriate grapher pin on the nodes. For details on this method, see the Reference Manual. For details on the graph model, see the section *Using the Graph Model*.

Global Link Style

To set the global link style, use the following method:

```
void IlvHierarchicalLayout::setGlobalLinkStyle (IlvLayoutLinkStyle style);
```

The valid values for style are:

◆ `IlvLayoutPolylineLinkStyle`

All links get a polyline shape. A polyline shape consists of a sequence of line segments that are connected at bend points. The line segments can be turned into any direction. This is the default value.

◆ `IlvLayoutOrthogonalLinkStyle`

All links get an orthogonal shape. An orthogonal shape consists of orthogonal line segments that are connected at bend points. An orthogonal shape is a polyline shape where the segments can be turned only in directions of 0, 90, 180 or 270 degrees.

◆ `IlvLayoutStraightLineLinkStyle`

All links get a straight-line shape. All intermediate bend points (if any) are removed. This often causes overlapping nodes and links.

◆ `IlvLayoutNoReshapeLinkStyle`

None of the links is reshaped in any manner.

◆ `IlvLayoutMixedLinkStyle`

Each link can have a different link style. In this case, the style of each individual link can be set such that different link shapes can occur in the same graph.

To obtain the current choice, use the following method:

```
IlvLayoutLinkStyle IlvHierarchicalLayout::getGlobalLinkStyle()
const;
```

Individual Link Style

All links have the same style of shape unless the global link style is `MIXED_STYLE`. Only when the global link style is `MIXED_STYLE` can each link have an individual link style.

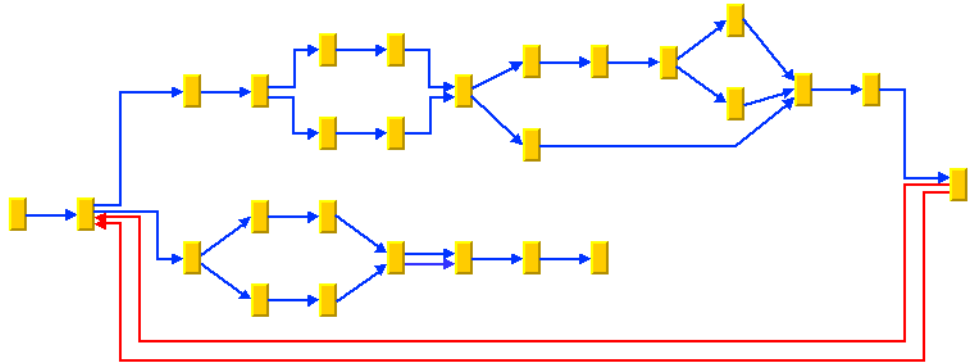


Figure 4.30 *Different Link Styles Mixed in the Same Drawing*

To set and retrieve the style of an individual line, use the following methods:

```
void IlvHierarchicalLayout::setLinkStyle(lvAny link,
IlvLayoutLinkStyle style);

IlvLayoutLinkStyle IlvHierarchicalLayout::getLinkStyle(lvAny
link) const;
```

The valid values for the local link style are the same as for the global link style:

- ◆ `IlvLayoutPolylineLinkStyle`
- ◆ `IlvLayoutOrthogonalLinkStyle`
- ◆ `IlvLayoutStraightLineLinkStyle`

◆ `IlvLayoutNoReshapeLinkStyle`

Note: The link style of the Hierarchical Layout requires links in an `IlvGrapher` that can be reshaped. Links of type `IlvLinkImage`, `IlvOneLinkImage`, `IlvDoubleLinkImage`, `IlvOneSplineLinkImage` and `IlvDoubleSplineLinkImage` cannot be reshaped. You can use the class `IlvPolylineLinkImage` instead.

Connector Style

The layout algorithm positions the end points of links (the connector pins) at the nodes automatically. The connector style parameter specifies how these end points are calculated.

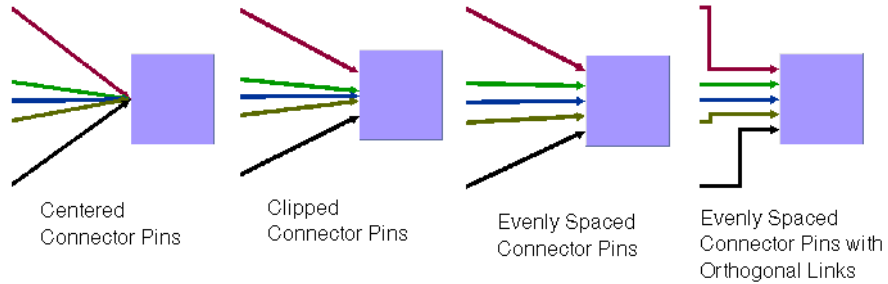


Figure 4.31 Connector Styles

To set the connector style, use the following method:

```
void  
IlvHierarchicalLayout::setConnectorStyle(IlvLayoutConnectorStyle  
style);
```

The following options are available for style:

◆ `IlvLayoutCenteredPins`

The end points of the links are placed in the center of the border where the links are attached. This option is well-suited for polyline links and straight-line links. It is less well-suited for orthogonal links, because orthogonal links can look ambiguous in this style.

◆ `IlvLayoutClippedPins`

Each link pointing to the center of the node is clipped at the node border. The connector pins are placed at the points on the border where the links are clipped. This option is particularly well-suited for polyline links.

◆ `IlvLayoutEvenlySpacedPins`

The connector pins are evenly distributed along the node border. This style guarantees that the end points of the links do not overlap. This is the best style for orthogonal links and works well for other link styles.

◆ `IlvLayoutAutomaticPins`

The connector style is selected automatically depending on the link style. If any of the links have an orthogonal style, the algorithm chooses evenly spaced connectors. If all the links are straight, it chooses centered connectors. Otherwise, it chooses clipped connectors.

To obtain the current choice, use the following method:

```
IlvLayoutConnectorStyle
IlvHierarchicalLayout::getConnectorStyle() const;
```

Note: *The connector style parameter requires grapher pins at the nodes of an `IlvGrapher` that allow connector pins to be placed freely at the node border. It is recommended that you install `IlvRelativeLinkConnector` at nodes that are contained in an `IlvGrapher`.*

Link Priority

The layout algorithm tries to place the nodes such that all links are short, point in the flow direction, and do not cross each other. However, this is not always possible. Often, links cannot have the same length. If the graph has cycles, some links must be reversed against the flow direction. If the graph is nonplanar, some links have to cross each other.

The link priority parameter controls which links should be selected if long, reversed, or crossing links are necessary. Links with a low priority are more likely to be selected than links with a high priority. This does not mean that low-priority links are always longer, reversed, or crossed, because the graph may have a structure such that no long, reversed or crossing links are necessary.

To set or retrieve the link priority, use the following methods.

```
void IlvHierarchicalLayout::setLinkPriority(IlAny link, IlDouble
priority);
IlFloat IlvHierarchicalLayout::getLinkPriority(IlAny link) const;
```

The default value of the link priority is 1.0. Negative link priorities are not allowed.

For an example of using the link priority, consider a cycle `A->B->C->D->E->A`. It is impossible to lay out this graph without reversing any link. Therefore, the layout algorithm selects one link to be reversed. To control which link is selected, you can give one link a lower priority than the others. This link will be reversed. In Figure 4.32, the bottom layout shows the use of the link priority. The link `C->D` was given the priority 0.5, while all the other links have the priority 1.0. Therefore `C-D` is reversed. The top layout in Figure 4.32 shows what happens when all links have the same priority. Link `E->A` is reversed.

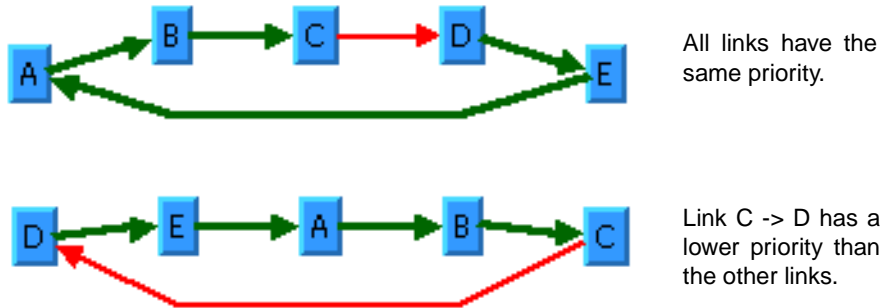


Figure 4.32 Working With Link Priorities

Level Index

In Step 1 of the layout algorithm (the leveling phase), the nodes are partitioned into levels. These levels are indexed starting from 0. For instance, when the flow direction is to the bottom, the nodes of the level with index 0 are placed at the topmost horizontal level line, and the nodes with larger level index are placed at a lower position than the nodes with smaller level index (see Figure 4.26). The layout algorithm calculates these level indices automatically.

You can affect how the levels are partitioned by specifying the level indices for some nodes. The nodes are placed in the specified level.

To specify the level index of a node, use the following method:

```
void IlvHierarchicalLayout::setSpecNodeLevelIndex(IlAny node,
IlInt index);
```

The default value is -1. If the default value is used or if a node is set to a negative level index, the layout algorithm automatically calculates an appropriate level index during the leveling phase of the algorithm.

To obtain the current level for a node, use the following method:

```
IlInt IlvHierarchicalLayout::getSpecNodeLevelIndex(IlAny node)
const;
```

Warning: Using arbitrarily large level indices is not recommended. For instance, if you set the level index of a node to 100000, the layout algorithm would create 100,000 levels even if the graph has much fewer nodes. This would cause the layout algorithm to become unnecessarily slow.

Figure 4.33 illustrates a hierarchical layout with the same level specified for each node of the graph.

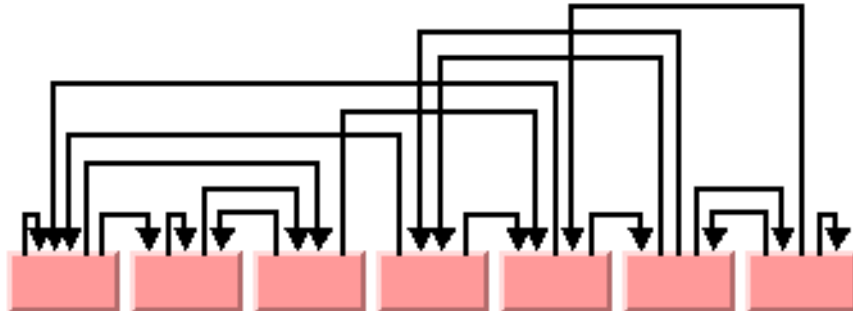


Figure 4.33 All Nodes Fixed at Same Level

Position Index

In Step 2 of the layout algorithm (the crossing reduction phase), the nodes are ordered within the levels. All nodes that belong to the same level get a position index starting from 0. For instance, when the flow direction is to the bottom, the node with the position index 0 is placed in the left-most position within its level. The nodes with a larger position index are placed more to the right than the nodes with a smaller position index in the same level. The nodes of different levels are independent. The node of the first level with the position index 0 is left of the node of the first level with the position index 1, but not necessarily left of a node of another level with position index 0. Note that long links crossing a level also obtain a position index (see Figure 4.26). The layout algorithm calculates these position indices automatically.

You can affect how the nodes are positioned within each level by specifying the position index of some nodes. The nodes are placed at the specified position within their level.

To specify the position index of a node, use the following method:

```
void IlvHierarchicalLayout::setSpecNodePositionIndex(IlAny node,
    IlInt index);
```

The default value is -1. If the default value is used, if a node is set to a negative position index, or if a node is set to a position index that is larger than the number of nodes of its level, the layout calculates automatically an appropriate position index during the crossing reduction step.

To obtain the current position index of a node, use the following method:

```
IlInt IlvHierarchicalLayout::getSpecNodePositionIndex(IlAny node)
const;
```

Spacing Parameters

The spacing of the layout is controlled by three kinds of spacing parameters: the minimal offset between nodes, the minimal offset between parallel segments of links and the minimal

offset between a node border and a bend point of a link or a link segment that is parallel to this border. The offset between parallel segments of links is at the same time the offset between bend points of links. All three kind of parameters occur in both directions: horizontally and vertically.

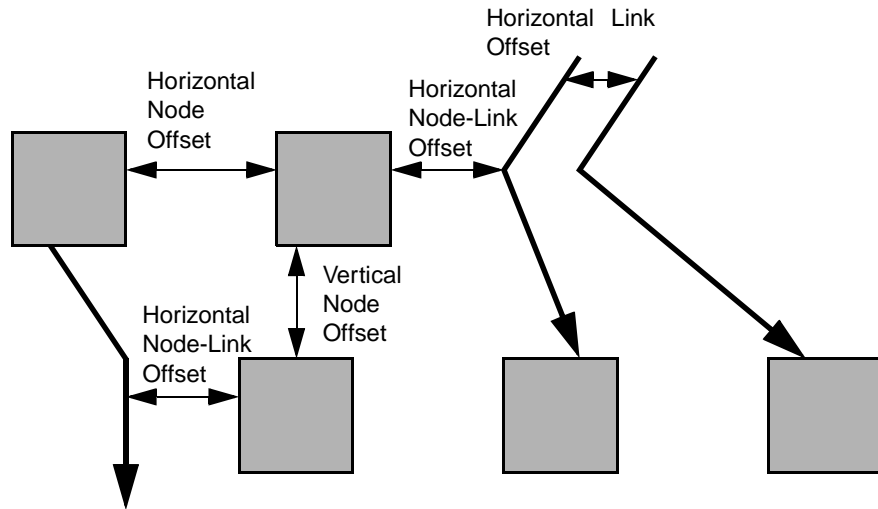


Figure 4.34 Spacing Parameters

The spacing parameters can be set for the horizontal direction by the following methods:

```
void IlvHierarchicalLayout::setHorizontalNodeOffset (IlvPos
offset);
void IlvHierarchicalLayout::setHorizontalLinkOffset (IlvPos
offset);
void IlvHierarchicalLayout::setHorizontalNodeLinkOffset (IlvPos
offset);
```

The spacing parameters can be set for the vertical direction by the following methods:

```
void IlvHierarchicalLayout::setVerticalNodeOffset (IlvPos offset);
void IlvHierarchicalLayout::setVerticalLinkOffset (IlvPos offset);
void IlvHierarchicalLayout::setVerticalNodeLinkOffset (IlvPos
offset);
```

The spacing parameters can be obtained by the corresponding methods:

```
IlvPos IlvHierarchicalLayout::getHorizontalNodeOffset () const;
IlvPos IlvHierarchicalLayout::getHorizontalLinkOffset () const;
IlvPos IlvHierarchicalLayout::getHorizontalNodeLinkOffset ()
const;
```

```

IlvPos IlvHierarchicalLayout::getVerticalNodeOffset() const;
IlvPos IlvHierarchicalLayout::getVerticalLinkOffset() const;
IlvPos IlvHierarchicalLayout::getVerticalNodeLinkOffset() const;

```

For a layout with horizontal levels (the flow direction is to the top or to the bottom), the horizontal node offset is the minimal distance between nodes of the same level. The vertical node offset is the minimal distance between nodes of different levels, that is, the minimal distance between the levels. For non-orthogonal link styles, the horizontal link offset is basically the minimal distance between bend points of links. The horizontal node-link offset is the minimal distance between the node border and the bend point of a link. For horizontal levels, the vertical link offset and the vertical node-link offset play a role only if the link shapes are orthogonal.

Similarly, for a layout with vertical levels (the flow direction is to the left or to the right), the vertical node offset controls node distances within the levels. The horizontal node offset is the minimal distance between the levels. In this case, the vertical link offset and the vertical node-link offset always play a role, while the horizontal link offset and the horizontal node-link offset affect the layout only with orthogonal links.

For orthogonal links, the horizontal link offset is the minimal distance between parallel, vertical link segments. The vertical link offset is the minimal distance between parallel, horizontal link segments. However, the layout algorithm cannot always satisfy these offset requirements. If a node is very small but has many incident links, it may be impossible to place the links orthogonally with the specified minimal link distance on the node border. In this case, the algorithm places some link segments closer than the specified link offset.

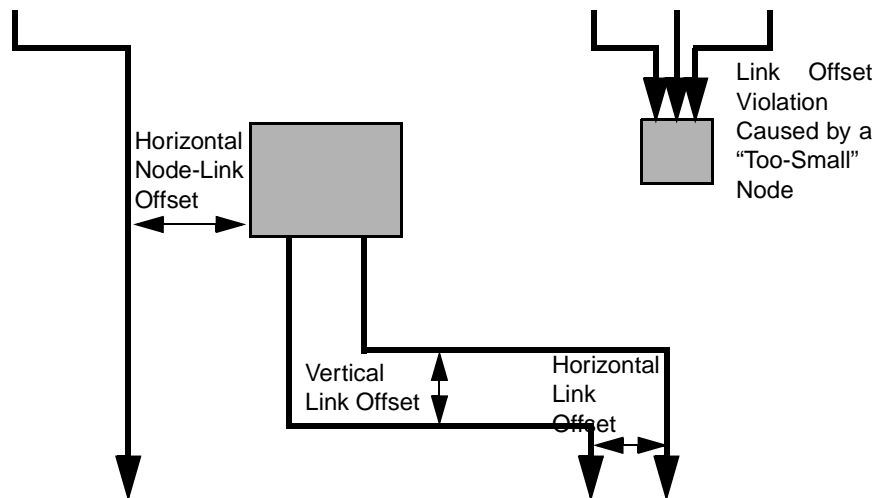


Figure 4.35 Spacing Parameters for Orthogonal Links

Sequences of Graph Layout

In some circumstances, you may need to use a sequence of layouts on the same graph. For example:

- ◆ You work with graphs that become out-of-date and need to extend the graph. If you perform a layout on the extended graph, you probably want to identify the parts that were already laid out in the original graph. The layout should not change very much when compared to the layout of the original graph.
- ◆ The first layout results in a drawing with minor deficiencies. You want to solve these deficiencies manually and perform a second layout to clean up the drawing. For the parts of the graph that were already acceptable after the first layout, the second layout probably should not change them very much.

The Hierarchical Layout supports sequences of layout that “do not change very much.” It allows you to preserve the level index of a node and the position index of the node within the level. By doing this, the relative position of a node compared to a previous layout does not change. The algorithm calculates new coordinates for the nodes and new routings for the links to adjust the absolute positions of the objects in order to clean up the drawing. The relative order such as “node A is in a level above node B” or “node A is left of node B” can be preserved.

The parameters for hierarchical sequences are described in:

- ◆ *Calculated Level Index*
- ◆ *Calculated Position Index*
- ◆ *Layout Refinements*

Calculated Level Index

The layout algorithm allows you to access the level index that was calculated for a node by a previous layout. To do this, use the following method:

```
IlInt IlvHierarchicalLayout::getCalcNodeLevelIndex(IlAny node)
const;
```

If the node was never laid out, this method returns -1. Otherwise, it returns the previous level index of the node.

The method can be used to specify the level index for the next layout in the following way:

```
IlInt index = layout->getCalcNodeLevelIndex(node);
layout->setSpecNodeLevelIndex(node, index);
```

When this is done, it ensures that the node is placed at the same level as in the previous layout.

If the graph is detached from the layout algorithm, the calculated level index of a node is set back to -1.

Note: You should be aware of the difference between the methods

`IlvHierarchicalLayout::getCalcNodeLevelIndex` and `IlvHierarchicalLayout::getSpecNodeLevelIndex`. The first one returns the level index calculated by the previous layout. The second one returns the specified level index, even if there was no previous layout. For instance, consider two nodes A and B. Node A has no specified level index and node B has a specified level index 5. Before the first layout, the method `IlvHierarchicalLayout::getCalcNodeLevelIndex` returns -1 for both nodes because the levels have not been calculated yet. However, `IlvHierarchicalLayout::getSpecNodeLevelIndex` returns -1 for A and 5 for B. After the first layout, node A may be placed at level 4. Now, `IlvHierarchicalLayout::getCalcNodeLevelIndex` returns 4 for node A and 5 for node B and `IlvHierarchicalLayout::getSpecNodeLevelIndex` still returns -1 for A and 5 for B.

Calculated Position Index

The layout algorithm allows you to access the position index within a level that was calculated for a node by a previous layout. To do this, use the following method:

```
IlInt IlvHierarchicalLayout::getCalcNodePositionIndex(IlAny node)
const
```

If the node was never laid out, this method returns -1. Otherwise, it returns the previous position index of the node within its level.

To ensure that the node is placed at the same level at the same relative position as in the previous layout, use the following:

```
layout->setSpecNodeLevelIndex(node,
                             layout->getCalcNodeLevelIndex(node));
layout->setSpecNodePositionIndex(node,
                                 layout->getCalcNodePositionIndex(node));
```

If the graph is detached from the layout algorithm, the calculated position index of a node is set back to -1.

Note: You should be aware of the difference between the methods

`IlvHierarchicalLayout::getCalcNodePositionIndex` and

`IlvHierarchicalLayout::getSpecNodePositionIndex`. The first one returns the position index calculated by the previous layout and -1 if there was no previous layout. The second one returns the specified position index, even if there was no previous layout. This behavior is similar to the behavior of the specified and calculated level index (see the section *Calculated Level Index*).

Layout Refinements

The following example shows how to use the calculated and specified level and position indices to refine a layout. Assume that you want to place a leaf node in the level with the highest index (see Figure 4.36). Since you do not know before laying out the graph how many levels will be created, you cannot specify the highest level index at the beginning. You must perform a first layout, iterate over the nodes, and retrieve the highest calculated level. If the leaf node is not already placed at the highest level index, you must specify this level

index for the leaf node while preserving the level and position indices of all other nodes. Then perform the layout a second time. The following code fragment shows how to do this.

```
// ...
IlvHierarchicalLayout* layout = new IlvHierarchicalLayout();
layout->attach(grapher);

// perform a first layout
IlvGraphLayoutReport* layoutReport = layout->performLayout();

// ...

// calculate the maximal level index, and preserve the relative node positions
// for the next layout

IlInt maxLevelIndex = -1;
IlUInt count;
IlvGraphic* const* allObjects = grapher->getObjects(count);
for (i = 0 ; i < count ; ++i) {
    IlvGraphic* node = allObjects[i];
    if (grapher->isNode(node)) {
        IlInt levelIndex = layout->getCalcNodeLevelIndex(node);
        IlInt positionIndex = layout->getCalcNodePositionIndex(node);
        if (maxLevelIndex < levelIndex)
            maxLevelIndex = levelIndex;
        layout->setSpecNodeLevelIndex(node, levelIndex);
        layout->setSpecNodePositionIndex(node, positionIndex);
    }
}

// if the leaf node was not at maximal level index, specify the maximal
// level index for the leaf node and perform layout again

if (layout->getCalcNodeLevelIndex(leafnode) != maxLevelIndex) {
    layout->setSpecNodeLevelIndex(leafnode, maxLevelIndex);
    layout->setSpecNodePositionIndex(leafnode, -1);
    layoutReport = layout->performLayout();
    // ...
}

layout->detach();
delete layout;
```

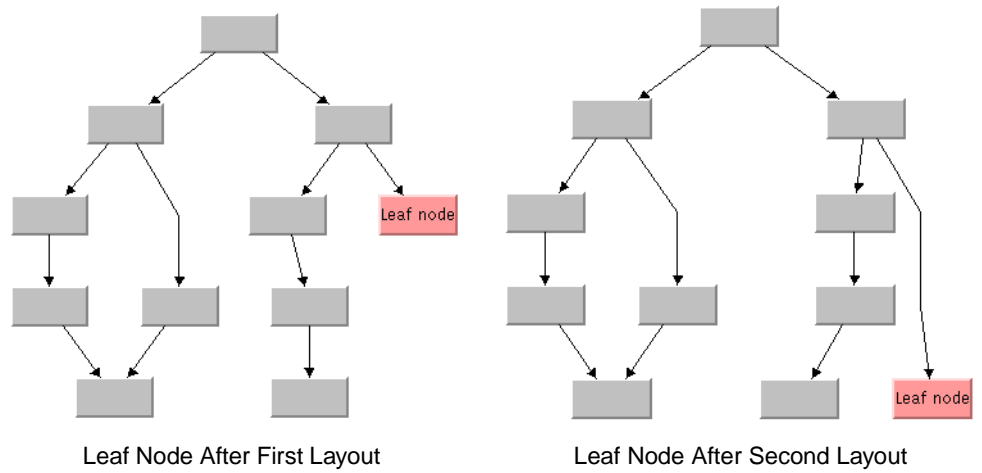



Figure 4.36 Moving the Leaf Node to the Lowest Level (Highest Level Index)

Orthogonal Link Layout

In this section, you will learn about the *Orthogonal Link Layout* algorithm from the IBM ILOG Views Graph Layout package (class `IlvOrthogonalLinkLayout` from the library `ilvorthlink`).

Samples

Here are sample drawings produced with the Orthogonal Link Layout:

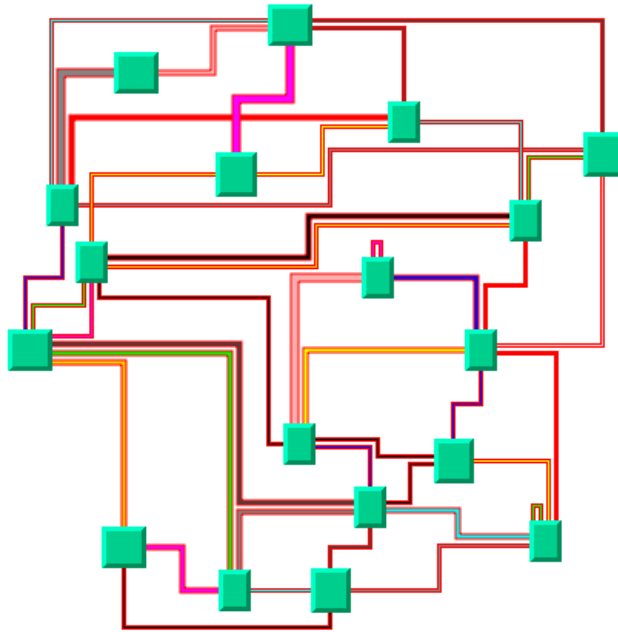


Figure 4.37 Orthogonal Link Drawing Produced with the Orthogonal Link Option

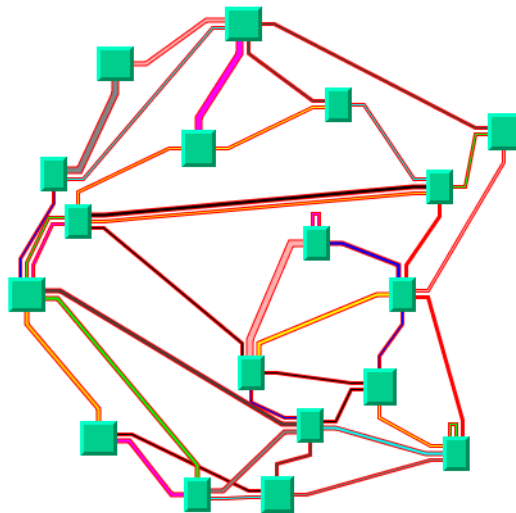


Figure 4.38 Orthogonal Link Drawing Produced with the Direct Links Option

What Types of Graphs?

Any type of graph:

- ◆ Connected and disconnected
- ◆ Planar and nonplanar graphs

Application Domains

Application domains of the Orthogonal Layout include:

- ◆ Electrical engineering (circuit block diagrams)
- ◆ Industrial engineering (schematic design diagrams, equipment/resource control charts)
- ◆ Business processing (entity relation diagrams)
- ◆ Software management/software (re-)engineering (data inspector diagrams)
- ◆ Database and knowledge engineering (sociology, genealogy)
- ◆ CASE tools (design diagrams)

Features

- ◆ Reshapes the links of a graph in either an “orthogonal” or a “direct” style, without moving the nodes.
- ◆ Efficient, scalable algorithm.
- ◆ The shapes of the links are computed in such a way as to reduce the number of link-to-link and link-to-node crossings.
- ◆ Supports links with different widths.
- ◆ Automatically arranges the final segments of the links (the segments near the origin or destination node) to obtain a “bundle” of parallel links.
- ◆ Allows you to specify which side of the node (top, bottom, left, or right) a link can be connected to.
- ◆ Supports self-links (that is, links with the same origin and destination node).
- ◆ Supports multiple links (that is, more than one link between the same origin and destination nodes).
- ◆ Allows you to specify “pinned” (fixed) links that the layout algorithm cannot reshape.

Limitations

- ◆ Link-to-link and link-to-node crossings cannot always be avoided, especially in highly connected graphs with links between distant nodes. This limitation is closely related to the efficiency issues of this intrinsically complex layout problem.

Brief Description of the Algorithm

The Orthogonal Link Layout algorithm is based on a combinatorial optimization that chooses the “optimal” shape of the links in order to minimize a cost function. This cost function is proportional to the number of link-to-link and link-to-node crossings.

For efficiency reasons, the basic shape of each link is chosen from a set of predefined shapes. These shapes are different for each link-style option. See the orthogonal link style in Figure 4.37 and the direct link style in Figure 4.38.

The shape of a link also depends on the relative position of the origin and destination nodes. For instance, when two nodes are very close or they overlap, the shape of the link is chosen to provide the best visibility of the link.

The exact shape of a link is computed taking into account additional constraints. The layout algorithm tries to:

- ◆ Minimize the number of crossings between the links incident to a given side of a node.
- ◆ Space the final segments of the links incident to a given side of a node equally on the node, no matter what their width.

Code Sample

Below is a code sample using the `IlvOrthogonalLinkLayout` class:

```
// ...
IlvGrapher* grapher = new IlvGrapher(display);

// ... Fill in the grapher with nodes and links here

IlvOrthogonalLinkLayout* layout = new IlvOrthogonalLinkLayout();
layout->attach(grapher);

// Set the layout parameters, e.g., the offset between links:
layout->setLinkOffset(5);

// Perform the layout
IlvGraphLayoutReport* layoutReport = layout->performLayout();
if (layoutReport->getCode() != IlvLayoutReportLayoutDone)
    IlvWarning("Layout not done. Error code = %d\n", layoutReport->getCode());
// ...
// If this grapher is not anymore subject of layout:
layout->detach();

// Once the layout algorithm is not anymore needed:
delete layout;
```

Parameters

The Orthogonal Link Layout uses generic parameters, common to other graph layouts, and specific parameters applicable in orthogonal link layouts only. Refer to the following sections for general information on parameters among the graph layouts:

- ◆ *Generic Parameters Support*
- ◆ *Layout Characteristics*

The Orthogonal Link Layout parameters are described in detail in this topic under:

- ◆ *Generic Parameters*
- ◆ *Specific Parameters*

Generic Parameters

The `IlvOrthogonalLinkLayout` class supports the following generic parameters as defined in the class `IlvGraphLayout`:

- ◆ *Allowed Time*
- ◆ *Preserve Fixed Links*

The following comments describe the particular way in which these parameters are used by this subclass.

Allowed Time

The layout algorithm stops if the allowed time setting has elapsed. (For a description of this layout parameter in the `IlvGraphLayout` class, see [Allowed Time](#).)

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (For more information on link parameters in the `IlvGraphLayout` class, see [Preserve Fixed Links](#).)

Specific Parameters

The following parameters are specific to the `IlvOrthogonalLinkLayout` class:

- ◆ *Dimensional Parameters*
- ◆ *Link Style*
- ◆ *Number of Optimization Iterations*
- ◆ *Same Shape for Multiple Links*
- ◆ *Link Crossing Penalty*

Dimensional Parameters

Figure 4.39 illustrates the dimensional parameters used in the Orthogonal Link Layout algorithm. These parameters are:

- ◆ *Link Offset*
- ◆ *Minimum Final Segment Length*

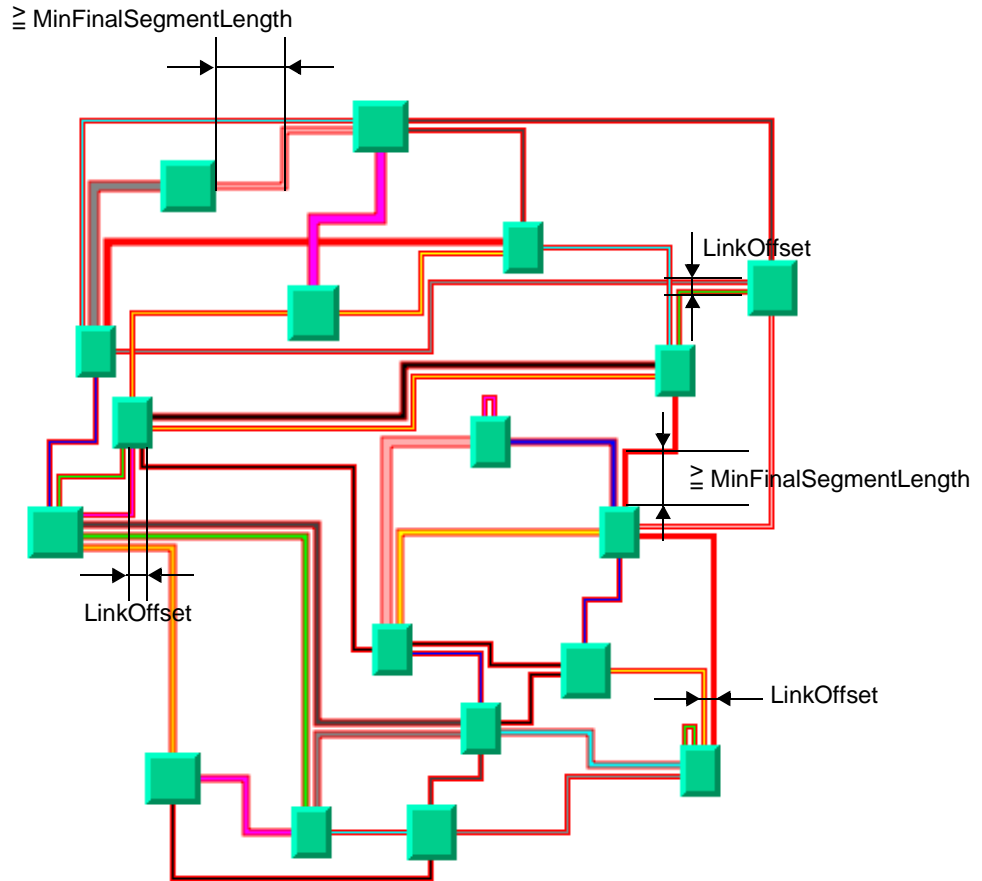


Figure 4.39 Dimensional Parameters for the Orthogonal Link Layout Algorithm

Link Offset

The layout algorithm computes the final connecting segments of the links (that is, the segments near the origin and destination nodes) in order to obtain parallel lines spaced at a user-defined distance. Since the links can have different widths, it takes into account the width of the links, when computing the offset. To specify the offset, use the method:

```
void IlvOrthogonalLinkLayout::setLinkOffset(IlUInt offset);
```

To obtain the current value, use the method:

```
IlUInt IlvOrthogonalLinkLayout::getLinkOffset() const;
```

Minimum Final Segment Length

You can specify a minimum value for the length of the final connecting segments of the links (that is, the segments near the origin and destination nodes) using the method:

```
void IlvOrthogonalLinkLayout::setMinFinalSegmentLength(IIUInt
length);
```

To obtain the current value, use the method:

```
IIUInt IlvOrthogonalLinkLayout::getMinFinalSegmentLength() const;
```

The default value is 10.

Link Style

The layout algorithm provides two link styles. To specify the link style, the following method can be used:

```
void IlvOrthogonalLinkLayout::setLinkStyle(IIvLayoutLinkStyle
style);
```

The valid values for `style` are:

- ◆ `IIvLayoutOrthogonalLinkStyle`

The links are reshaped to a polygonal line of alternating horizontal and vertical segments. Figure 4.37 shows a layout produced with this style. This is the default.

- ◆ `IIvLayoutDirectLinkStyle`

The links are reshaped to a polygonal line composed of three segments: a straight-line segment that starts and ends with a small horizontal or vertical segment. Figure 4.38 shows a layout produced with this style.

Note: *The layout algorithm calls the method*

`IIvGraphModel::ensureReshapeableLinks` on the attached graph model to ensure that all the links can be reshaped as needed, including their connection points. With an `IIvGrapher`, this method may replace links with the appropriate type of link and install the appropriate link connector on the nodes. For details on this method, see the Reference Manual. For details on the graph model, see *Using the Graph Model*.

Note: *The link style requires links in an `IIvGrapher` that can be reshaped. Links of type*

`IIvLinkImage`, `IIvOneLinkImage`, `IIvDoubleLinkImage`, `IIvOneSplineLinkImage`, and `IIvDoubleSplineLinkImage` cannot be reshaped. You can use the class `IIvPolylineLinkImage` instead.

To obtain the current choice, use the following method:

```
IIvLayoutLinkStyle IlvOrthogonalLinkLayout::getLinkStyle() const;
```

Number of Optimization Iterations

The link shape optimization is stopped if the number of iterations exceeds the allowed number of iterations or the time exceeds the allowed time. To specify this number, use the method:


```
void IlvOrthogonalLinkLayout::setAllowedNumberOfIterations (IlUInt
iterations);
```

To obtain the current value, use the method:

```
IlUInt IlvOrthogonalLinkLayout::getAllowedNumberOfIterations ()
const
```

Note: You may want to disable the link shape optimization by setting the number of iterations to zero in order to increase the speed of the layout process.

Same Shape for Multiple Links

You can force the layout algorithm to compute the same shape for all the links having common origin and destination nodes. The links will have parallel shapes. To enable or disable this option, use the method:

```
void
IlvOrthogonalLinkLayout::setSameShapeForMultipleLinks (IlBoolean
option);
```

To obtain the current value, use the method:

```
IlBoolean IlvOrthogonalLinkLayout::isSameShapeForMultipleLinks ()
const;
```

The default value is `IlFalse`.

Link Crossing Penalty

The computation of the shape of the links is driven by the objective to minimize a cost function, which is proportional to the number of link-to-link and link-to-node crossings. By default, these two types of crossings have equal weights. You can increase or decrease the weight of the **link-to-node crossings** using the method:

```
void IlvOrthogonalLinkLayout::setLinkToNodeCrossingPenalty (IlUInt
penalty);
```

The default value is 1. To increase the possibility of obtaining a layout with no link-to-node crossings (or with only a few crossings), set the parameter to a value greater than one. For example, set penalty to 5.

To obtain the current value, use the method:

```
IlUInt IlvOrthogonalLinkLayout::getLinkToNodeCrossingPenalty ()
const;
```

You can also increase or decrease the weight of the **link-to-link crossings** using the method:

```
void IlvOrthogonalLinkLayout::setLinkToLinkCrossingPenalty (IlUInt
penalty);
```

The default value is 1. To increase the possibility of obtaining a layout with no link-to-link crossings (or with a only few crossings), set the parameter to a value greater than one. For example, set `penalty` to 5.

To obtain the current value, use the method:

```
IlUInt IlvOrthogonalLinkLayout::getLinkToLinkCrossingPenalty()  
const;
```

Random Layout

In this section, you will learn about the *Random Layout* algorithm from the IBM® ILOG® Views Graph Layout package (class `IlvRandomLayout` from the library `ilvrandom`).

Sample

Here is a sample drawing produced with the Random Layout:

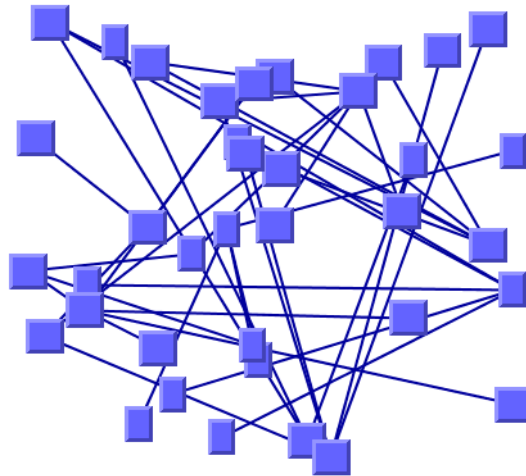


Figure 4.40 Graph Drawing Produced with the Random Layout

What Types of Graphs?

Any type of graph:

- ◆ Connected and disconnected
- ◆ Planar and nonplanar graphs

Features

Random placement of the nodes of a grapher inside a given region.

Limitations

- ◆ The algorithm computes random coordinates for the upper-left corner of the graphic objects representing the nodes. In some cases, this may not be appropriate.
 - ◆ To ensure that the nodes do not overlap the margins of the layout region, the algorithm computes the coordinates randomly inside a region whose width and height are smaller than the width and height of the layout region. The difference is the maximum width and the maximum height of the nodes, respectively. In some cases, this may not be appropriate.
-

Brief Description of the Algorithm

The Random Layout algorithm is not really a layout algorithm. It simply places the nodes at randomly computed positions inside a user-defined region. Nevertheless, the Random Layout algorithm may be useful when a random, initial placement is needed by another layout algorithm or in cases where an attractive, readable drawing is not important.

Code Sample

Below is a code sample using the `IlvRandomLayout` class:

```
// ...
IlvGrapher* grapher = new IlvGrapher(display);

// ... Fill in here the grapher with nodes and links in

IlvRandomLayout* layout = new IlvRandomLayout();
layout->attach(grapher);

// Perform the layout
IlvGraphLayoutReport* layoutReport = layout->performLayout();
if (layoutReport->getCode() != IlvLayoutReportLayoutDone)
    IlvWarning("Layout not done. Error code = %d\n", layoutReport->getCode());

// ...
// If the grapher is not anymore subject of layout:
layout->detach();

// Once the layout algorithm is not anymore needed:
delete layout;
```

Parameters

The Random Layout uses generic parameters, common to other graph layouts, and specific parameters applicable in random layouts only. Refer to the following sections for general information on parameters among the graph layouts:

- ◆ *Generic Parameters Support*
- ◆ *Layout Characteristics*

The Random Layout parameters are described in detail in this topic under:

- ◆ *Generic Parameters*
- ◆ *Specific Parameters*

Generic Parameters

The `IlvRandomLayout` class supports generic parameters defined in the `IlvGraphLayout` class. The following comments describe the particular way in which these parameters are used by this subclass:

- ◆ *Layout Region*
- ◆ *Preserve Fixed Links*
- ◆ *Preserve Fixed Nodes*
- ◆ *Random Generator Seed Value*

Layout Region

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways to specify the layout region are available for this subclass. (For a description of this parameter in the `IlvGraphLayout` class, see [Layout Region](#).)

Remember that if you are using the default settings, there must be a view (an instance of `IlvView`) attached to the grapher. If no view is attached, and no layout region is explicitly specified, the layout algorithm cannot produce reasonable results.

If a layout region is specified, this region is used. If no layout region is specified but a view is attached, the size and position of the graph drawing is computed to approximately fill the view.

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (For more information on link parameters in the `IlvGraphLayout` class, see [Preserve Fixed Links](#).)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (For more information on node parameters in the `IlvGraphLayout` class, see Preserve Fixed Nodes.)

Random Generator Seed Value

The Random Layout uses a random number generator to compute the coordinates. You can specify a particular value to be used as a seed value. (For more information on this parameter in the `IlvGraphLayout` class, see Random Generator Seed Value.) For the default behavior, the random generator is initialized using the current system clock. Therefore, different layouts are obtained if you perform the layout repeatedly on the same graph.

Specific Parameters

Information on the Link Style parameter specific to the `IlvRandomLayout` class is as follows.

Link Style

When the layout algorithm moves the nodes, straight-line links, such as instances of `IlvLinkImage`, will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout algorithm to automatically remove all the intermediate points of the links (if any). To do this, the following method is provided:

```
void IlvRandomLayout::setLinkStyle(IlvLayoutLinkStyle style);
```

The valid values are:

◆ `IlvLayoutNoReshapeLinkStyle`

None of the links is reshaped in any manner.

◆ `IlvLayoutStraightLineLinkStyle`

All the intermediate points of the links (if any) are removed. This is the default value.

Note: If the `IlvLayoutStraightLineLinkStyle` style is selected, the layout algorithm calls the method `IlvGraphModel::ensureStraightLineLinks()` on the attached graph model to ensure that all the links can have a straight-line shape. With an `IlvGrapher`, this method may replace links with new type of links. For details on this method, see the Reference Manual. For details on the graph model, see the section Using the Graph Model.

To obtain the current choice, use the following method:

```
IlvLayoutLinkStyle IlvRandomLayout::getLinkStyle() const;
```

Bus Layout

In this section, you will learn about the *Bus Layout* algorithm from the ILOG Views Graph Layout package (class `IlvBusLayout` from the library `ilvbus`).

Sample

Here is a sample drawing produced with the Bus Layout:

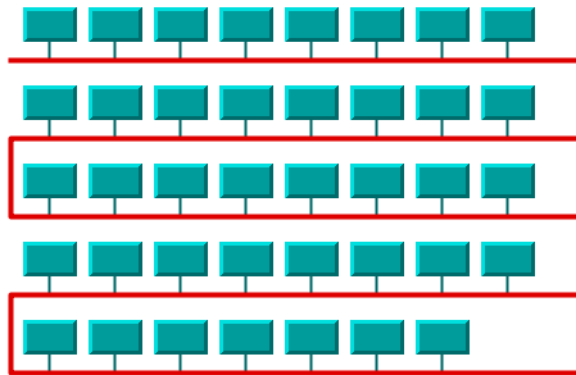


Figure 4.41 Bus Topology Produced with the Bus Layout using the No Ordering Option

What Types of Graphs?

- ◆ Bus network topologies (a set of nodes connected to a bus object)

Application Domains

Application domains of the Bus Layout include:

- ◆ Telecom and networking (LAN diagrams)
- ◆ Electrical engineering (circuit block diagrams)
- ◆ Industrial engineering (equipment/resource control charts)

Features

- ◆ Displays bus topologies.

- ◆ Takes into account the size of the nodes so that no overlapping occurs.
- ◆ Provides several ordering options. The nodes can be arranged on the bus by height, in a user-defined order, or in an arbitrary order.
- ◆ Allows easy customization of the dimensional parameters.

Brief Description of the Algorithm

Bus topology is well-known in network management and telecommunications fields. The Bus Layout class can display these topologies nicely. It represents the “bus” as a “serpent” polyline. The width of the “serpent” is user-defined (via the width of the layout region parameter) and the height is computed so that enough space is available for all the nodes.

Code Sample

Below is a code sample using the `IlvBusLayout` class:

```
// ...
IlvGrapher* grapher = new IlvGrapher(display);

// ... Fill in the grapher with nodes and links here

// Create the bus node; the number of points and
// the coordinates are not important
IlvPoint point(10,10);
IlvPolyline* bus = new IlvPolyline(display, 1, &point);
grapher->addNode(bus);

// ... Fill in the grapher with links between each node
// and the bus here

IlvBusLayout* layout = new IlvBusLayout();
layout->attach(grapher);

// Specify the bus node
layout->setBus(bus);

IlvGraphLayoutReport* layoutReport = layout->performLayout();
if (layoutReport->getCode() != IlvLayoutReportLayoutDone)
    IlvWarning("Layout not done. Error code = %d\n", layoutReport->getCode());
// ...
// If the grapher is not anymore subject of layout:
layout->detach();

// Once the layout algorithm is not anymore needed:
delete layout;
```

Parameters

The Bus Layout uses generic parameters, common to other graph layouts, and specific parameters applicable in bus layouts only. Refer to the following sections for general information on parameters among the graph layouts:

- ◆ *Generic Parameters Support*
- ◆ *Layout Characteristics*

The Bus Layout parameters are described in detail in this topic under:

- ◆ *Generic Parameters*
- ◆ *Specific Parameters*

Generic Parameters

The `IlvBusLayout` class supports generic parameters defined in the `IlvGraphLayout` class. The following comments describe the particular way in which these parameters are used by this subclass:

- ◆ *Layout Region*
- ◆ *Preserve Fixed Links*
- ◆ *Preserve Fixed Nodes*

Layout Region

The layout algorithm uses the layout region setting (either your own or the default setting) to control the size and the position of the graph drawing. All three ways of specifying the layout region are available for this subclass. (For a description of this parameter in the `IlvGraphLayout` class, see [Layout Region](#).)

Remember that if you are using the default settings, there must be a view (an instance of `IlvView`) attached to the grapher. If no view is attached, and no layout region is explicitly specified, the layout algorithm cannot produce reasonable results.

If a layout region is specified, this region is used. If no layout region is specified but a view is attached, the size and position of the graph drawing is computed to approximately fill the view.

The size of the layout is chosen with respect to the layout region (see [Figure 4.42](#)). The height of the layout region is not taken into account. The height of the layout will be smaller or larger, depending on the number of nodes, the size of the nodes, and the other specified parameters.

Preserve Fixed Links

The layout algorithm does not reshape the links that are specified as fixed. (For more information on link parameters in the `IlvGraphLayout` class, see Preserve Fixed Links.)

Preserve Fixed Nodes

The layout algorithm does not move the nodes that are specified as fixed. (For more information on node parameters in the `IlvGraphLayout` class, see Preserve Fixed Nodes.)

Specific Parameters

The following parameters are specific to the `IlvBusLayout` class:

- ◆ *Order*
- ◆ *Bus Node*
- ◆ *Link Style*
- ◆ *Dimensional Parameters*

Order

The order parameter specifies how to arrange the nodes.

To specify the ordering option for the nodes, use the method:

```
void IlvBusLayout::setOrdering(IlvBusOrder ordering);
```

The valid values are:

- ◆ `IlvBusNoOrdering`

The nodes are arranged on the bus in an arbitrary order. This is the default value.

- ◆ `IlvBusOrderByHeight`

The nodes are arranged on the bus according to height, starting at the upper-left corner of the bus with the tallest node. This option can save vertical space.

- ◆ `IlvBusOrderByIndex`

This option allows you to specify a particular order for the nodes. The nodes are arranged on the bus according to their index values, starting at the upper-left corner of the bus with the node with the smallest index. Nodes for which you do not specify an index are placed after the nodes for which an index is specified.

The `IlvBusOrderByIndex` option allows you to specify the order of the nodes, according to physical, geographical data for example. If this option is chosen, the algorithm sorts the nodes in ascending order according to their index values.

To obtain the current value, use the method:

```
IlvBusOrder IlvBusLayout::getOrdering() const;
```

The index is an integer value associated with a node and is specified using the method:

```
void IlvBusLayout::setIndex(IlAny node, IlInt index);
```

The values of the indexes cannot be negative. To obtain the current index of a node, use the method:

```
IlInt IlvBusLayout::getIndex(IlAny node) const;
```

If no index is specified for the node, the value `NoIndex` is returned.

The ordering options for the Bus Layout are illustrated in Table 4.5.

Table 4.5 Ordering Options of the Nodes for the Bus Layout Algorithm

Ordering	Layout
IlvBusNoOrdering	

Table 4.5 Ordering Options of the Nodes for the Bus Layout Algorithm

Ordering	Layout
IlvBusOrderByHeight	
IlvBusOrderByIndex	

Bus Node

To represent bus topologies, the algorithm reshapes a special node, called the “bus node”, and gives it a “serpent” form. This bus node must be an instance of the `IlvPolyline` class. Before performing the layout, you must create this object and add it to the grapher as a node. (The number of points in the object you create is not important.) Then, you must specify the node as “bus node” using the method:

```
void IlvBusLayout::setBus(IlvPolyline* bus);
```

To obtain the current selection for the bus node, use the method:

```
IlvPolyline* IlvBusLayout::getBus() const;
```

You can also use subclasses of `IlvPolyline`. (See the code provided in Code Sample.)

Link Style

When the layout algorithm moves the nodes, straight-line links, such as instances of `IlvLinkImage`, will automatically “follow” the new positions of their end nodes. If the grapher contains other types of links (for example, `IlvPolylineLinkImage` or `IlvDoubleSplineLinkImage`), the shape of the link may not be appropriate because the intermediate points of the link will not be moved. In this case, you can ask the layout

algorithm to automatically remove all the intermediate points of the links (if any). To do this, the following method is provided:

```
void IlvBusLayout::setLinkStyle(IlvLayoutLinkStyle style);
```

The valid values are:

- ◆ `IlvLayoutNoReshapeLinkStyle`
None of the links is reshaped in any manner.
- ◆ `IlvLayoutStraightLineLinkStyle`
All the intermediate points of the links (if any) are removed. This is the default value.

Note: If the `IlvLayoutStraightLineLinkStyle` style is selected, the layout algorithm calls the method `IlvGraphModel::ensureStraightLineLinks` on the attached graph model to ensure that all the links have a straight-line shape. With an `IlvGrapher`, this method may replace links with new types of links. For details on this method, see the *Reference Manual*. For details on the graph model, see the section *Using the Graph Model*.

To obtain the current choice, use the following method:

```
IlvLayoutLinkStyle IlvBusLayout::getLinkStyle() const;
```

Dimensional Parameters

Figure 4.42 illustrates the dimensional parameters used in the Bus Layout algorithm. These parameters are:

- ◆ *Horizontal Offset*
- ◆ *Vertical Offset to Level*
- ◆ *Vertical Offset to Previous Level*
- ◆ *Margin*
- ◆ *Margin on Bus*

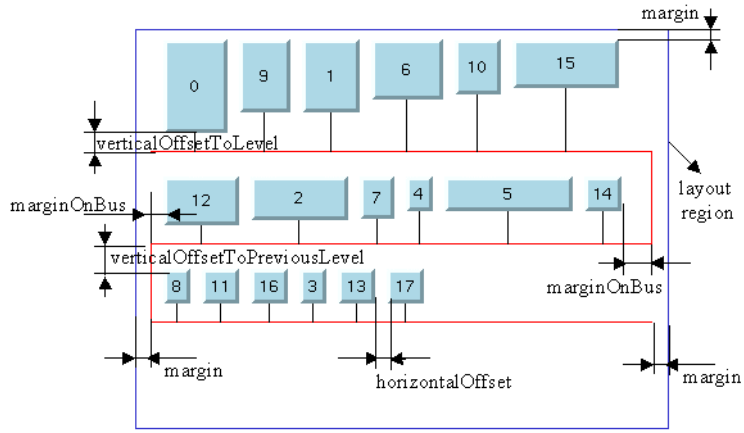


Figure 4.42 Dimensional Parameters for the Bus Layout Algorithm

Horizontal Offset

This parameter represents the horizontal distance between two nodes. To specify the horizontal offset, use the method:

```
void IlvBusLayout::setHorizontalOffset(IlUInt offset);
```

To obtain the current value, use the method:

```
IlUInt IlvBusLayout::getHorizontalOffset() const;
```

Vertical Offset to Level

This parameter represents the vertical distance between a row of nodes and the next horizontal segment of the bus node. To specify this parameter, use the method:

```
void IlvBusLayout::setVerticalOffsetToLevel(IlUInt offset);
```

To obtain the current value, use the method:

```
IlUInt IlvBusLayout::getVerticalOffsetToLevel() const;
```

Vertical Offset to Previous Level

This parameter represents the vertical distance between a row of nodes and the previous horizontal segment of the bus node. To specify this parameter, use the method:

```
void IlvBusLayout::setVerticalOffsetToPreviousLevel(IlUInt offset);
```

To obtain the current value, use the method:

```
IlUInt IlvBusLayout::getVerticalOffsetToPreviousLevel() const;
```

Margin

This parameter represents the offset distance between the layout region and the bounding rectangle of the layout. To specify this parameter, use the method:

```
void IlvBusLayout::setMargin(IlUInt margin);
```

To obtain the current value, use the method:

```
IlUInt IlvBusLayout::getMargin() const;
```

Margin on Bus

On the odd horizontal levels (first, third, fifth, and so on) of the bus starting from the top, this parameter represents the offset distance between the left side of the first node on the left and the left side of the bus object.

On the even horizontal levels (second, fourth, sixth, and so on) of the bus starting from the top, this parameter represents the offset distance between the right side of the last node on the right and the right side of the bus object. (See Figure 4.42 for an illustration of the margin-on-bus parameter.)

To specify this parameter, use the method:

```
void IlvBusLayout::setMarginOnBus(IlUInt margin);
```

To obtain the current value, use the method:

```
IlUInt IlvBusLayout::getMarginOnBus() const;
```


Using Advanced Features

This chapter describes advanced features for using the Graph Layout package of IBM® ILOG® Views. The following topics are covered:

- ◆ *Using a Layout Report*
- ◆ *Using Layout Event Listeners*
- ◆ *Using the Graph Model*
- ◆ *Laying Out a Non-Views Grapher*
- ◆ *Using the Filtering Features to Lay Out a Part of an IlvGrapher*
- ◆ *Laying Out Graphs with Nonzoomable Graphic Objects*
- ◆ *Defining a New Type of Layout*
- ◆ *Questions and Answers about Using the Layout Algorithms*

Using a Layout Report

Layout reports are objects used to store information about the particular behavior of a layout algorithm. After the layout is completed, this information is available to be read from the layout report.

Layout Report Classes

Each layout class instantiates a particular class of `IlvGraphLayoutReport` each time the layout is performed. Table 5.1 shows the layout classes and their corresponding layout reports.

Table 5.1 *Layout Report Classes*

Layout Class	Layout Report Class
<code>IlvTreeLayout</code>	<code>IlvGraphLayoutReport</code>
<code>IlvHierarchicalLayout</code>	<code>IlvGraphLayoutReport</code>
<code>IlvOrthogonalLinkLayout</code>	<code>IlvGraphLayoutReport</code>
<code>IlvRandomLayout</code>	<code>IlvGraphLayoutReport</code>
<code>IlvBusLayout</code>	<code>IlvGraphLayoutReport</code>

Creating a Layout Report

All layout classes inherit the `IlvGraphLayout::performLayout` method from the `IlvGraphLayout` class. This method calls `IlvGraphLayout::createLayoutReport` to obtain a new instance of the layout report. This instance is returned when `IlvGraphLayout::performLayout` returns. The default implementation in the base layout class creates an instance of `IlvGraphLayoutReport`. Some subclasses override this method to return an appropriate subclass. Other classes, such as `IlvRandomLayout`, do not need specific information to be stored in the layout report and do not override `IlvGraphLayout::createLayoutReport`. In this case, the base class `IlvGraphLayoutReport` is used.

When using the layout classes provided with IBM ILOG Views, you do not need to instantiate the layout report yourself. This is done automatically.

The instantiation is made by internal methods and is managed by the class. Thus, you do not need to delete the instance returned by the `IlvGraphLayout::performLayout` method.

Reading a Layout Report

To read a layout report, all you need to do is store the layout report instance returned by the `IlvGraphLayout::performLayout` method and read the information, as shown in the following example for the Spring Embedder Layout:

```
IlvGraphLayoutReport* layoutReport = layout->performLayout();
if (layoutReport->getCode() == IlvLayoutReportLayoutDone)
    IlvPrint("Layout done.");
else IlvWarning("Layout not done. Error code = %d\n",
               layoutReport->getCode());
```

Information Stored in a Layout Report

The base class `IlvGraphLayoutReport` stores the following information:

◆ Code

This field contains information about special, predefined cases that may have occurred during the layout. The possible values are the following:

- `IlvLayoutReportLayoutDone` appears if the layout was performed successfully.
- `IlvLayoutReportNotNeeded` appears if the layout was not performed because no changes occurred in the grapher and parameters since the last time the layout was performed successfully.
- `IlvLayoutReportEmptyGrapher` appears if the grapher is empty.
- `IlvLayoutReportNoMoveableNode` appears if you specified all the nodes as fixed.

To read the code, use the method:

```
int IlvGraphLayoutReport::getCode()
```

◆ Layout Time

This field contains the total duration of the layout algorithm at the end of the layout. To read the time (in milliseconds), use the method:

```
IlvRunTimeType IlvGraphLayoutReport::getLayoutTime()
```

Using Layout Event Listeners

The layout event listeners mechanism provides a way to inform the end user of what is happening during the layout. At times, a layout algorithm may take a long time to execute, especially when dealing with large graphs. In addition, an algorithm may not converge in some cases. No matter what the situation, the end user should be informed of the events that occur during the layout. This can be done by implementing a simple progress bar or by

displaying appropriate information, such as the current cost function value, after each iteration or step.

The layout event listener is defined by the `IlvGraphLayoutListener` class. To receive the layout events delivered during the layout, a class must implement the `IlvGraphLayoutListener` interface and must register itself using the `addGraphLayoutEventListener` method of the `IlvGraphLayout` class.

When you subclass the `IlvGraphLayoutListener` class, you must implement the `IlvGraphLayout::layoutStepPerformed` method. The layout algorithm will call this method on all the registered layout event listeners, passing the layout report as an argument. In this way, you can read information about the current state of the layout report. (For example, you can read this information after each iteration or step of the layout algorithm).

The following example shows how to implement a layout event listener. This example uses the Orthogonal Link Layout.

```
class OrthogonalIterationListener
: public IlvGraphLayoutListener
{
public:
    OrthogonalIterationListener()
    : IlvGraphLayoutListener(), i(0)
    { }
    void layoutStepPerformed(const IlvGraphLayoutEvent& event)
    {
        ++i;
        IlvOrthogonalLinkLayout* layout =
            (IlvOrthogonalLinkLayout*)event.getLayout();
        IlvPrint("iteration %d over %d",
                i,
                layout->getAllowedNumberOfIterations());
    }
private:
    IlvInt i;
};
```

Then, register the listener on the layout instance as follows:

```
IlvGraphLayout* layout = new IlvOrthogonalLinkLayout();
OrthogonalIterationListener* listener = new OrthogonalIterationListener();

layout->addGraphLayoutEventListener(listener);
```

Using the Graph Model

The `IlvGraphModel` class defines a suitable, generic API for graphs that have to be laid out with IBM® ILOG® Views graph layout algorithms.

All the layout algorithms provided in IBM ILOG Views are designed to lay out a graph model. This allows applications to benefit from the graph layout algorithms whether or not they use the IBM ILOG Views grapher `IlvGrapher`). However, to make things very simple for the common case of applications that manipulate an `IlvGrapher`, it is not mandatory to work directly with the graph model except for some of the advanced features such as filtering.

The Graph Model Concept

With a graph model, you can use already-built graphs, nodes, and links that have been developed without IBM® ILOG® Views and apply the layout algorithms of IBM ILOG Views. The graph model provides the basic, generic operations for performing the layout. A subclass must be written to adapt the graph model to specific graph, node, and link objects. This subclass plays the role of an “adapter” or bridge between the application objects and the graph model. This often makes it much easier to add graph features to existing applications.

Figure 5.1 shows the relationship between the graph model and graph layout algorithms, IBM ILOG Views graphers, non-Views graphers, and manager views.

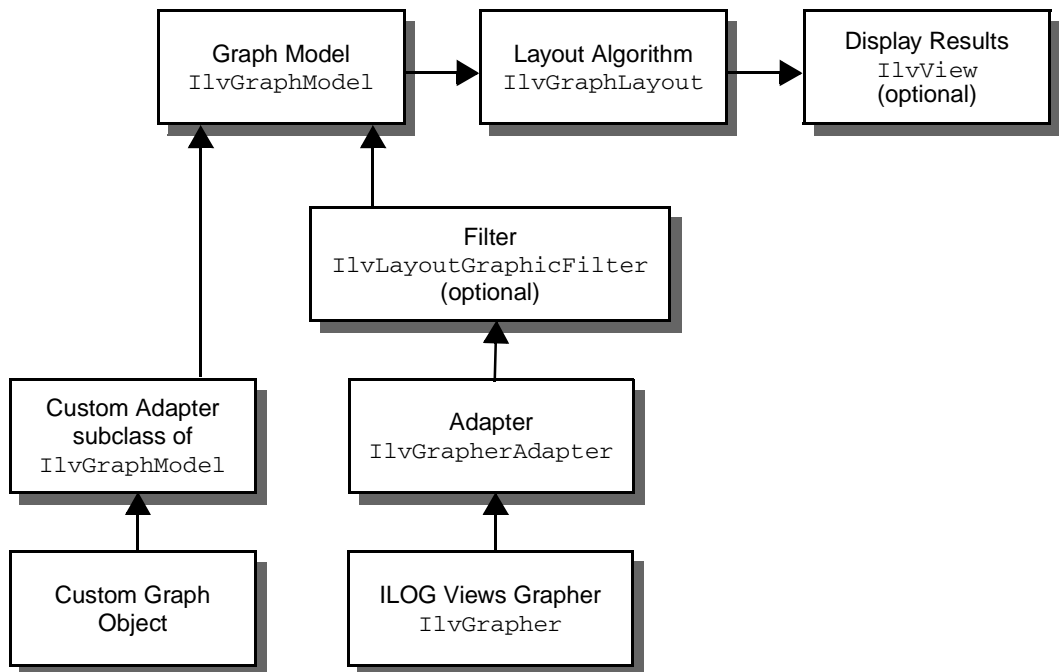


Figure 5.1 Graph Model in the IBM ILOG Views Graph Layout Package

You can see from this diagram that instead of using a concrete graph class such as `IlvGrapher` directly, the layout algorithms interact with the graph via the graph model. This is the key for achieving a truly generic graph layout framework. Note that the use of an `IlvView` to display the result of the layout is not mandatory.

The `IlvGraphModel` Class

The `IlvGraphModel` class is an abstract class. Because it does not provide a concrete implementation of a graph data structure, a complete implementation must be provided by “adapter” classes. The adapters extend the `IlvGraphModel` class and must use an underlying graph data structure. A special adapter class called `IlvGrapherAdapter` is provided so that an `IlvGrapher` can be used as the underlying graph data structure.

***Note:** If an application uses the `IlvGrapher` class, the grapher can be attached directly to the layout instance without explicitly using a graph model. (See the `IlvGraphLayout::attach(IlvGrapher)` method.) In this case, the appropriate adapter (`IlvGrapherAdapter`) will be created internally. This adapter can be retrieved using the `IlvGraphLayout::getGraphModel()` method, which will return an instance of `IlvGrapherAdapter`.*

The methods defined in the `IlvGraphModel` class can be divided into several categories that provide information on the structure of the graph, the geometry of the graph, modification of the graph geometry, and notification of changes in the graph.

Information on the Structure of the Graph

The following methods of the `IlvGraphModel` class allow the layout algorithms to retrieve information on the structure of the graph:

```
IlList IlvGraphModel::getNodesAndLinks()
IlList IlvGraphModel::getNodes()
IlUInt IlvGraphModel::getNodesCount()
IlList IlvGraphModel::getLinks()
IlUInt IlvGraphModel::getLinksCount()
IlBoolean IlvGraphModel::isNode(IlAny obj)
IlBoolean IlvGraphModel::isLink(IlAny obj)
IlList IlvGraphModel::getLinks(IlAny node)
IlUInt IlvGraphModel::getLinksCount(IlAny node)
IlList IlvGraphModel::getLinksFrom(IlAny node)
IlUInt IlvGraphModel::getLinksFromCount(IlAny node)
```

```

IList IlvGraphModel::getLinksTo(IlAny node)
IUInt IlvGraphModel::getLinksToCount(IlAny node)
IList IlvGraphModel::getNeighbors(IlAny node)
IUInt IlvGraphModel::getNodeDegree(IlAny node)
IlAny IlvGraphModel::getFrom(IlAny link)
IlAny IlvGraphModel::getTo(IlAny link)
IlAny IlvGraphModel::getOpposite(IlAny link, IlAny node)
IBoolean IlvGraphModel::isLinkBetween(IlAny node1, IlAny node2)

```

Most of these methods have a name and definition very similar to the corresponding methods of the `IlvGrapher` class. The main difference is that the arguments of the `IlvGraphModel` methods are `IlAny` instead of `IlvGraphic` or `IlvLinkImage`.

Information on the Geometry of the Graph

The following methods of the `IlvGraphModel` class allow the layout algorithms to retrieve information on the geometry of the graph:

```

IlvGraphModel::boundingBox(IlAny nodeOrLink, IlvRect& rect)
IlvPoint* IlvGraphModel::getLinkPoints(IlAny link, IUInt& count)
IlvPoint IlvGraphModel::getLinkPointAt(IlAny link, IUInt index)
IInt IlvGraphModel::getLinkPointsCount(IlAny link)
IUInt IlvGraphModel::getLinkWidth(IlAny link)

```

The `IlvGraphModel::boundingBox` method is called by a layout algorithm whenever it needs to get the position and the dimension of a node or a link. The other methods are used mainly by link layout algorithms.

Modification of the Geometry of the Graph

The following methods of the `IlvGraphModel` class allow a layout algorithm to modify the geometry of the graph:

```

void IlvGraphModel::moveNode(IlAny node, IInt x, IInt y,
IBoolean redraw)
void IlvGraphModel::reshapeLink(IlAny link, IlvPoint fromPoint,
IlvPoint* points,
IUInt startIndex, IUInt length,
IlvPoint toPoint, IBoolean redraw)

```

Layout algorithms that compute new coordinates for the nodes use the `IlvGraphModel::moveNode` method. Link layout algorithms that compute new shapes for the links call one of the `IlvGraphModel::reshapeLink` methods.

Notification of Changes

The following methods of the `IlvGraphModel` class allow a layout algorithm to be notified of changes in the graph:

```
void IlvGraphModel::addGraphModelListener (IlvGraphModelListener*
listener)

void
IlvGraphModel::removeGraphModelListener (IlvGraphModelListener*
listener)

void IlvGraphModel::fireGraphModelEvent (IlvGraphModelEvent&
event)
```

A “change” in the graph can be a structure change (that is, a node or a link was added or removed) or a geometry change (that is, a node or a link was moved or reshaped). The graph model event listener mechanism provides a means to keep the layout algorithms informed of these changes. When the layout algorithm is restarted on the same graph, it is able to detect whether the graph has changed since the last time the layout was successfully performed. If necessary, the layout can be performed again. If there is no change in the graph, the layout algorithm can avoid unnecessary work by not performing the layout.

The graph model event listener is a subclass of the `IlvGraphModelListener` class. To receive the graph model events (that is, instances of the `IlvGraphModelEvent` class), a subclass of the `IlvGraphModelListener` class must register itself using the `IlvGraphModel::addGraphModelListener` method of the `IlvGraphModel` class.

Note: *The creation of the graph model event listener is handled transparently by the `IlvGraphLayout` class. Therefore, there is usually no need to manipulate this listener directly.*

Storing and Retrieving Data Objects (“Properties”)

The following methods of the `IlvGraphModel` class allow a layout algorithm to store data objects for each node:

```
void IlvGraphModel::setProperty (IlAny nodeOrLink, const char* key,
IlAny value)

IlAny IlvGraphModel::getProperty (IlAny nodeOrLink, const char*
key)
```

The layout algorithm may need to associate a set of properties with the nodes and links of the graph. Properties are a set of `key-value` pairs, where the `key` is a `const char*` object and the `value` can be any kind of information value.

Note: *Creating a property and associating it with a node and link is handled transparently by the layout algorithm whenever it is necessary. Therefore, there is usually no need to manipulate the properties directly. However, if needed, you can do this in your own subclass of `IlvGraphLayout`. In this case, you must define `cleanObjectProperties` in your subclass.*

Using the `IlvGrapherAdapter`

The `IlvGrapherAdapter` class is a concrete subclass of `IlvGraphModel` that allows an `IlvGrapher` to be laid out using the layout algorithms provided in IBM ILOG Views. It provides an implementation for all the abstract methods of `IlvGraphModel`. It also provides an overridden implementation of some nonabstract methods of `IlvGraphModel` to improve efficiency by taking advantage of the characteristics of the `IlvGrapher`.

If an application uses the `IlvGrapher` class, the grapher can be attached directly to the layout instance without explicitly using the adapter. (See the `IlvGraphLayout::attach(IlvGrapher*)` method.) In this case, an `IlvGrapherAdapter` is created internally by the layout class. The adapter can be retrieved using the `IlvGraphLayout::getGraphModel()` method, which will return an instance of `IlvGrapherAdapter`.

Additionally, the `IlvGrapherAdapter` class provides a way to filter the `IlvGrapher`. By using the filtering mechanism, you specify a particular set of nodes and links that have to be taken into account by the layout algorithm. (See *Using the Filtering Features to Lay Out a Part of an `IlvGrapher`*.)

The `IlvGrapherAdapter` class also allows you to specify the `IlvTransformer` that has to be used for computing the geometry of the graph. (See *Laying Out Graphs with Nonzoomable Graphic Objects*.)

Note: *For details on how to write your own adapter, see *Laying Out a Non-Views Grapher*.*

Laying Out a Non-Views Grapher

Note: *To understand this section better, read *Using the Graph Model* first.*

It is sometimes necessary to add graph layout features to an existing application. If the application already uses the IBM ILOG Views grapher (`IlvGrapher`) to manipulate and display graphs, using the graph layout algorithms provided in IBM ILOG Views is a straightforward process. No adapter has to be written.

However, the case may arise where an application uses its own classes for nodes, links, and graphs, and where, for some reason, you do not want to replace these classes with IBM ILOG Views classes. To enable the graph layout algorithms to work with these graph objects, a custom adapter (that is, a subclass of `IlvGraphModel`) must be written.

The adapter must implement all the abstract methods of the `IlvGraphModel` class. The nonabstract methods of this class have a default implementation that is really functional. However, they may not be optimal because they do not take advantage of the characteristics of the underlying graph implementation. For better performance, the following nonabstract methods can be overridden in the adapter class:

```
IlUInt IlvGraphModel::getNodesCount()
IlUInt IlvGraphModel::getLinksCount()
IlUInt IlvGraphModel::getLinksCount(IlAny node)
IlUInt IlvGraphModel::getLinksFromCount(IlAny node)
IlUInt IlvGraphModel::getLinksToCount(IlAny node)
IlUInt IlvGraphModel::getLinkPointAt(IlAny link, IlUInt index)
```

The efficiency of the layout algorithm depends directly on the efficiency of the implementation of the adapter class and the underlying graph data structure.

Using the Filtering Features to Lay Out a Part of an `IlvGrapher`

Note: To understand this section better, read *Using the Graph Model* first.

Applications sometimes need to perform the layout algorithm on a subset of the nodes and links of a graph. If the graph is not an `IlvGrapher`, the custom adapter should support the filtering of a graph. (See *Laying Out a Non-Views Grapher*.) The methods that are related to the structure of the graph (`IlvGrapherAdapter::getNodes`, `IlvGrapherAdapter::getLinks`, `IlvGrapherAdapter::getNeighbors`, and so on) must behave just as if the graph has changed in some way. They must take into account only the nodes and links that belong to the part of the graph that must be laid out.

For applications that use `IlvGrapher`, the filtering feature is built into the `IlvGrapherAdapter`. To do this, the `IlvGrapherAdapter` needs a way to know, for each

node or link, whether it must be taken into account during the layout. This is the role of the “filter” class, `IlvLayoutGraphicFilter`.

The `IlvLayoutGraphicFilter` class defines this main method:

```
IlBoolean IlvLayoutGraphicFilter::accept(IlvGraphic nodeOrLink)
```

If a filter is specified, the `IlvGrapherAdapter` calls the `IlvLayoutGraphicFilter::accept` method for each node or link whenever necessary. If the method returns `IlTrue`, the `IlvGrapherAdapter` considers the node or the link as part of the graph that needs to be laid out. Otherwise, it ignores the node or the link.

To specify a filter on an `IlvGrapherAdapter`, use the following method of the `IlvGrapherAdapter` class:

```
void IlvGrapherAdapter::setFilter(IlvLayoutGraphicFilter* filter)
```

To remove the filter, call the `IlvGrapherAdapter::setFilter` method with a null argument.

To obtain the filter that has been specified, use the method:

```
IlvLayoutGraphicFilter* IlvGrapherAdapter::getFilter()
```

Note: All overridden implementations of the `IlvLayoutGraphicFilter::accept` method must respect the following rule: a link cannot be accepted by the filter if any of its end nodes (origin or destination nodes) are not accepted.

Note: The filter is not managed by the adapter. You have to store it and destroy it when it is no longer needed.

There are two ways to filter an `IlvGrapher`, by layers or by graphic objects. The two methods can be combined.

Filtering by Layers

Inside an `IlvGrapher`, nodes and links can be managed by layers. (See the `IlvManagerLayer` class). IBM ILOG Views allows you to specify that only nodes and links belonging to certain layers have to be taken into account when performing the layout. Use the following methods of the `IlvGrapherAdapter` class:

```
void IlvGrapherAdapter::addLayer(IlvManagerLayer* layer)
```

```
IlBoolean IlvGrapherAdapter::removeLayer(IlvManagerLayer* layer)
```

```
IlBoolean IlvGrapherAdapter::removeAllLayers()
```

To get an enumeration of the manager layers to be taken into account during the layout, use the method:

```
const IlArray& IlvGrapherAdapter::getLayers()
```

If no layers have been specified or all the specified layers have been removed, all layers in the `IlvGrapher` are used. In this case, the `IlvGrapherAdapter::getLayers` method returns an empty array.

When at least one layer is specified, an `IlvLayoutGraphicFilter` is created internally if it has not already been specified using the `IlvGrapherAdapter::setFilter` method. The default implementation of its `IlvLayoutGraphicFilter::accept` method will automatically check whether a node or a link received as an argument belongs to one of the specified layers.

Filtering by Graphic Objects

The nodes and links to be taken into account during the layout can also be filtered individually. To do this, a custom subclass of `IlvLayoutGraphicFilter` must be written. The filtering rules have to be embedded in the overridden implementation of the `IlvLayoutGraphicFilter::accept` method. For example, user properties could be used by an application to “mark” nodes and links that have to be accepted by the filter. The filter class could then be written as follows. In this example, the name of the property is stored in the variable `FILTER_PROP`.

```
class LayoutFilter
: public IlvLayoutGraphicFilter
{
public:
    LayoutFilter()
    {
    }
    IlvBoolean accept(IlvGraphic* obj)
    {
        IlvAny prop = obj->getProperty(IlvGetSymbol(FILTER_PROP));
        if (!prop)
            return IlvFalse;
        // accept a link only if its two end-nodes are accepted
        if (obj->isSubtypeOf(IlvLinkImage::ClassInfo()->getClassName()) {
            IlvLinkImage* link = (IlvLinkImage*)obj;
            return (link->getFrom()->getProperty(IlvGetSymbol(FILTER_PROP)) &&
                    link->getTo()->getProperty(IlvGetSymbol(FILTER_PROP)));
        }
        return IlvTrue;
    }
};
```

Laying Out Graphs with Nonzoomable Graphic Objects

Note: To understand this section better, read *Using the Graph Model* first.

Graph layout algorithms have to deal with the geometry of the graph, that is, the position and shape of the nodes and links. In addition, they must deal with the layout of an `IlvGrapher`. The nodes of an IBM ILOG Views `IlvGrapher` can be any graphic object, that is, any subclass of `IlvGraphic`. The position and size of the nodes are given by their `IlvGraphic::boundingBox(IlvTransformer t)` method and usually depend on the transformer used for their display. Therefore, when an `IlvGrapher` has to be laid out, the geometry of the grapher must be considered for a given value of the transformer.

The most natural transformer value that could be chosen is the “identity” transformer. An identity transformer has no translation, zoom, or rotation factors. In terms of IBM ILOG Views, this would mean that the geometry of the `IlvGrapher` would be considered in the manager coordinates, not in the manager view coordinates (transformed coordinates). However, the special case of *nonzoomable* graphic objects must be taken into account. For this case, the idea of simply using the geometry of the grapher in manager coordinates is not pertinent.

A Special Case: Nonzoomable Graphic Objects

A graphic object is said to be *zoomable* if its bounding box follows the zoom level. Otherwise, the object is *nonzoomable*. (To know whether a graphic object is zoomable, use its `IlvBoolean zoomable()` method, or check its documentation.)

If all the nodes and links of an `IlvGrapher` are zoomable graphic objects, a layout obtained on the basis of the graph geometry in manager coordinates will “look” the same for any value of the transformer used for the display. Simply speaking, the drawing of the graph will just be zoomed, or translated.

When at least one nonzoomable graphic object is used as a node in an `IlvGrapher`, the geometry of the grapher in manager coordinates can no longer be used. When drawn with different transformer values (for instance, at different zoom levels), the same `IlvGrapher` can look very different.

Reference Transformers

When a grapher contains nonzoomable graphic objects, it is not possible to deal with the geometry of the `IlvGrapher` based on the graph objects bounding boxes systematically computed for an identity transformer (manager coordinates). To ensure that the drawing of the laid-out graph is always correct, including the case where nonzoomable graphic objects are present, the transformer used for the display must be considered. Generally speaking, the layout of an `IlvGrapher` depends on the transformer.

Instead of dealing with the issue of zoomable/nonzoomable objects and transformers at the level of the layout algorithms, the IBM ILOG Views Graph Layout package delegates this task to the `IlvGrapherAdapter`.

Layout algorithms interact with the geometry of the graph using generic methods of the graph model (`IlvGraphModel`), such as `IlvGraphModel::boundingBox(IlAny nodeOrLink)`. The distinction between zoomable and nonzoomable objects, and the notion of transformer (`IlvTransformer`), have been pushed completely outside this level of the layout framework. The layout algorithms consider the geometry of the graph exactly as it is provided by the graph model. From the point of view of the layout algorithms, the problem of zoomable and nonzoomable objects is completely transparent. Therefore, when writing a layout algorithm, you do not need to be concerned with that.

In the case of an `IlvGrapher`, the `IlvGrapherAdapter` needs to compute the geometry of the graph for a given transformer. This is what we call the *reference transformer*. Usually, the reference transformer is the transformer that is currently being used for the display of the `IlvGrapher`.

How a Reference Transformer is Used

For a simple example of how a reference transformer is used, consider the `IlvGraphModel::boundingBox(IlAny nodeOrLink)` method. This abstract method of the `IlvGraphModel` class is implemented in the `IlvGrapherAdapter`. To compute the bounding box, it calls the `IlvGraphic::boundingBox(IlvTransformer t)` method of the graphic object that it receives as an argument. However, it does not handle zoomable objects and nonzoomable objects in the same way.

If the graphic object is zoomable, the `IlvGrapherAdapter::boundingBox(IlAny nodeOrLink)` method of the `IlvGrapherAdapter` returns the bounding box in manager coordinates by calling `IlvGraphic::boundingBox(null)`.

If the graphic object is nonzoomable, the `IlvGrapherAdapter::boundingBox(IlAny nodeOrLink)` method computes the bounding box according to the reference transformer and returns a rectangle obtained by applying the inverse transformation to this rectangle. (See the `IlvTransformer::inverse(IlvRect rect)` method.)

The geometry of the `IlvGrapher` is computed in such a manner that the resulting drawing inside an `IlvMgrView` using the reference transformer will look fine.

Reference Views

Optionally, an `IlvMgrView` can be specified as a *reference view* for the `IlvGrapherAdapter`. If a reference view is specified, its current transformer (at the moment when the layout is started) is automatically used as the reference transformer. Usually, applications use the same manager view that is used for the display of the `IlvGrapher` as the reference view (but this is not mandatory).

To specify the reference view, use the following method:

```
void IlvGrapherAdapter::setReferenceView(IlvMgrView view)
```

To get the current reference view, use the method:

```
IlvMgrView IlvGrapherAdapter::getReferenceView()
```

If no view has been specified as the reference view, the method returns `null`.

Specifying a Reference Transformer

A reference transformer can be specified explicitly using the method:

```
void IlvGrapherAdapter::setReferenceTransformer (IlvTransformer
transformer)
```

The current reference transformer is returned by the method:

```
IlvTransformer IlvGrapherAdapter::getReferenceTransformer ()
```

In most cases, it is not necessary to specify a reference transformer because the last method automatically chooses it according to the following rules:

- ◆ If a reference transformer is specified, the specified transformer is returned.
- ◆ If a reference view has been specified, the transformer of the reference view is returned.
- ◆ If the `IlvGrapher` attached to the `IlvGrapherAdapter` has at least one manager view, the transformer of the first manager (as returned by the method `IlvManager::getViews()`) is returned.

The only cases when you may need to specify a reference transformer or a reference view are the following:

- ◆ The `IlvGrapher` contains nonzoomable objects (that is, the layout cannot be correctly computed independently of the transformer used for drawing the graph) and more than one manager view is attached to the grapher.
- ◆ The `IlvGrapher` contains nonzoomable objects and you want to perform the layout without attaching a manager view to the grapher. (Therefore, the default rule for choosing the current transformer of the first manager view as the reference transformer cannot be applied.)

If a grapher containing nonzoomable objects is displayed simultaneously in several views, you can use the `IlvGrapherAdapter::setReferenceView` method to indicate the view for which you want the drawing of the graph to be optimal.

If you specified a reference transformer but want to reset this setting and go back to the default behavior, call the method `setReferenceTransformer` with a `null` argument.

Note that if you override the `IlvGrapherAdapter::setReferenceTransformer` method, you must call `IlvGrapherAdapter::setReferenceTransformer` of the super class to notify the `IlvGrapherAdapter` that the reference transformer has changed.

Note also that a call to the `IlvGrapherAdapter::setReferenceView` method overrides the effect of a call to the `IlvGrapherAdapter::setReferenceTransformer` method. In the same way, a call to the `IlvGrapherAdapter::setReferenceTransformer`

method overrides the effect of a call to the `IlvGrapherAdapter::setReferenceView` method.

Defining a New Type of Layout

If the layout algorithms provided in IBM® ILOG® Views do not meet your needs, you can develop your own layout algorithms by subclassing `IlvGraphLayout`.

When a subclass of `IlvGraphLayout` is created, it automatically fits into the generic layout framework of IBM ILOG Views and benefits from its infrastructure: generic parameters, notification of progress, and the capability to lay out any graph object using the generic graph model.

Sample Code

To illustrate the basic ideas for defining a new layout, the following simple example shows a possible implementation of the simplest layout algorithm, the Random Layout. The new layout class is called `MyRandomLayout`.

The following shows the skeleton of the class:

```
class MyRandomLayout
: public IlvGraphLayout
{
public:
    MyRandomLayout()
    {
    }
protected:
    void layout();
};
```

The constructor is empty. Then, the abstract method `layout()` of the superclass is implemented as follows:

```

void
MyRandomLayout::layout()
{
    // obtain the graph model
    IlvGraphModel* graphModel = getGraphModel();

    // obtain the layout report
    IlvGraphLayoutReport* layoutReport = getLayoutReport();

    IlvBoolean atLeastOneNodeMoved = IlvFalse;

    // obtain the layout region
    IlvRect rect;
    getLayoutRegion(rect);
    IlvPos xMin = rect.x();
    IlvPos yMin = rect.y();
    IlvPos xMax = rect.right();
    IlvPos yMax = rect.bottom();

    // initialize the random generator
    IlvRandom* random = (isUseSeedValueForRandomGenerator()) ?
        new IlvRandom(getSeedValueForRandomGenerator()) :
        new IlvRandom();

    // get the objects in the grapher
    IlvList* nodes = graphModel->getNodes();

    // browse the objects in the grapher
    IlvLink* l = nodes->getFirst();
    while (l) {
        IlvAny node = l->getValue();
        l = l->getNext();

        // skip fixed nodes
        if (isPreserveFixedNodes() && isFixed(node))
            continue;

        // compute coordinates
        IlvPos x = xMin + (IlvPos)((xMax - xMin) * random->nextFloat());
        IlvPos y = yMin + (IlvPos)((yMax - yMin) * random->nextFloat());

        // move the node to the computed position
        graphModel->moveNode(node, x, y, IlvFalse);
        atLeastOneNodeMoved = IlvTrue;

        // notify listeners on layout events
        layoutStepPerformed();
    }
    delete random;

    // set the layout report code
    if (atLeastOneNodeMoved)
        layoutReport->setCode(IlvLayoutReportLayoutDone);
    else
        layoutReport->setCode(IlvLayoutReportNoMoveableNode);
}

```


Note that the `layout()` method is protected, which is the access type of the method in the base class. This will not prevent a user outside the module that contains the class from performing the layout, because it is started using the public method `performLayout`.

Steps for Implementing the Layout Method

In our example, the `layout` method is implemented using the following main steps:

1. Obtain the graph model (`getGraphModel()` on the layout instance).

```
IlvGraphModel* graphModel = getGraphModel();
```

2. Obtain the instance of the layout report that is automatically created when the `performLayout` method from the superclass is called (`getLayoutReport()` on the layout instance).

```
IlvGraphLayoutReport* layoutReport = getLayoutReport();
```

3. Obtain the layout region parameter (`getLayoutRegion()` on the layout instance) to compute the area where the nodes will be placed.

```
IlvRect rect;  
getLayoutRegion(rect);
```

4. Initialize the random generator. (For information on the seed value parameter, see the Random Generator Seed Value.)

```
IlvRandom* random = (isUseSeedValueForRandomGenerator()) ?  
    new IlvRandom(getSeedValueForRandomGenerator()) :  
    new IlvRandom();
```

5. Get a list of the nodes (`getNodes()` on the graph model instance).

```
IlList* nodes = graphModel->getNodes();
```

6. Browse the nodes, skipping fixed nodes (`isFixed(node)` on the layout instance) if asked by the user (`isPreserveFixedNodes()` on the layout instance).

```
// browse the objects in the grapher  
IlLink* l = nodes->getFirst();  
while (l) {  
    IlAny node = l->getValue();  
    // ...  
}
```

7. Move each node to the newly computed coordinates inside the layout region (`graphModel.moveNode`).

```
graphModel->moveNode(node, x, y, IlFalse);
atLeastOneNodeMoved = IlTrue;
```

8. Notify the listeners on layout events that a new node was positioned (`layoutStepPerformed()` on the layout instance). This allows the user to implement, for example, a progress bar if a layout event listener was registered on the layout instance.

```
layoutStepPerformed();
```

9. Finally, set the appropriate code in the layout report.

```
if (atLeastOneNodeMoved)
layoutReport->setCode(IlvLayoutReportLayoutDone);
else
layoutReport->setCode(IlvLayoutReportNoMoveableNode);
```

Of course, depending on the characteristics of the layout algorithm, some of these steps may be different or unnecessary, or other steps may be needed.

Questions and Answers about Using the Layout Algorithms

Table 5.2 provides some helpful suggestions for using the layout algorithms. You may find some answers to questions that come up when using the Graph Layout package.

Table 5.2 Questions and Answers about the Layout Algorithms

Question	Answer
I perform the layout and nothing happens (no node is moved). Why?	<p>One possible reason may be: the layout algorithms provided in IBM ILOG Views are all designed to do nothing, by default, if no change occurred in the graph since the last time the layout was performed successfully on the same graph. A change means that a node was moved, or a node or link was added, removed, or reshaped.</p> <p>Another possible reason may be: an error or a special case occurred during the layout. You should call the <code>IlvGraphLayoutReport::getCode()</code> method on the instance of layout report returned by the <code>IlvGraphLayout::performLayout</code> method. Check this value with respect to the documentation of the appropriate layout report class.</p>
After performing the layout, the graph is laid out far from its initial position. Why?	<p>Most of the layout algorithms use a layout region parameter to control the size and position of the layout. (For details of this parameter in the <code>IlvGraphLayout</code> class, see Layout Region). Depending on the value of this parameter, the nodes may be moved far from their initial positions.</p> <p>To know whether a layout algorithm is designed to use a layout region parameter, check the documentation to see if the layout class overrides the <code>supportsLayoutRegion()</code> method of the base class in order to return <code>ILTrue</code>.</p>
When I use certain layout algorithms on certain graphs, there are overlapping nodes. Why and what can I do?	<p>One possible reason may be related to the different ways layout algorithms deal with the size of the nodes. The Tree, Hierarchical, and Bus algorithms always avoid overlapping nodes. (The Orthogonal Link algorithm does not move the nodes.)</p> <p>In any case, if the layout algorithm supports the layout region mechanism, you should try to increase the size of the layout region. For example, if your graph contains hundreds of nodes, it is not reasonable to use a small layout region, such as 600x600. There will not be enough space for all the nodes. You should try a larger layout region, for example 5000x5000.</p>

Table 5.2 *Questions and Answers about the Layout Algorithms (Continued)*

Question	Answer
<p>In some networks, there are two subnetworks that are not connected, how will this affect the layout algorithms provided in IBM ILOG Views?</p>	<p>This depends on the layout class you use:</p> <ul style="list-style-type: none"> ◆ <code>IlvBusLayout</code>: It will work on the “connected component” of the graph that contains the “bus object”. (You must specify the bus object as a parameter.) The other nodes that are not connected to the bus will not be moved. ◆ <code>IlvTreeLayout</code>, <code>IlvHierarchicalLayout</code>: No problem, it works well on these graphs. For the Circular and Radial Tree Layout, each connected subgraph will be laid out separately and positioned on a row/column grid. ◆ <code>IlvOrthogonalLinkLayout</code>: No problem, the algorithm does not differentiate between connected and disconnected graphs. <p>However, note that you can always deal with a disconnected graph by cutting it into several connected subgraphs (clusters). You can either create a new graph for each connected graph and lay out the new graph separately, or use the filtering feature to lay out each connected graph.</p>
<p>There are some attributes of the network that we know about (for instance, we know what the core switch is and what the center should be). Are such attributes taken into account by the layout algorithm?</p>	<p>It depends on the layout algorithm. In the Tree Layout algorithm, you can specify the root node. In the Bus Layout algorithm, you can specify the bus object. In the Hierarchical Layout algorithm, you can specify node position indices and level indices.</p>
<p>If I use IBM ILOG Views on different computers, I sometimes get different layouts for the same graph with the same parameters. Why?</p>	<p>There are two possible reasons:</p> <ul style="list-style-type: none"> ◆ Different computers may be slower or faster. If the layout algorithm you use stops the computation when the specified allowed time has elapsed, a slower computer will cause the computation to stop earlier. That may be the cause of different results. This may happen even with the same computer if the load of the computer is increased. You can try to increase the allowed time specification. ◆ If you use a layout algorithm that uses the random generator and if you use the default option for the seed value (that is, the system clock is used), you get different results even on the same computer.

Index

A

accept
 IlvLayoutGraphicFilter class **143**
addGraphLayoutEventListener method
 IlvGraphLayout class **136**
addGraphModelListener method
 IlvGraphModel class **140**
addLayer
 IlvLayoutGraphicFilter class **143**
alignment
 free layout mode **62, 63**
 mixed **63**
allowed time parameter
 Hierarchical Layout **90**
 Orthogonal Link Layout **111**
 Tree Layout **56**
angle layout criteria **26**
area layout criteria **26**
attach method
 IlvGraphLayout class **29**
attaching a grapher **29**
automatic layout **27**

B

bends layout criteria **26**
boundingBox method
 IlvGraphic class **145**
 IlvGraphModel class **139**

Bus Layout

applicable graph types **119**
application domains **119**
bus node parameter **124**
code sample **120**
description **120**
features **119**
generic parameters **121**
horizontal offset parameter **126**
link style parameter **124**
margin on bus parameter **127**
margin parameter **127**
order parameter **122**
ordering options table **123**
parameters **121**
sample drawing **119**
specific parameters **122**
vertical offset parameter **126**
vertical offset to previous level parameter **126**
bus node parameter, Bus Layout **124**

C

C++
 prerequisites **9**
 calculated level index
 Hierarchical Layout **102**
 calculated position index
 Hierarchical Layout **103**
compass directions

- Tree Layout **58**
- connector style
 - Hierarchical Layout **96**
- createLayoutReport method
 - IlvGraphLayout class **134**
- creating a layout report **134**

D

- defining a new type of layout **148**
- detach method
 - IlvGraphLayout class **31**
- detaching a grapher **31**

E

- east-west neighbors **80**
- examples
 - extracted **11**

F

- features
 - generic parameter descriptions **20**
 - layout algorithms **17**
- fields of application for graph layouts **21**
- filtering
 - by graphic objects **144**
 - by layers **143**
- filtering features **142**
- fireGraphModelEvent method
 - IlvGraphModel class **140**
- fixed links parameter
 - Bus Layout **122**
 - Hierarchical Layout **90**
 - Orthogonal Link Layout **111**
 - Random Layout **117**
 - Tree Layout **56**
- fixed nodes parameter
 - Bus Layout **122**
 - Hierarchical Layout **90**
 - Random Layout **118**
 - Tree Layout **56**
- flow direction
 - free layout mode **60**

- flow direction parameter
 - Hierarchical Layout **90**
- free layout mode
 - alignment **62**
 - alignment of individual nodes **63**
 - description **59**
 - experts spacing parameters **69**
 - flow direction **60**
 - global alignment **62**
 - global link style **66**
 - individual link style **66**
 - link style **65**
 - spacing parameters **67**
 - tip-over alignment **64**

G

- getAlignment method
 - IlvTreeLayout class **64**
- getAllowedNumberOfIterations method
 - IlvOrthogonalLinkLayout class **114**
- getAllowedTime method
 - IlvGraphLayout class **32**
- getAspectRatio method
 - IlvTreeLayout class **75**
- getBranchOffset method
 - IlvTreeLayout class **69**
- getBus method
 - IlvBusLayout class **124**
- getCalcBackwardTreeLinks method
 - IlvTreeLayout class **82**
- getCalcForwardTreeLinks method
 - IlvTreeLayout class **82**
- getCalcNodeLevelIndex method
 - IlvHierarchicalLayout class **102**
- getCalcNodePositionIndex method
 - IlvHierarchicalLayout class **103**
- getCalcNonTreeLinks method
 - IlvTreeLayout class **82**
- getCalcRoots method
 - IlvTreeLayout class **57**
- getCode method
 - IlvGraphLayoutReport class **135**
- getConnectorStyle method
 - IlvHierarchicalLayout class **97**

getEastNeighbor method
 IlvTreeLayout class **81**
 getFilter
 IlvLayoutGraphicFilter class **143**
 getFlowDirection method
 IlvHierarchicalLayout class **91**
 IlvTreeLayout class **62**
 getFrom method
 IlvGraphModel class **139**
 getGlobalAlignment method
 IlvTreeLayout class **63**
 getGlobalLinkStyle method
 IlvHierarchicalLayout class **95**
 getHorizontalLinkOffset method
 IlvHierarchicalLayout class **100**
 getHorizontalNodeLinkOffset method
 IlvHierarchicalLayout class **100**
 getHorizontalNodeOffset method
 IlvHierarchicalLayout class **100**
 getHorizontalOffset method
 IlvBusLayout class **126**
 getIndex method
 IlvBusLayout class **123**
 getLayers
 IlvLayoutGraphicFilter class **143**
 getLayoutMode method
 IlvTreeLayout class **59**
 getLayoutRegion method
 IlvGraphLayout class **33**
 getLayoutTime method
 IlvGraphLayoutReport class **135**
 getLevelJustification method
 IlvHierarchicalLayout class **93**
 IlvTreeLayout class **72**
 getLinkPointAt method
 IlvGraphModel class **139, 142**
 getLinkPoints method
 IlvGraphModel class **139**
 getLinkPointsCount method
 IlvGraphModel class **139**
 getLinkPriority method
 IlvHierarchicalLayout class **97**
 getLinks method
 IlvGraphModel class **138**
 getLinksCount method
 IlvGraphModel class **138, 142**
 getLinksFrom method
 IlvGraphModel class **138**
 getLinksFromCount method
 IlvGraphModel class **138, 142**
 getLinksTo method
 IlvGraphModel class **139**
 getLinksToCount method
 IlvGraphModel class **139, 142**
 getLinkStyle method
 IlvBusLayout class **125**
 IlvHierarchicalLayout class **95**
 IlvOrthogonalLinkLayout class **113**
 IlvRandomLayout class **118**
 IlvTreeLayout class **66, 67**
 getLinkToNodeCrossingPenalty method
 IlvOrthogonalLinkLayout class **114, 115**
 getLinkWidth method
 IlvGraphModel class **139**
 getMargin method
 IlvBusLayout class **127**
 getMarginOnBus method
 IlvBusLayout class **127**
 getMaxChildrenAngle method
 IlvTreeLayout class **76**
 getMinFinalSegmentLength method
 IlvOrthogonalLinkLayout class **113**
 getNeighbors method
 IlvGraphModel class **139**
 getNodes method
 IlvGraphModel class **138**
 getNodesAndLinks method
 IlvGraphModel class **138**
 getNodesCount method
 IlvGraphModel class **138, 142**
 getNodesDegree method
 IlvGraphModel class **139**
 getOpposite method
 IlvGraphModel class **139**
 getOrdering method
 IlvBusLayout class **122**
 getOrthForkPercentage method
 IlvTreeLayout class **69**
 getOverlapPercentage method
 IlvTreeLayout class **69**

- getParentChildOffset method
 - IlvTreeLayout class **69**
- getPosition method
 - IlvTreeLayout class **58**
- getProperty method
 - IlvGraphModel class **140**
- getReferenceTransformer
 - IlvLayoutGraphicFilter class **147**
- getReferenceView
 - IlvLayoutGraphicFilter class **147**
- getSeedValueForRandomGenerator method
 - IlvGraphLayout class **36**
- getSiblingOffset method
 - IlvTreeLayout class **69**
- getSpecNodeLevelIndex method
 - IlvHierarchicalLayout class **98**
- getSpecNodePositionIndex method
 - IlvHierarchicalLayout class **99**
- getSpecRoots method
 - IlvTreeLayout class **57**
- getTipOverBranchOffset method
 - IlvTreeLayout class **69**
- getTo method
 - IlvGraphModel class **139**
- getVerticalLinkOffset method
 - IlvHierarchicalLayout class **101**
- getVerticalNodeLinkOffset method
 - IlvHierarchicalLayout class **101**
- getVerticalNodeOffset method
 - IlvHierarchicalLayout class **101**
- getVerticalOffsetToLevel method
 - IlvBusLayout class **126**
- getWestNeighbor method
 - IlvTreeLayout class **81**
- global alignment
 - free layout mode **62**
- global link style
 - free layout mode **66**
- global link style parameter
 - Hierarchical Layout **94**
- Graph Layout
 - basic procedure for using **39**
 - introduction **25**
 - sample application **40**
- graph layout

- basic concepts **25**
- fields of application **21**
- filtering by graphic objects **144**
- filtering by layers **143**
- filtering features **142**
- in IBM ILOG Views **27**
- in IBM ILOG Views Studio **41**
- nonzoomable graphic objects **145**
- questions and answers **152**
- reference transformers **145**
- reference views **146**
- specifying a reference transformer **147**
- with nonzoomable graphic objects **144**
- Graph Layout package
 - available layout styles **17**
 - description of **15**
 - features **17**
 - features for using layout algorithms **20**
- graph model
 - concept **137**
 - description of **136**
 - information on the geometry of the graph **139**
 - information on the structure of the graph **138**
 - modifying the geometry of the graph **139**
 - notification of changes of the graph **140**
 - retrieving data objects **140**
 - storing data objects **140**
- grapher
 - attaching **29**
 - detaching **31**
 - laying out a non-Views grapher **141**
- grapher adapter **141**
- GraphLayoutListener interface **136**

H

- Hierarchical Layout
 - applicable graph types **86**
 - application domains **86**
 - calculated level index **102**
 - calculated position index **103**
 - code sample **89**
 - connector style **96**
 - description **87**
 - features **86**

- flow direction parameter **90**
- generic parameters **89**
- global link style parameter **94**
- individual link style parameter **95**
- layout sequences **102**
- level index parameter **98**
- level justification parameter **92**
- limitations **87**
- link priority parameter **97**
- link style parameter **93**
- parameters **89**
- position index **99**
- refining a layout **104**
- sample drawings **84**
- spacing parameters **99**
- specific parameters **90**
- horizontal offset parameter, Bus Layout **126**

I

IBM ILOG Views

- 2D Graphics Professional product **28**

- IBM ILOG Views Studio graph layout extension **41**

IlvBusLayout class

- getBus method **124**
- getHorizontalOffset method **126**
- getIndex method **123**
- getLinkStyle method **125**
- getMargin method **127**
- getMarginOnBus method **127**
- getOrdering method **122**
- getVerticalOffsetToLevel method **126**
- setBus method **124**
- setHorizontalOffset method **126**
- setIndex method **123**
- setLinkStyle method **125**
- setMargin method **127**
- setMarginOnBus method **127**
- setOrdering method **122**
- setVerticalOffsetToLevel method **126**
- setVerticalOffsetToPreviousLevel method **126**

IlvGrapher class **142**

- boundingBox method **145**

IlvGrapherAdapter class **141**

IlvGraphLayout class **28**

- addGraphLayoutEventListener method **136**
- attach method **29**
- createLayoutReport method **134**
- detach method **31**
- getAllowedTime method **32**
- getLayoutRegion method **33**
- getSeedValueForRandomGenerator method **36**
- isAnimate method **32**
- isFixed method **34, 35**
- isLayoutTimeElapsed method **32**
- isPreserveFixedLinks method **34**
- isPreserveFixedNodes method **35**
- isUseDefaultParameters method **37**
- isUseSeedValueForRandomGenerator method **36**
- layout method **29, 150**
- layoutReport method **29**
- layoutStepPerformed method **136**
- performLayout method **30, 134**
- setAllowedTime method **32**
- setAnimate method **32**
- setFixed method **34, 35**
- setLayoutRegion method **33**
- setSeedValueForRandomGenerator method **36**
- setUseDefaultParameters method **37**
- supportsAllowedTime method **32**
- supportsAnimation method **32**
- supportsLayoutRegion method **34, 152**
- supportsPreserveFixedLinks method **34**
- supportsPreserveFixedNodes method **35**
- supportsRandomGenerator method **36**
- unfixAllLinks method **34**
- unfixAllNodes method **35**

IlvGraphLayoutListener class **136**

IlvGraphLayoutReport class

- description **134**
- getCode method **135**
- getLayoutTime method **135**
- stored information **135**

IlvGraphModel class

- addGraphModelListener method **140**
- boundingBox method **139**
- description of **138**
- fireGraphModelEvent method **140**

- getFrom method **139**
- getLinkPointAt method **139, 142**
- getLinkPoints method **139**
- getLinkPointsCount method **139**
- getLinks method **138**
- getLinksCount method **138, 142**
- getLinksFrom method **138**
- getLinksFromCount method **138, 142**
- getLinksTo method **139**
- getLinksToCount method **139, 142**
- getLinkWidth method **139**
- getNeighbors method **139**
- getNodeDegree method **139**
- getNodes method **138**
- getNodesAndLinks method **138**
- getNodesCount method **138, 142**
- getOpposite method **139**
- getProperty method **140**
- getTo method **139**
- isLink method **138**
- isLinkBetween method **139**
- isNode method **138**
- moveNode method **139**
- removeGraphModelListener method **140**
- reshapeLink method **139**
- setProperty method **140**
- IlvHierarchicalLayout class
 - getCalcNodeLevelIndex method **102**
 - getCalcNodePositionIndex method **103**
 - getConnectorStyle method **97**
 - getFlowDirection method **91**
 - getGlobalLinkStyle method **95**
 - getHorizontalLinkOffset method **100**
 - getHorizontalNodeLinkOffset method **100**
 - getHorizontalNodeOffset method **100**
 - getLevelJustification method **93**
 - getLinkPriority method **97**
 - getLinkStyle method **95**
 - getSpecNodeLevelIndex method **98**
 - getSpecNodePositionIndex method **99**
 - getVerticalLinkOffset method **101**
 - getVerticalNodeLinkOffset method **101**
 - getVerticalNodeOffset method **101**
 - setConnectorStyle method **96**
 - setFlowDirection method **91**

- setGlobalLinkStyle method **94**
- setHorizontalLinkOffset method **100**
- setHorizontalNodeLinkOffset method **100**
- setHorizontalNodeOffset method **100**
- setLevelJustification method **92**
- setLinkPriority method **97**
- setLinkStyle method **95**
- setSpecNodeLevelIndex method **98**
- setSpecNodePositionIndex method **99**
- setVerticalLinkOffset method **100**
- setVerticalNodeLinkOffset method **100**
- setVerticalNodeOffset method **100**
- IlvLayoutGraphicFilter class
 - accept method **143**
 - addLayer method **143**
 - getFilter method **143**
 - getLayers method **143**
 - getReferenceTransformer method **147**
 - getReferenceView method **147**
 - removeAllLayers method **143**
 - removeLayer method **143**
 - setFilter method **143**
 - setReferenceTransformer method **147**
 - setReferenceView method **146**
- IlvOrthogonalLinkLayout class
 - getAllowedNumberOfIterations method **114**
 - getLinkStyle method **113**
 - getLinkToNodeCrossingPenalty method **114, 115**
 - getMinFinalSegmentLength method **113**
 - isSameShapeForMultipleLinks method **114**
 - setAllowedNumberOfIterations method **113**
 - setLinkStyle method **113**
 - setLinkToLinkCrossingPenalty method **114**
 - setLinkToNodeCrossingPenalty method **114**
 - setMinFinalSegmentLength method **112**
 - setOffset method **112**
 - setSameShapeForMultipleLinks method **114**
- IlvRandomLayout class
 - getLinkStyle method **118**
 - setLinkStyle method **118**
- ilvtree library **49**
- IlvTreeLayout class **49**
 - getAlignment method **64**
 - getAspectRatio method **75**

- getBranchOffset method **69**
- getCalcBackwardTreeLinks method **82**
- getCalcForwardTreeLinks method **82**
- getCalcNonTreeLinks method **82**
- getCalcRoots method **57**
- getEastNeighbor method **81**
- getFlowDirection method **62**
- getGlobalAlignment method **63**
- getLayoutMode method **59**
- getLevelJustification method **72**
- getLinkStyle method **66, 67**
- getMaxChildrenAngle method **76**
- getOrthForkPercentage method **69**
- getOverlapPercentage method **69**
- getParentChildOffset method **69**
- getPosition method **58**
- getSiblingOffset method **69**
- getSpecRoots method **57**
- getTipOverBranchOffset method **69**
- getWestNeighbor method **81**
- isRootPosition method **58**
- setAlignment method **63, 65**
- setAllowedTime method **77**
- setAspectRatio method **75, 77**
- setBranchOffset method **68, 75**
- setEastWestNeighboring method **81**
- setFlowDirection method **61**
- setGlobalAlignment method **62, 64**
- setGlobalLinkStyle method **66**
- setLayoutMode method **59, 60, 70, 73, 74, 78, 79**
- setLevelJustification method **72**
- setLinkStyle method **67**
- setMaxChildrenAngle method **76**
- setOrthForkPercentage method **69**
- setOverlapPercentage method **69**
- setParentChildOffset method **68, 75**
- setPosition method **58**
- setPreference method **58**
- setRoot method **57**
- setRootPreference method **58**
- setSiblingOffset method **68, 75**
- setTipOverBranchOffset method **68**
- setWestEastNeighboring method **81**
- incremental layout **27**
- individual link style parameter

- Hierarchical Layout **95**
- instantiating a subclass **29**
- isAnimate method
 - IlvGraphLayout class **32**
- isFixed method
 - IlvGraphLayout class **34, 35**
- isLayoutTimeElapsed method
 - IlvGraphLayout class **32**
- isLink method
 - IlvGraphModel class **138**
- isLinkBetween method
 - IlvGraphModel class **139**
- isNode method
 - IlvGraphModel class **138**
- isPreserveFixedLinks method
 - IlvGraphLayout class **34**
- isPreserveFixedNodes method
 - IlvGraphLayout class **35**
- isRootPosition method
 - IlvTreeLayout class **58**
- isSameShapeForMultipleLinks method
 - IlvOrthogonalLinkLayout class **114**
- isUseDefaultParameters method
 - IlvGraphLayout class **37**
- isUseSeedValueForRandomGenerator method
 - IlvGraphLayout class **36**

L

- layout algorithm
 - basic procedure for using **39**
 - choosing **45**
 - table of additional information **49**
 - table of applicable graphs **47**
 - table of generic parameters supported **48**
- layout criteria
 - angle **26**
 - area **26**
 - bends **26**
 - link crossings **26**
 - symmetries **26**
- layout method
 - IlvGraphLayout class **29, 150**
 - steps for implementing **150**
- layout methods

- automatic **27**
- incremental **27**
- semi-automatic **27**
- static **27**
- layout modes
 - Tree Layout **59**
- layout region parameter
 - Bus Layout **121**
 - Random Layout **117**
- layout report
 - class table **134**
 - creating **134**
 - elapsed time **135**
 - information stored in report **135**
 - reading **135**
 - using **133**
- layoutReport method
 - IlvGraphLayout class **29**
- layouts
 - Bus Layout **119**
 - characteristics **49**
 - defining a new type **148**
 - Hierarchical Layout **84**
 - Orthogonal Link Layout **106**
 - performing **30**
 - procedure for implementing the layout method **150**
 - Random Layout **115**
 - sample code for defining a new type **148**
 - Tree Layout **49**
- layoutStepPerformed method
 - IlvGraphLayout class **136**
- level index parameter
 - Hierarchical Layout **98**
- level justification parameter
 - Hierarchical Layout **92**
- level layout mode
 - description **70**
- link crossing penalty parameter, Orthogonal Link Layout **114**
- link crossings layout criteria **26**
- link offset parameter, Orthogonal Link Layout **112**
- link priority parameter
 - Hierarchical Layout **97**
- link style
 - free layout mode **65, 66**
- link style parameter

- Bus Layout **124**
- Hierarchical Layout **93**
- Orthogonal Link Layout **113**
- Random Layout **118**
- links
 - retrieving categories **81**
- listener, layout event
 - code example **136**
 - description **135**
 - IlvGraphLayoutListener interface **136**

M

- manual
 - naming conventions **10**
 - notation **10**
 - organization **10**
- margin on bus parameter, Bus Layout **127**
- margin parameter, Bus Layout **127**
- minimum final segment parameter, Orthogonal Link Layout **112**
- moveNode method
 - IlvGraphModel class **139**

N

- naming conventions **10**
- notation **10**
- number of optimization iterations parameter, Orthogonal Link Layout **113**

O

- order parameter, Bus Layout **122**
- Orthogonal Link Layout
 - applicable graph types **108**
 - application domains **108**
 - code sample **110**
 - description **109**
 - features **108**
 - generic parameters **110**
 - limitations **109**
 - link crossing penalty parameter **114**
 - link offset parameter **112**
 - link style parameter **113**

- minimum final segment parameter **112**
- number of optimization iterations parameter **113**
- parameters **110**
- same shape for multiple links parameter **114**
- sample drawing **106**
- specific parameters **111**

P

parameters

- Bus Layout **121**
- description of generic parameters **31**
- generic, Bus Layout **121**
- generic, Hierarchical Layout **89**
- generic, Orthogonal Link Layout **110**
- generic, Random Layout **117**
- generic, Tree Layout **56**
- Hierarchical Layout **89**
- Orthogonal Link Layout **110**
- Random Layout **117**
- specific, Bus Layout **122**
- specific, Hierarchical Layout **90**
- specific, Orthogonal Link Layout **111**
- specific, Tree Layout **56**
- supported by layout algorithms (table) **48**
- Tree Layout **55**

parameters of `IlvGraphLayout`

- allowed time **31**
- animation **32**
- description of **31**
- layout region **33**
- preserved fixed links **34**
- preserved fixed nodes **35**
- random generator seed value **35**
- use default parameters **37**

`performLayout` method

- `IlvGraphLayout` class **30, 134**

position

- Tree Layout **58**

position index

- Hierarchical Layout **99**

preserve fixed links parameter

- Bus Layout **122**
- Hierarchical Layout **90**
- Orthogonal Link Layout **111**

- Random Layout **117**
- Tree Layout **56**

preserve fixed nodes parameter

- Bus Layout **122**
- Hierarchical Layout **90**
- Random Layout **118**
- Tree Layout **56**

R

radial layout mode

- description **72**

random generator seed value parameter

- Random Layout **118**

Random Layout

- applicable graph types **115**
- code sample **116**
- description **116**
- features **116**
- generic parameters **117**
- limitations **116**
- link style parameter **118**
- parameters **117**
- sample drawing **115**

reading a layout report **135**

reference transformer **146, 147**

reference transformers **145**

reference views **146**

refining a layout

- Hierarchical Layout **104**

related documentation **11**

`removeAllLayers`

- `IlvLayoutGraphicFilter` class **143**

`removeGraphModelListener` method

- `IlvGraphModel` class **140**

`removeLayer`

- `IlvLayoutGraphicFilter` class **143**

report information **135**

report, layout **133**

`reshapeLink` method

- `IlvGraphModel` class **139**

root node parameter

- Tree Layout **57**

root node parameter options

- Tree Layout **57**

S

same shape for multiple links parameter, Orthogonal Link Layout **114**

sample application **40**

semi-automatic layout **27**

setAlignment method
IlvTreeLayout class **63, 65**

setAllowedNumberOfIterations method
IlvOrthogonalLinkLayout class **113**

setAllowedTime method
IlvGraphLayout class **32**
IlvTreeLayout class **77**

setAnimate method
IlvGraphLayout class **32**

setAspectRatio method
IlvTreeLayout class **75, 77**

setBranchOffset method
IlvTreeLayout class **68, 75**

setBus method
IlvBusLayout class **124**

setConnectorStyle method
IlvHierarchicalLayout class **96**

setEastWestNeighboring method
IlvTreeLayout class **81**

setFilter
IlvLayoutGraphicFilter class **143**

setFixed method
IlvGraphLayout class **34, 35**

setFlowDirection method
IlvHierarchicalLayout class **91**
IlvTreeLayout class **61**

setGlobalAlignment method
IlvTreeLayout class **62, 64**

setGlobalLinkStyle method
IlvHierarchicalLayout class **94**
IlvTreeLayout class **66**

setHorizontalLinkOffset method
IlvHierarchicalLayout class **100**

setHorizontalNodeLinkOffset method
IlvHierarchicalLayout class **100**

setHorizontalNodeOffset method
IlvHierarchicalLayout class **100**

setHorizontalOffset method
IlvBusLayout class **126**

setIndex method
IlvBusLayout class **123**

setLayoutMode method
IlvTreeLayout class **59, 60, 70, 73, 74, 78, 79**

setLayoutRegion method
IlvGraphLayout class **33**

setLevelJustification method
IlvHierarchicalLayout class **92**
IlvTreeLayout class **72**

setLinkPriority method
IlvHierarchicalLayout class **97**

setLinkStyle method
IlvBusLayout class **125**
IlvHierarchicalLayout class **95**
IlvOrthogonalLinkLayout class **113**
IlvRandomLayout class **118**
IlvTreeLayout class **67**

setLinkToLinkCrossingPenalty method
IlvOrthogonalLinkLayout class **114**

setLinkToNodeCrossingPenalty method
IlvOrthogonalLinkLayout class **114**

setMargin method
IlvBusLayout class **127**

setMarginOnBus method
IlvBusLayout class **127**

setMaxChildrenAngle method
IlvTreeLayout class **76**

setMinFinalSegmentLength method
IlvOrthogonalLinkLayout class **112**

setOffset method
IlvOrthogonalLinkLayout class **112**

setOrdering method
IlvBusLayout class **122**

setOrthForkPercentage method
IlvTreeLayout class **69**

setOverlapPercentage method
IlvTreeLayout class **69**

setParentChildOffset method
IlvTreeLayout class **68, 75**

setPosition method
IlvTreeLayout class **58**

setPreference method
IlvTreeLayout class **58**

setProperty method
IlvGraphModel class **140**

- setReferenceTransformer
 - IlvLayoutGraphicFilter class **147**
- setReferenceView
 - IlvLayoutGraphicFilter class **146**
- setRoot method
 - IlvTreeLayout class **57**
- setRootPreference method
 - IlvTreeLayout class **58**
- setSameShapeForMultipleLinks method
 - IlvOrthogonalLinkLayout class **114**
- setSeedValueForRandomGenerator method
 - IlvGraphLayout class **36**
- setSiblingOffset method
 - IlvTreeLayout class **68, 75**
- setSpecNodeLevelIndex method
 - IlvHierarchicalLayout class **98**
- setSpecNodePositionIndex method
 - IlvHierarchicalLayout class **99**
- setTipOverBranchOffset method
 - IlvTreeLayout class **68**
- setUseDefaultParameters method
 - IlvGraphLayout class **37**
- setVerticalLinkOffset method
 - IlvHierarchicalLayout class **100**
- setVerticalNodeLinkOffset method
 - IlvHierarchicalLayout class **100**
- setVerticalNodeOffset method
 - IlvHierarchicalLayout class **100**
- setVerticalOffsetToLevel method
 - IlvBusLayout class **126**
- setVerticalOffsetToPreviousLevel method
 - IlvBusLayout class **126**
- setWestEastNeighboring method
 - IlvTreeLayout class **81**
- spacing parameters
 - for experts **69**
 - free layout mode **67**
 - Hierarchical Layout **99**
- specifying
 - reference transformer **147**
- static layout **27**
- supportsAllowedTime method
 - IlvGraphLayout class **32**
- supportsAnimation method
 - IlvGraphLayout class **32**

- supportsLayoutRegion method
 - IlvGraphLayout class **34, 152**
- supportsPreserveFixedLinks method
 - IlvGraphLayout class **34**
- supportsPreserveFixedNodes method
 - IlvGraphLayout class **35**
- supportsRandomGenerator method
 - IlvGraphLayout class **36**
- symmetries layout criteria **26**

T

- time, stop computation algorithms **31**
- tip-over alignment
 - free layout mode **64**
- tip-over layout modes
 - description **77**
- Tree Layout
 - algorithm description **53**
 - applicable graph types **51**
 - application domains **51**
 - code sample **55**
 - compass directions **58**
 - features **52**
 - free layout mode **59**
 - generic parameters **56**
 - incremental changes **83**
 - interactive editing **83**
 - layout mode **59**
 - learning about **49**
 - level layout mode **70**
 - limitations **52**
 - parameters **55**
 - position **58**
 - radial layout mode **72**
 - retrieving link categories **81**
 - root node parameter **57**
 - root node parameter options **57**
 - sample drawings **50**
 - specific parameters **56**
 - specifying the order of children **83**
 - tip-over layout modes **77**
 - tips for experts **79**
 - tips for experts, east-west neighbors **80**

U

unfixAllLinks method

 IlvGraphLayout class **34**

unfixAllNodes method

 IlvGraphLayout class **35**

V

vertical offset parameter, Bus Layout **126**

vertical offset to previous level parameter, Bus Layout **126**