## Preliminaries

The src/code folder contains Java classes. JavaScript classes are generated from Java classes using GWT and Google closure compiler via ant script (build.xml). See file readme-sources.txt for licensing information on the tools/libraries used.

The initial low-level Java code lays in the package lv.lumii.layoutengine. The documentation (JavaDoc) for it can be found in the folder doc/internal\_javadoc.

Sergejs Kozlovičs developed two wrappers over the initial low-level Java code. These wrappers lay in the packages lv.lumii.diagramlayout and lv.lumii.dialoglayout. It is advised to use these wrappers instead of the initial low-level code (unless you have certain reasons to do otherwise). The wrappers provide two Java classes:

* IMCSDiagramLayout for laying out graph diagrams, and
* IMCSDialogLayout for laying out dialog windows.

The client package contains glue code for generating JavaScript from Java. The JavaScript code is compiled as imcs\_layoutengine.js script, which can be embedded into your web page. In order to include it in your code use the following HTML string:  
<script type="text/javascript" src="imcs\_layoutengine.js"></script>

The script provides two JavaScript classes with the same names as the corresponding Java classes: IMCSDiagramLayout and IMCSDialogLayout.

The documentation below explains the API of these classes in JavaScript syntax. If you prefer to use the Java version, the syntax is similar (id’s are of type long; x, y, width and height are doubles in Java; refer to the Java code for more details).

## IMCSDiagramLayout Introduction

You can initialize the corresponding instance using the JavaScript new operator:  
var layout = new IMCSDiagramLayout();

If you do not pass any constructor parameters, or pass the string “UNIVERSAL”, then the universal layout algorithm will be used. Other layout algorithms are “SYMMETRIC”, “VERTICAL” (for an edge a→b the layout algorithm will return y\_max[a]≤y\_min[b]), “INVERSE\_VERTICAL”, “HORIZONTAL”, and “INVERSE\_HORIZONTAL”. Pass the corresponding string to the constructor:

var layout = new IMCSDiagramLayout(“UNIVERSAL”);

Then you can add/remove boxes and lines (using appropriate methods) as well as to call either the arrangeIncrementally(), or arrangeFromScratch() method. They both return the calculated layout information.

An important feature of the JavaScript implementation is that you specify integer IDs for boxes and lines. For instance, these IDs may be equal to database primary keys or references to model repository objects.

## IMCSDiagramLayout Methods

**IMCSDiagramLayout.addBox(boxId, x, y, w, h)**adds a box with the given id (some integer), x, y, width, and height to the layout;  
during the initial layout, when no coordinates of boxes are known, you may specify x=y=0;  
if a new box is added to the existing layout, use the current mouse position for x and y;  
returns whether the operation succeeded;

**IMCSDiagramLayout.addLine(lineId, srcId, tgtId, options)**adds an orthogonal line connecting the two boxes with the given id-s to the layout; optional options are in the following format (default values are specified):  
{  
 lineType: “ORTHOGONAL”, // or “POLYLINE”, or “STRAIGHT”  
 startSides: 15, // =parseInt('1111', 2)=0b1111, see bit mask values below  
 endSides: 15, // =parseInt('1111', 2)=0b1111, see bit mask values below  
}

Bit mask values:  
0b0001: top (if y=0 is on top) = min\_y = 1  
0b0010: right = 2  
0b0100: bottom (if y=0 is on top) = max\_y = 4  
0b1000: left = 8

returns whether the operation succeeded;

**IMCSDiagramLayout.addLineLabel(labelId, lineId, w, h, placement)**adds the label with the given labelId to the line with the given lineId; the size of the label is w\*h; the position is specified in the placement attribute and is one of the following values:

* "start-left", i.e., left, if we follow the line direction (from start to end), near the start
* "start-right",
* "end-left", i.e., left, if we follow the line direction, near the end
* "end-right",
* "middle-left",
* "middle-right".

returns whether the operation succeeded;

**IMCSDiagramLayout.removeLine(lineId)**removes the given line from the layout;  
the layout is not re-arranged;  
returns whether the operation succeeded;

**IMCSDiagramLayout.removeBox(boxId)**remove the box as well as the incident lines from the layout;  
the layout is not re-arranged;  
returns the array of id-s of the removed lines (perhaps, an empty array);  
on error, returns false

**IMCSDiagramLayout.moveBox(boxId, newX, newY)**Warning: arrange() or arrangeFromScratch() must be called some time before, if you use a hierarchical layout other from “VERTICAL”!  
sets new desired coordinates for the box with the given id;  
the layout is not re-arranged (call arrange() after the desired coordinates of all the desired boxes are set);  
returns whether the operation succeeded;

**IMCSDiagramLayout.resizeBox(boxId, w, h)**sets new desired dimensions for the box with the given id;  
the layout is not re-arranged (call arrange() after the desired dimensions of all the desired boxes are set);  
returns whether the operation succeeded;

**IMCSDiagramLayout.moveLine(lineId, srcId, tgtId, points)**Warning: arrange() or arrangeFromScratch() must be called some time before, if you use a hierarchical layout other from “VERTICAL”!  
sets new line start and end boxes and (optionally) line points;  
the points are specified as an array of objects with the x and y attributes;  
the layout is not re-arranged (call arrange() after all desired manipulations are called);  
returns whether the operation succeeded;

**IMCSDiagramLayout.arrangeIncrementally()**  
arranges the diagram taking into a consideration recently added elements and trying to preserve existing coordinates;  
returns an objects with the "boxes", "lines", and "labels" maps containing information about the layout:

* the boxes map is in the form <id> -> {x, y, width, height};
* the lines map is in the form <id> -> [ {x:x1,y:y1}, {x:x2,y:y2}, ... ]
* the labels map is in the form <id> -> {x, y, width, height};

**IMCSDiagramLayout.arrangeFromScratch()**  
arranges the diagram from scratch not preserving existing coordinates;  
returns an objects with the "boxes", "lines", and "labels" maps containing information about the layout:

* the boxes map is in the form <id> -> {x, y, width, height};
* the lines map is in the form <id> -> [ {x:x1,y:y1}, {x:x2,y:y2}, ... ]
* the labels map is in the form <id> -> {x, y, width, height};

## IMCSDialogLayout Introduction

IMCSDialogLayout is a pseudo class that implements IMCS layout algorithms for laying out dialog windows in JavaScript.

Additional information on how to use the layout or on the meaning of certain parameters can be found in the papers:

1. Kozlovics, S.: A Dialog Engine Metamodel for the Transformation-Driven Architecture. In: Scientific Papers, University of Latvia. vol. 756, pp. 151-170 (2010)
2. Kozlovics, S.: Calculating The Layout For Dialog Windows Specified As Models. In: Scientific Papers, University of Latvia. vol. 787, pp. 106-124 (2012)

You can initialize the corresponding instance using the JavaScript new operator:  
var dialogLayout = new IMCSDialogLayout(callback);

The callback parameter is a JavaScript object, which defines certain functions for getting information about dialog components to be layed out. Each dialog component is referenced by an id (the rComponent parameter, which is a natural number). The callback functions are as follows:

**getAnchor(rComponent)**

Specifies what coordinates are passed back to the layout callback of the given component. Possible return values are:

* "parent" (parent’s x and y are considered zeroes, when laying out child elements)
* "sibling" (component’s x and y are relative to the previous sibling)
* "zero" (component’s x and y are relative to the form’s left-top corner, which has coordinates (0;0) ).
* ? "grandparent" (grandparent’s x and y are considered zeroes, when laying out grandchild elements)

Default is “parent”.

**load(rComponent)**

Creates a HTML (dojo, jQuery, or other) element for the given rComponent (but not for its children). The load function will be called for all child components recursively by IMCSDialogLayout.

If the component is being loaded asynchronously, call IMCSDialogLayout.loadStarted(rComponent) during load() and call IMCSDialogLayout.loadFinished(rComponent) when the component is fully loaded, e.g.,

callback.load = function load(rComponent) {

dialogLayout.loadStarted(rComponent);

asyncFunctionToLoadComponent(rComponent, function(callback\_args) {

dialogLayout.loadFinished(rComponent);

});

}

**getChildren(rComponent)**

Returns an array of numbers, which identify child component of the given rComponent.

**getBounds(rComponent)**

Returns an object contacting information of the component bounds in the following fields.

For leaf components (without children):

* minimumWidth
* maximumWidth
* preferredWidth
* minimumHeight
* maximumHeight
* preferredHeight
* leftMargin
* rightMargin
* topMargin
* bottomMargin

For containers (components that may contain other components) the following fields are also considered:

* leftPadding
* rightPadding
* topPadding
* bottomPadding
* horizontalSpacing
* verticalSpacing
* horizontalAlignment ( “LEFT”, “RIGHT”, or “CENTER”)
* verticalAlignment ( “TOP”, “BOTTOM”, or “CENTER”)

**getHorizontalRelativeInfo(rComponent)**

Returns an object contacting relative sizes w.r.t. to other components in the same group. See also getHorizontalRelativeInfoGroup/getVerticalRelativeInfoGroup, which define relative groups.

The fields are:

* minimumRelativeWidth
* preferredRelativeWidth
* maximumRelativeWidth

**getVerticalRelativeInfo(rComponent)**

Returns an object contacting relative sizes w.r.t. to other components in the same group. See also getHorizontalRelativeInfoGroup/getVerticalRelativeInfoGroup, which define relative groups.

The fields are:

* minimumRelativeHeight
* preferredRelativeHeight
* maximumRelativeHeight

**getHorizontalRelativeInfoGroup(rComponent)**

**getVerticalRelativeInfoGroup(rComponent)**

Return an integer representing a group. Relative infos are attached to groups defined by these integers. The same integer can be returned for horizontal and vertical groups to specify horizontal and vertical dimensions that relate on each other.

**getLayoutName(rComponent)**

Returns null, if a component is a leaf-component, which does not contain children (e.g., a button or an input field). Otherwise, returns one of the following values:

* "VerticalBox"
* "HorizontalBox"
* "VerticalScrollBox"
* "HorizontalScrollBox"
* "ScrollBox"
* "Column"
* "Row"
* "Stack"

**layout(rComponent, x, y, w, h)**

When the layout has been calculated, this function is called for each dialog component to set its physical coordinates and dimensions. The numbers are relative to the parent. For the root forms x=y=0.

**destroy(rComponent)**

When a form (or a part of the form) is unloaded (e.g., during refresh), the corresponding components in the subtree must be destroyed. This function is called for such components to physically remove them (e.g., from the HTML DOM).

## IMCSDialogLayout Methods

**IMCSDialogLayout.loadAndLayout(rForm)**

Loads the component tree starting from the root component rForm and performs the initial layout (e.g., the layout callback function will be called for each component).

**IMCSDialogLayout.refreshAndLayout (rRootComponent, formWidth, formHeight)**

When some component subtree is changed, this function may be called to re-load that subtree and to perform incremental layout of the dialog window. If form width and height have been changed (e.g.,during resize), they may also be specified (thus, the form will try to be of that size).