Model Visualization

Markus Voelter & others,
voelter@acm.org

Version 0.1

Introduction

Things that are described graphically are easier to comprehend than textual descriptions, right? What is most important regarding comprehensibility is the alignment of the concepts that need to be conveyed with the abstractions in the language. A well-designed textual notation can go a long way. Of course, for certain kinds of information, a graphical notation is better: relationships between entities, the timing/sequence of events or some kind of signal/data flow. On the contrary, rendering expressions graphically is a dead end. When deciding about a suitable notation, you might want to consider the following two forces: in most (but not all!) tool environments, editors for textual notations (incl. code completion, syntax highlighting and the like) are much easier to build and evolve than really user-friendly and scalable graphical editors. Also, instead of using full-blown graphical editing, you might want to consider textual editing plus graphical visualization (read only, auto-layout, created via transformation). In many cases, this is absolutely good enough.

The model visualization package contains tools for visualization with graphviz and ZEST.

ZEST

ZEST is a visualization framework implemented on top of GEF. The model visualization project adds the following ingredients:

- a meta model for describing graphs

- a textual language for expressing instances of that meta model

- a generic viewer, that renders instances of the meta model in an Eclipse view

The Meta Model & Language

The textual DSL has been built with oAW Xtext. You can find it in the following projects or plugins, respectively:

org.openarchitectureware.vis.graphmm
org.openarchitectureware.vis.graphmm.editor

Manually Written Graph Example

The following is an example of a simple graph built with the language. The meaning of the various constructs should be obvious. More documentation will be added later.

```
prolog {
  iconbasepath "platform:/plugin/de.voelter.zest.example/icons/";
}

graph g1 {
  nodes {
        s1 label "S1" tooltip "S1 Tooltip" icon "icon1.gif" category "a"
              userData
                      test : text = "This is a test";
                      test2 : int = "2";
                      test3 : boolean = "true";
              ;
        s2 label "S2" icon "icon2.gif" category "a";
        s3 label "shift click here" source
"platform:/resource/de.voelter.zest.example/src/de/voelter/zest/example/test.gmm,
100,120";
        s4 label "S4" linecolor red category "b";
        s5 label "S5" linecolor gray category "b";
        s6 label "S6" linecolor darkBlue fillcolor yellow;
        s7 label "S7" textcolor green fillcolor blue;
        s8 label "S8" textcolor darkBlue;
        s9 label "S9" textcolor red;
        container sc label "SC"
              contained graph scchild {
                      nodes {
                              sc1 label "sc1";
                              sc2 label "sc2";
                      }
                      edges {
                              sc1 -> sc2;
                              sc1 -> s6;
                      }
              }
        ;
  }

  edges {
        s1 -- sc label "containeredge";
        s1 -- s2 label "Hallo!" tooltip "s1 -> s2"
              icon "icon2.gif" color black weight -50;
        s2 -- s3 tooltip "s2 -> s3" color cyan weight 0;
        s3 -> s1 tooltip "s3 -> s1" color darkGreen weight 50;
        s4 -> s5 width 3 style dotted;
        s1 -> s4 width 4 style dashDotted;
        s1 -> s5 width 2 category "x";
        s1 -> s5 width 1 category "y";
        s5 -> s6;
  }
}

graph g2 {
  nodes {
        n1 label "n1";
        n2 label "n2";
  }

  edges {
        n1 -> n2 label "label";
  }
}
```

Transformation towards the meta model

The typical use case for visualization is not to manually type a graph using the textual DSL. Instead, usually, you will write a model to model transformation that maps your own domain specific language to the graph meta- model. The following is an example of the transformation that visualizes Ecore. As you can see, we actually create two different graphs.

```
import ecore;
import graphmm;

extension graphmmlib;

Object top(EPackage p):
  toCollection(p);

create GraphCollection toCollection( EPackage p ):
  graphs.add( toAssocGraph(p)) ->
  graphs.add( toInheritGraph(p));

create Graph toAssocGraph(EPackage p):
  setName( "assoc & attr" ) ->
  nodes.add( p.toNode("ass") ) ->
  nodes.addAll( p.eClassifiers.typeSelect(EClass).toNode("ass") ) ->
  edges.addAll( p.eClassifiers.typeSelect(EClass).toEdge("ass") ) ->
  edges.addAll( p.eClassifiers.typeSelect(EClass).eReferences.toEdge("ass") ) ->
  nodes.addAll( p.eClassifiers.typeSelect(EClass).eAttributes.toNode("ass") ) ->
  edges.addAll( p.eClassifiers.typeSelect(EClass).eAttributes.toEdge("ass") );

create Graph toInheritGraph(EPackage p):
  setName( "inheritance" ) ->
  addLayoutHint( "zestlayout", "tree" ) ->
  nodes.add( p.toNode("i") ) ->
  nodes.addAll( p.eClassifiers.typeSelect(EClass).toNode("i") ) ->
  edges.addAll( p.eClassifiers.typeSelect(EClass).iterateSuperTypes("i") );

iterateSuperTypes(EClass c, Object tkn):
  c.eSuperTypes.toInheritanceEdge(c, tkn);

create SimpleDirectedEdge toInheritanceEdge( EClass super,
                           EClass sub, Object tkn ):
  setSource( super.toNode(tkn) ) ->
  setTarget( sub.toNode(tkn) );

create SimpleNode toNode( EPackage p, Object tkn ):
  setLabel( p.name ) ->
  setCategory("package") ->
  setFillcolor( ColorEnum::white);

create SimpleNode toNode( EClass cls, Object tkn ):
  setLabel( cls.name ) ->
  setFillcolor( cls.abstract ? ColorEnum::white : ColorEnum::lightGray );

create SimpleNode toNode( EAttribute attr, Object tkn ):
  setCategory( "attributes" ) ->
  setLabel( attr.name ) ->
  setFillcolor( ColorEnum::lightGreen );

create SimpleDirectedEdge toEdge( EReference ref, Object tkn ):
  setLabel( ref.name ) ->
  setCategory("associations") ->
  setWeight( ref.containment ? 50 : 0 ) ->
  setWidth( ref.containment ? 2 : 1 ) ->
  setSource( ((EClass)ref.eContainer).toNode(tkn) ) ->
  setTarget( ((EClass)ref.eType).toNode(tkn) );

create SimpleDirectedEdge toEdge( EAttribute a, Object tkn ):
  setCategory( "attributes" ) ->
  setWeight( 100  ) ->
  setSource( ((EClass)a.eContainer).toNode(tkn) ) ->
  setTarget( a.toNode(tkn) );
```

```
create SimpleDirectedEdge toEdge( EClass c, Object tkn ):
  setCategory( "package" ) ->
  setSource( ((EPackage)c.eContainer).toNode(tkn) ) ->
  setTarget( c.toNode(tkn) );
```
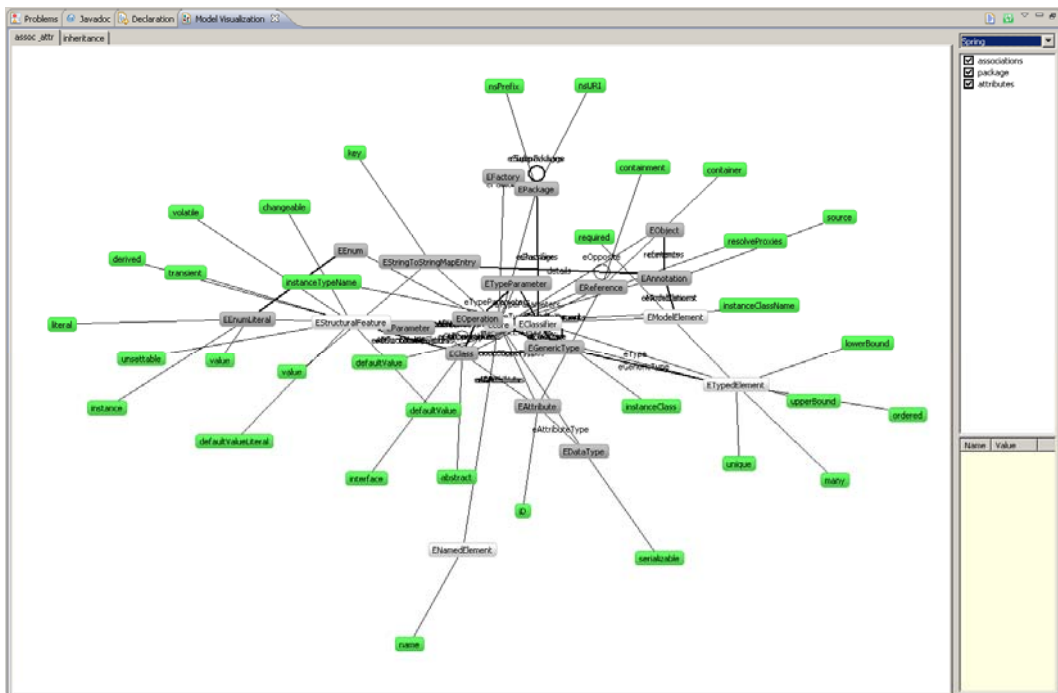
The Viewer

The viewer can be found in the following plug-in: *org.openarchitectureware.vis.zest.viewer.* Open the view via *Window/Show View/openArchitectureWare/Model Visualization.*

The generic viewer is able to visualize an instance of the meta model mentioned above. Currently, it is implemented in the following way:

- using the button or the menu entry at the top right side of the view, you run an openArchitectureWare workflow.

- The workflow can do basically anything, as long as it puts an instance of the above meta model into a workflow slot called *graphmodel.*

So, the workflow can either just parse a graph defined using the textual language, or it can run a model to model transformation as the one shown above.

The following is a screenshot of the view as it renders the result of the above model transformation.

The viewer currently provides the following features:

- changing the layout between the various layouts supported by ZEST

- show several graphs using different tabs

- hiding certain types of elements in the model, based on the *categories* field in the meta model

- selecting elements as well as related elements

- showing any number of name-value pairs associated with nodes or connections

- if notes contain data in the *source* field, the viewer can go back to that location in the source code

Future Work

The current state of the ZEST viewer is basically a prototype. To make it really useful, the following improvements should be implemented, and I am looking for people who help with that.

- The layout must be improved, for example by reducing the metaphor of nodes having negative electric charge and connections acting as springs. Using this approach, the layout can be implemented as a numerical simulation of the forces between the nodes.

- To avoid disorientation in a new rendering of the graph, it would be useful to be able to pin down the position of certain nodes. When the graph is rerendered her, those nodes remain exactly that position.

- If a node contains children, a feature that opens up a new graph showing that inside of the note in a new tab would be useful.

- The ability to add custom figures to test graphs would be important. Especially the ability to have compartmented rectangles (think: UML Class) would be worthwhile.

- Currently, the complete graph is rendered. It would be helpful to only render a part of it, and then when selecting a menu entry on a note, expand the graph dynamically. This way, you couldn't explore your way through the graph.

Of course, many more features are possible and useful, if you have suggestions or want to help please let me know: *voelter@acm.org*
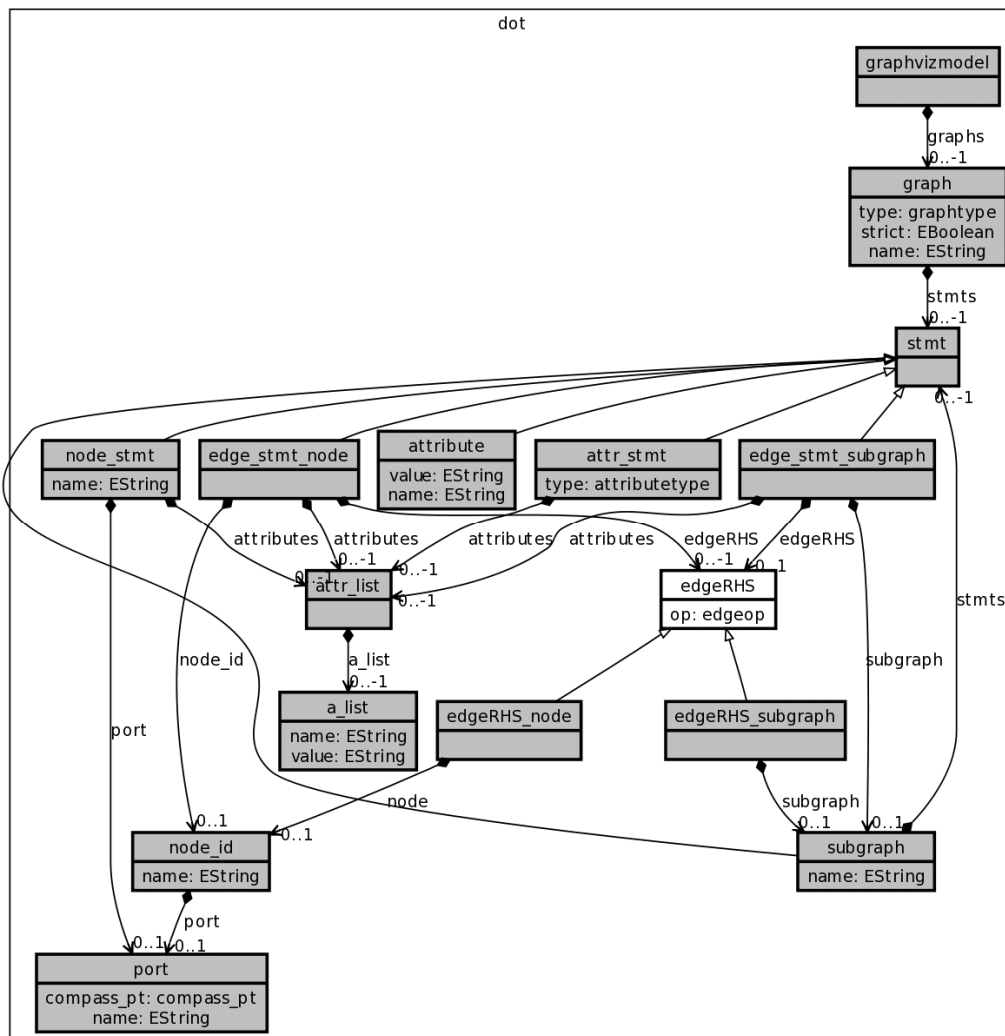
Graphviz

Graphviz is an open source tool that rendes images based on a textual description of a graph. It is quite powerful and widely used. It is available on many platforms. Please download it from www.graphviz.org and install it on your machine.

General Approach

In order to visualize a model, you have to write a model to model transformation from your DSL meta model to the meta model of the input language for Graphviz. The oAW Graphviz support comes with this meta model, it's name is *dot*. After running the transformation for a given input model, you have to call a code generator (also supplied with the oAW Graphviz integration plugins) that generates the text representation of the dot file. This generator also generates a batch file `processdot.bat` (and a shell script `processdot.sh`) that contains calls to the `dot.exe` renderer to generate the actual GIF images. To make that batch file work, make sure you set the GRAPHVIZ_BIN environment variable to point to the `bin` directory of your Graphviz installation.

Dot language reference

The metamodel of the dot language is shown in the figure below. The diagram was generated with the Graphviz visualization itself, using the transformation described in the previous section.

dot

graphvizmodel

graphs
0..-1

graph
type: graphtype
strict: EBoolean
name: EString

stmts
0..-1

stmt
0..-1

node_stmt
name: EString

edge_stmt_node

attribute
value: EString
name: EString

attr_stmt
type: attributetype

edge_stmt_subgraph

attributes attributes   attributes attributes   edgeRHS   edgeRHS
0..-1   0..-1   0..-1   0..1

attr_list
0..-1

edgeRHS
op: edgeop

stmts

node_id

port

a_list
0..-1

a_list
name: EString
value: EString

edgeRHS_node

edgeRHS_subgraph

subgraph

node

0..1   0..1

subgraph
0..1   0..1

node_id
name: EString

subgraph
name: EString

port
0..1   0..1

port
compass_pt: compass_pt
name: EString

Types

This section describes the types of the dot language. Features are denoted by a qualifier "T", which has the following meaning:

Abbreviation	Meaning
P	Optional property.
P1	Mandatory property.
R	0..1 reference.
R1	1..1 reference.
R*	0..* reference.
R+	1..* reference.
EP	An extension function with zero or one argument. It can be treated like a property except that the function must called with brackets(). The function can be called on elements of the specified type. The extension is provided by dotlib.
O	An extension function that takes at least one additional argument besides the caller argument. This function behaves like an operation. The extension is provided by

dotlib.

dot::graphvizmodel

This is the root element containing graph instances.

T	name	type / arguments	description
R*	graphs	dot::graph	Contained graphs.
O	addGraph	dot::graph	The graph instance to add to the model.
O	addGraphs	Collection[dot::graph]	A collection of graphs to add to the model.

dot::graph

This element represents a named graph instance, Supertype: *Eobject*

T	name	type / arguments	description
P1	name	String	The name of the graph.
P1	type	graphtype	Specifies whether a graph is directed (graphtype::digraph) or undirected (graphtype::graph).
P	strict	EBoolean	A graph may be declared a strict digraph. This forbids the creation of self-arcs and multi-edges.
R*	stmts	dot::stmt	Statements defining this graph.
O	setFont	String	Sets the default font for the graph.
O	addStatement	dot::stmt	Adds a statement to the graph.
O	addStatements	List[dot::stmt]	Adds the given statements to the graph.
EP	filename	String	The filename for the resulting dot file: "<name>.dot"
O	setAttribute	String name	Sets an edge attribute.
		String value	The attribute's value.

dot::stmt

This is the base class for statement types.

Supertype: *EObject*

Subtypes: *edge_stmt_node, edge_stmt_subgraph, node_stmt, attribute, attr_stmt, subgraph*

Abstract: *true*

dot::node_stmt

Represents a node in the graph, Supertype: *stmt*

T	name	type	description
P	name	String	Node name.
R	port	dot::port	
R*	attributes	dot::attr_list	Attributes of this element.
O	setStyle	String	Set style property of the node.
O	setFont	String	Sets the font for the node label.
O	setFontSize	String	Sets the font size.
O	setLabel	String	Sets the node label.
O	setLabel	List[String]	Sets the node label which consists of several sections. This helps to create record layouts.
O	setShape	String	Sets the node shape style.
O	setName	String	Sets the node's name.
O	setFillColor	String	Sets the fill color.
O	setAttribute	String name	Sets an edge attribute.
		String value	The attribute's value.

dot::edge_stmt_node

This is an edge, Supertype: *stmt*

T	name	type / arguments	description
R	node_id	node_id	Node identifier.
R*	edgeRHS	dot::edgeRHS	
R*	attributes	dot::attr_list	Attributes of this element.
O	setArrowHead	String	Sets the arrow head style.
O	setArrowTail	String	Sets the arrow tail style.
O	setStyle	String	Sets the edge's style property.
O	setWeight	String	Sets the line weight.
O	setColor	String	Sets the line and font color.
O	setFontcolor	String	Sets the font color.
O	setFont	String	Sets the font.
O	setFontSize	String	Sets the font size.
O	setLabel	String	Sets the edge label.
O	setHeadLabel	String	Sets the label for the edge's head.
O	setTailLabel	String	Sets the label for the edge's tail.

O	setLineColor	String	Sets the line color.
O	setAttribute	String name	Sets an edge attribute.
		String value	The attribute's value.

dot::attribute

Supertype: *stmt*

T	name	type	description
P	name	String	Attribute name.
P	value	String	Holds the attribute's value.

dot::attr_stmt

Supertype: *stmt*

T	name	type	description
P	type	dot::attributetype	Attribute type.
R*	attributes	dot::attr_list	Attributes of this element.

dot::edge_stmt_subgraph

Supertype: *stmt*

T	name	type	description
R	subgraph	dot::subgraph	The referenced subgraph.
R	edgeRHS	dot::edgeRHS	
R*	attributes	dot::attr_list	Attributes of this element.

dot::attr_list

Supertype: *EObject*

T	name	type	description
R*	a_list	dot::a_list	

dot::a_list

Supertype: *EObject*

T	name	type	description
P	name	String	List name.
P	value	String	The value.

dot::edgeRHS

Represents the target an edge is pointing to.

Supertype: *EObject*

T	name	type	description
R	op	dot::edgeop	Edge operation.

dot::edgeRHS_node

Supertype: edgeRHS

T	name	type	description
R	node	dot::node_id	Node identifier.

dot::edgeRHS_subgraph

Supertype: *edgeRHS*

T	name	type	description
R	subgraph	dot::subgraph	A subgraph.

dot::subgraph

Supertype: *stmt*

T	name	type / arguments	description
P	name	String	The name of the subgraph.
R*	stmts	dot::stmt	Statements defining this graph.
O	setName	String	Sets the subgraph's name.
O	addStatement	dot::stmt	Adds a statement to the subgraph.
O	addStatements	List[dot::stmt]	Adds the given statements to the subgraph.
O	setLabel	String	Sets the subgraph's label.
O	setAttribute	String name	Sets an edge attribute.
		String value	The attribute's value.

dot::node_id

Supertype: *EObject*

T	name	type	description
P	name	String	Node identifier value.
R	port	dot::port	Node port.

dot::port

Supertype: *EObject*

T	name	type	description
P	name	String	The name of the port.
P	compass_pt	dot::compass_pt	Where the port should be placed at the

element.	

dot::edgeop

Context: *edgeRHS#op*

name	description
directed	Directed edge.
undirected	Undirected edge.

dot::graphtype

Context: *graph#graphtype*

name	description
directed	Directed graph.
undirected	Undirected graph.

dot::attributetype
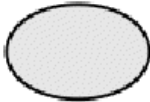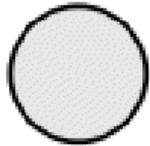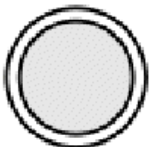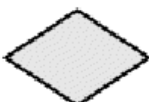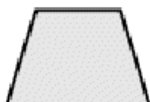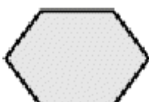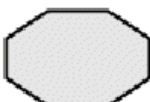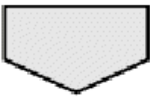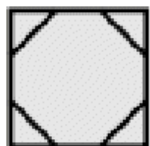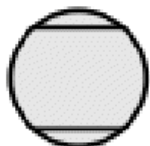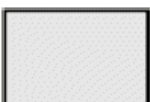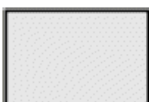
Context: *attr_stmt#type*

name	description
graph	Attribute denotes a graph.
node	Attribute denotes a node.
edge	Attribute denotes an edge.

dot::compass_pt

name	description
north	Direction north.
northeast	Direction northeast.
east	Direction east.
southeast	Direction southeast.
south	Direction south.
southwest	Direction southwest.
west	Direction west.
northwest	Direction northwest.

Shapes

box	polygon	ellipse	circle
doublecircle	point	egg	triangle
plaintext	diamond	trapezium	parallelogram
house	hexagon	septagon	octagon
doubleoctagon	tripleoctagon	invtriangle	invtrapezium
invhouse	Mdiamond	Msquare	Mcircle
rect	rectangle	none	

Arrows
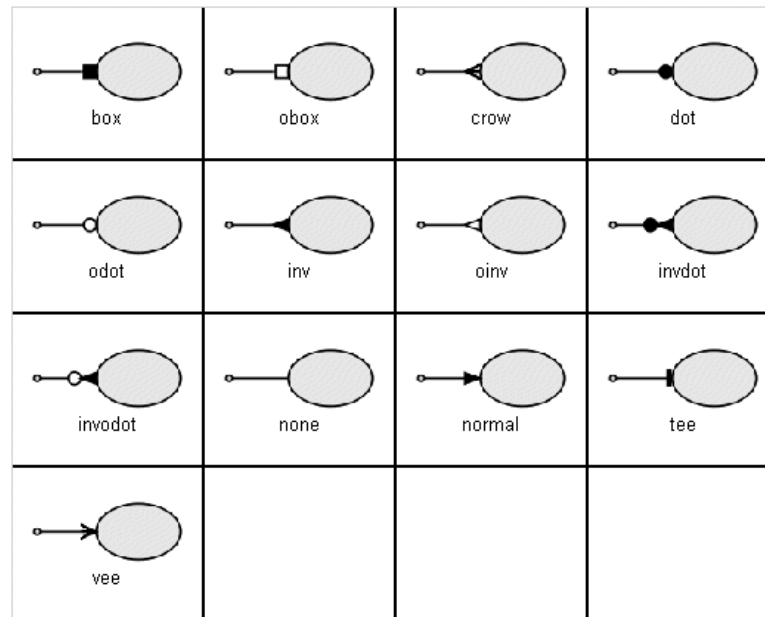
Output formats

Name	Description	Name	Description
bmp	Windows Bitmap Format	canon	
dot	DOT	xdot	DOT
dia	Dia format	eps	Encapsulated PostScript
fig	FIG	gtk	GTK canvas
gd **gd2**	GD/GD2 formats	gif	GIF
hpgl	HP-GL/2	ico	Icon Image File Format
imap **imap_np** **cmapx** **cmapx_np**	Server-side and client-side imagemaps	jpg jpeg jpe	JPEG
mif	FrameMaker MIF format	mp	MetaPost
pcl	PCL	pdf	Portable Document Format (PDF)
pic	PIC	plain plain-ext	Simple text format
png	Portable Network Graphics format	ps	PostScript
ps2	PostScript for PDF	svg svgz	Scalable Vector Graphics
tga	Truevision Targa Format	tif	TIFF (Tag Image File

	(TGA)	tiff	Format)
vml **vmlz**	Vector Markup Language (VML)	vrml	VRML
vtx	Visual Thought format	wbmp	Wireless BitMap format
xlib	Xlib canvas		

Workflow configuration

To invoke the transformation (and subsequently generate the dot text files) you need to write a workflow that calls into the `model2dotfile.oaw` workflow supplied with the oAW Graphviz integration.

```
<workflow>

  <property name="modelFile"
   value="platform:/resource/org.openarchitectureware.graphviz.ecoredemo/
              model/Ecore.ecore"/>
  <property name="targetDir" value="src-gen/" />

  <component id="read" class="org.eclipse.mwe.emf.Reader">
    <useSingleGlobalResourceSet value="true"/>
    <modelSlot value="model"/>
    <uri value="${modelFile}"/>
  </component>

  <cartridge file="model2dotfile.oaw"
    targetDir="${targetDir}"
    topFunctionCallExpression="ecore2dot::toGraphVizmodel(model)"
    inheritAll="true"/>
</workflow>
```

The `model2dotfile.oaw` workflow has some configuration parameters described in the table below.

name	description	example
targetDir	Output directory of resulting `.dot` files and `processdot[.bat|.sh ]`	src-gen
dotTargetDir	Output directory of images produced by Graphviz when calling the generated `processdot.[bat|sh]` command.	src-gen/images
topFunctionCall -Expression	This is the qualified path to the Xtend function to call to perform the M2M transformation.	ecore2dot::toGraphViz model(model)
pathToDotExe	The absolute path to the dot executable, only the directory the executable	/usr/local/ graphviz-2.12/bin/

	is contained in. Note that the default will only work on Windows. If the dot executable is already on your system path you can set this property to an empty string.
	Default: `%GRAPHVIZ_BIN%/`
outputFormat	Graphviz output format. Valid options see output formats.
	Default: gif

You may want to include the `SystemCommand` workflow component into your workflow to execute the processing of dot files after their generation.

```
<!-- Mac/Linux -->
<component class="oaw.util.stdlib.SystemCommand">
  <directory value="src-gen"/>
  <command value="sh"/>
  <arg value="processdot.sh"/>
</component>

<!-- Windows -->
<component class="oaw.util.stdlib.SystemCommand">
  <directory value="src-gen"/>
  <command value="cmd"/>
  <arg value="/c"/>
  <arg value="processdot.sh"/>
</component>
```

Example Transformation

The following is an example transformation, extensively commented. It visualizes meta models (i.e. any *.ecore* file).

```
// this is the source meta model - ecore for the example here
import ecore;
// this is the target meta model; dot is the language to
// create graphviz graphs
import dot;

// there's a number of utility functions for working with the
// dot meta model
extension dotlib;

// top level, we create graphvizmodel which contains
// a number of graphs, each eventually resulting in its
// own dot file, and GIF image
create dot::graphvizmodel toGraphVizmodel(EPackage p):
  // you can add a number of graphs. Here, we add
  // only one.
  addGraph( toGraph(p) )
  ;
// mapping a model element to a graph uses the mapToGraph
// function from dotlib. All those mapping functions need
// to be cached to make sure that if they are called several
// times they are evaluated only once.
cached toGraph(EPackage p):
```

```
    // creates the actual graph object
  p.mapToGraph()
    // each graph needs to have a name;
    // will result in the filename for the dot
    // and gif files for the graph
    .setName( "ecoremetamodel" )
    // a graph contains statements. Statements
    // can be nodes, edges or subgraphs. Here we
    // add a subgraph for the package.
    .addStatement( p.toSubgraph() )
  ;
cached toSubgraph( EPackage p ):
  // this call maps the package to a subgraph....
  p.mapToSubgraph()
    // every subgraph MUST have a name - very important!
    .setName("ecore")
    // this is the label shown in the diagram
    .setLabel("ecore")
    // again, we cann add statements to graphs:
    // nodes, edges and additional subgraphs.
    // here we add a node for each class in the EPackage
    .addStatements( p.classes().toNode() )
    // and then we create an edge for each reference
    // of an EClass to another EClass.
    .addStatements(
p.classes().eReferences.select(r|EClass.isInstance(r.eType)).toRefEdge() )
    // finally, we create an edge for the inheritance
    // relationships. Note the difference in the coding
    // between the implementation for the referneces
    // and for the inheritance. The difference is that
    // for the reference there's an object (an instance
    // of EReference) in the source model. For the in-
    // heritance thing, there isn't.
    .addStatements( p.classes().addSuperclassEdges() )
;

// create a Node for an EClass
cached toNode( EClass c ):
  // this one creates the actual node object
  c.mapToNode()
    // the record shape is special, it can have
    // several "compartments", like the UML class
    // symbol. That's what we need here.
    .setShape("record")
    // sets the label, using the special { and } character
    // do define compartments in the record shape.
    .setLabel( "{"+c.name+"|"+c.eAttributes.collect(a|a.name+":
"+a.eType.name).toString("\\n")+"}" )
    // sets the style to bold lines and filled
    .setStyle("bold, filled")
    // fill color, obviously
    .setFillColor("grey")
    // and line color.
    .setLineColor( "black" )
;

// creates an edge for an EReference
cached toRefEdge( EReference r ):
  // creates the actual edge. Parameters
  // are source node, target node, and the
  // edge node itself
  mapToDirectedEdge( r.eContainer, r.eType, r )
    // the label on the edge
    .setLabel( r.name )
    // the label at the target end of the
    // line
    .setHeadLabel( "        "+r.lowerBound.toString() + ".." +
r.upperBound.toString() )
    // set the arrow head type
    .setArrowHead("vee")
    // and use a diamond shape at the tail of
    // the arrow if it's a containment relaitonship
    .setArrowTail( r.containment ? "diamond" : "none" )
;
```

```
// this one is used to iterate over all the superclasses
// for a given class. In a very real sense, the only
// reson for this function is to declare a variable
// name for base!
addSuperclassEdges( EClass base ):
  // call toSuperclassEdge for all supertypes of
  // a given base clas
  base.eSuperTypes.toSuperclassEdge( base )
;

// creates a direct edge for an inheritance relationship
cached toSuperclassEdge( EClass super, EClass base ):
  // creates the actual edge. Parameters are
  // source node, target node and the node for
  // the edge itself (here: null)
  mapToDirectedEdge( base, super, null )
    // sets the arrowhead to be the
    // "UML inheritance arrow"
    .setArrowHead("empty")
;

// helper function returning all the EClasses
// in an EPackage
classes(EPackage p):
  p.eClassifiers.typeSelect(EClass);
```

You can now run the generated `processdot.bat` or `processdot.sh` to render the actual images.

References

This reference cannot cover a complete description of Graphviz and the dot language. So it is recommended to have the respective reference documents by hand to be able to use the full power of this library. See these links for further reading:

- www.graphviz.org, The Graphviz homepage.

- www.graphviz.org/doc/info/lang.html, The DOT language. A definition of the DOT abstract grammar.

- http://www.graphviz.org/doc/info/attrs.html, Node, Edge and Graph Attributes reference.

- Drawing graphs with dot, A user guide for DOT. Recommended reading.

- http://sourceforge.net/projects/eclipsegraphviz, An Eclipse plugin which provides a viewer for DOT graphs.