Eclipse-based graphical rendering and editing of Modelica code

LUNDS
UNIVERSITET
Lunds Tekniska Högskola

LTH School of Engineering at Campus Helsingborg
Department of Computer Science

Bachelor thesis:
Kristina Olsson
Lennart Moraeus

LTH School of Engineering
Lund University
Box 882
SE-251 08 Helsingborg
Sweden

LTH Ingenjörshögskolan vid Campus Helsingborg
Lunds universitet
Box 882
251 08 Helsingborg

Printed in Sweden
Media-Tryck
Biblioteksdirektionen
Lunds universitet
Lund 2011

Abstract

This report describes the development of software that generates graphical icons for classes written in Modelica, an object oriented language for modeling complex systems. The icon rendering is developed as a new feature in the JModelica IDE, a plugin for the multi-language software development environment Eclipse. The JModelica IDE is included in the distribution of the open source project JModelica.org, the aim of which is to create an industrially viable open source platform for optimization of Modelica models.

Icon rendering of Modelica models is a step towards integrating graphical editing in JModelica IDE, which will facilitate the editing and decrease the editing time.

Details of the implementation, as well as the difficulties encountered along the way, are discussed in this paper. A short introduction to the different projects and technologies relevant to the development is also given.

The project described in the paper was done at Modelon AB in Lund.

Keywords: JModelica IDE, Modelica, Grapical Development Environment, Java AWT, icon rendering, open-source

Sammanfattning

Denna rapport beskriver utvecklingen av mjukvara för rendering av grafiska ikoner utifrån klasser skrivna i Modelica, ett objektorienterat språk för modellering av komplexa system. Ikonrenderingen utvecklas som en funktionalitet i JModelica IDE, ett insticksprogram för utvecklingsmiljön Eclipse. JModelica IDE ingår i distributionen av open-source-projectet JModelica.org som har syftet att skapa en industriellt gångbar plattform byggd på öppen källkod för optimering av Modelica-modeller.

Ikonrendering utifrån Modelica-modeller är ett steg på vägen mot att integrera grafisk editering I Jmodelica IDE, vilket kommer att underlätta redigeringen och minska redigeringstiden.

Rapporten beskriver implementationen i detalj, samt diskuterar de svårigheter som dök upp under utvecklingsarbetet. En kort introduktion till de olika projekt och teknologier som är relevanta till arbetet ges också.

Det projekt som rapporten beskriver utfördes på Modelon AB i Lund.

Nyckelord: JModelica IDE, Modelica, grafisk utvecklingsmiljö, Java AWT, ikonrendering, open-source

Foreword

We would like to thank everyone at Modelon AB for providing a pleasant and stimulating environment and for always taking time to give advice and answer questions concering our project. Specifically, we would like to thank Jesper Mattsson and Johan Åkesson for their invaluable advice and support. We would also like to thank our supervisor at LTH, Mats Lilja.

List of Contents

1 Introduction

The work of this thesis project is a part of the open-source project JModelica.org [1]. JModelica.org is a result of research at the Department of Automatic Control, Lund University and is now maintained by Modelon AB where this bachelor thesis project takes place. The main objective of JModelica.org is to create an industrially viable open-source platform for optimization of Modelica models [1]. Modelica is a modeling language for large complex systems [2]. Modelica tools are used for similar purposes as Simulink for MATLAB, LMS Imagine.Lab AMESim and other multi-domain simulation environments. The work of JModelica.org is mainly concentrated around simulation and optimization, but there are also other interests such as the ability to offer a user-friendly integrated environment.

Modelon AB provides engineering services as well as full solutions in Modelica-based engineering and system design to customers all over the world. Industries that consult the expertise of Modelon AB include the automotive industry, the energy and process industry and the aerospace and defense industry. Modelon AB also develops Modelica libraries, distributes third-party Modelica development software and libraries, and is active in the development of the Modelica language. [3]

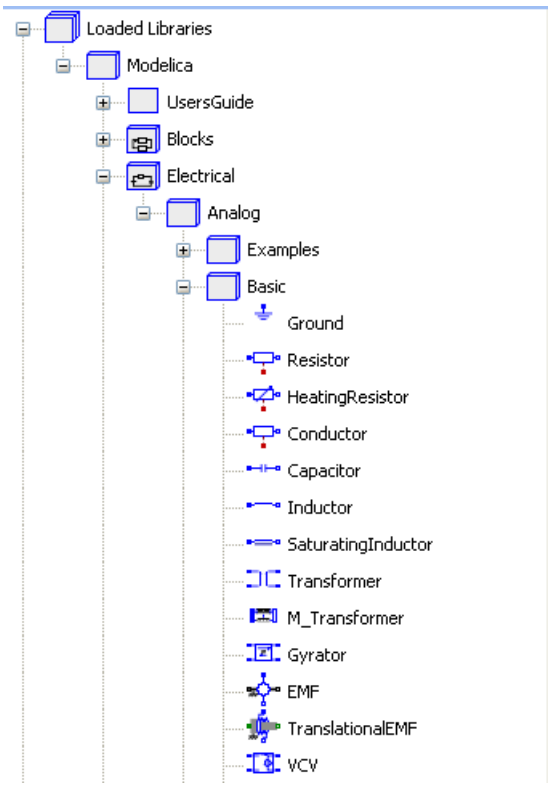


Figure 1: Icons in the JModelica IDE outline.

2 Goal

In a previous master thesis at Modelon AB, a basic integrated development environment for Modelica code named JModelica IDE was developed [4]. The purpose of the project is to begin the process of extending the JModelica IDE to also include a graphical development environment. This includes the following features

- Rendering of icons in the JModelica IDE outline
- Synchronization between the graphical layer and the textual layer
- Automatic generation of Modelica source code based on operations in the graphical layer
    - Creation of components (drag-and-drop from a class browser)
    - Connection of components
    - Editing of parameters of components

The introduction of a fully integrated graphical user interface - as described above - would be a significant improvement to the functionality of the JModelica IDE.

Since JModelica IDE is written in the Java language, the icon rendering must also be implemented using Java.

During the development process, the goals of the thesis project were continually reviewed so that we maintained a realistic idea of how much we would have time to implement. Early on in the project we realized that the implementation of rendering of icons in the JModelica IDE outline was complex enough to make up the entire project. This came as no surprise, as our supervisor had advised us from the beginning that this might turn out to be the case.

The question that this thesis explores is: is it possible to render icons in the outline of JModelica IDE from annotations in Modelica models?

3 Background

3.1 Modelica

Modelica is an open standard for equation and component based object-oriented modeling of complex physical systems. The fundamental structuring unit in Modelica is the class. Classes are divided in *Specialized Classes* depending on their purpose and usage. Model, connector and function is different type of classes with different restrictions. A model is a class without restrictions. A model is composed by components, which are instantiated classes and may inherit one or several classes. The fact that the model is divided into smaller real-world components makes the models easy to create and understand.

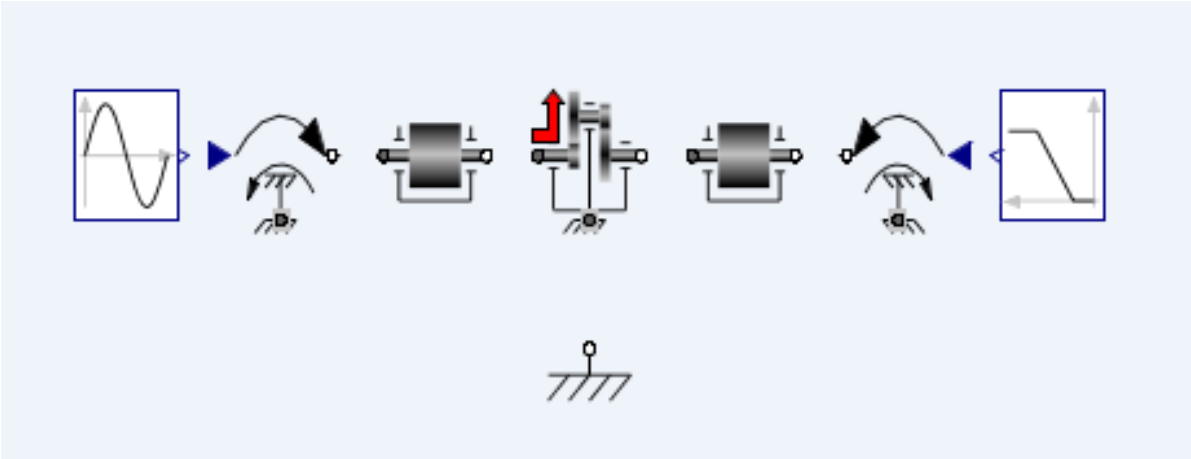


Figure 2: A Modelica model that consists of many small components. The icons are drawn by the JModelica IDE icon rendering software.

Development and maintenance of the open-standard Modelica Language Specification (MLS) and the free, open-source Modelica Standard Library (MSL) is manged by the non-profit organisation Modelica Association. There are several simulation environments available for Modelica, both free ones and commercial ones. [2]

3.1.1 Annotations

In Modelica, annotations are special parts of the source code that can be used to provide additional information. The annotations are standardized and defined in the language specification. Among the many possible uses for annotations is the ability to provide information about the graphical representation of a model. This is the information that is relevant to this project. Other uses for annotations include documentation and versioning.

Besides the standardized annotations Modelica tools are free to define new annotations and use them in any way that the tool developer finds useful. [2]

3.1.2 Graphical representation of models

This chapter gives a brief overview of the specification for graphical annotations in Modelica. For more detailed information, please refer to Appendix A, or chapter 17.5 of the MLS [5]. The MLS uses the the specialized Modelica class *record* to specify how the graphical information in the annotations is supposed to be written.

A graphical representation of a Modelica class consists of two abstraction layers, the *icon layer* and the *diagram layer*. A layer is represented in the annotation of a class by an *Icon* or *Diagram* record. Generally speaking, the *icon layer* is used for visualizing a specific model and does not show all of its details, while the *diagram layer* is used for displaying the model's components and the connections between them. Each *layer* has its own graphical representation for the class, consisting of a list of graphic primitives. These graphical primitives are represented by records named *Rectangle*, *Ellipse*, *Line*, *Polygon*, *Text* and *Bitmap*. All of the primitive records extend the record *GraphicItem* and all except for *Line* and *Bitmap* extend the record *FilledShape*. The records for the primitives contain various information about how they should be drawn, such as their color, shape and coordinates. The layers also contain a *CoordinateSystem* record which describes the context in which the coordinates of the primitives should be interpreted.

```
connector C
    annotation (
      Icon(
        coordinateSystem(
            preserveAspectRatio=true,
            extent={{-100,-100},{100,100}},
            grid={2,2}),
        graphics={
          Rectangle(
            extent={{-100,-100},{100,100}},
            lineColor={0,0,255})
        }
    ));
end C;
```

Figure 3: An example of a graphical annotation containing a layer (Icon) and a primitive (Rectangle). To the right, the rendered icon resulting from the annotation is shown (”C” is not part of the icon).

If a Modelica class contains a component (i.e. an instance of another class), then the component declaration can contain its own annotation. This annotation is called a *placement* annotation and describes how the graphical primitives of the component should be drawn in relation to the primitives of

the enclosing class. The *placement* provides this information for both the primitives in the icon layer and the ones in the diagram layer of the component. The layer that is used in any given case is determined by two factors: which layer is used for the enclosing class and the restriction of the class of the component.

When drawing the *icon* representation of a class, the *icon layer* of the component is used - provided that the component is a *connector* (a specialized Modelica class) and is not declared as *protected*. If the component does not meet these requirements, it is not shown at all when the icon of the enclosing class is drawn, regardless of the placement annotation. When drawing the *diagram* representation of a class, the *diagram layer* of the component is used for *connectors* and the icon layer is used for components of other class types. When drawing the *diagram* representation of a class the visibility of the components are irrelevant. Additionally, connections between components are shown as lines in the diagram layer of the enclosing class.

	component	draws component's ... layer
drawing icon layer	connector	icon
	other	none
drawing diagram layer	connector	diagram
	other	icon

Figure 4: A table displaying the logic of which layer of a component to use, depending on the component's class type.

Figure 2 in chapter 3.1 shows an example of this. It shows the diagram layer of a class and contains the icon layer representations of the components of the classs.

3.2 JastAdd

JastAdd is a java-based open source construction framework for building extensible compilers. The JastAdd source code consists of definition files and aspect files. Out of these files Java source files are generated which represent the source code of the program being compiled in an Abstract Syntax Tree (AST). The definition files contain the names of the classes that are to represent the AST nodes and the aspect files contain methods and attributes to be added to the given classes. The functionality of a compiler build with JastAdd is expanded by adding new aspects which is easy comparable to rewrite the Java source code. [6]

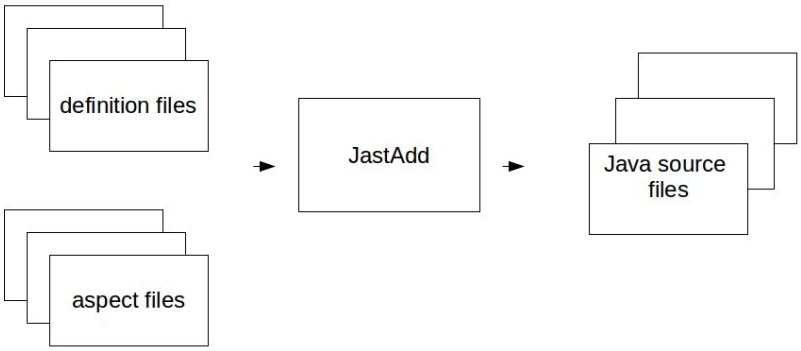


Figure 5: A diagram illustrating how JastAdd builds the Java classes that represent the AST nodes.

3.3 JModelica.org compiler

In JModelica.org a Modelica compiler is included. The compiler was created as a part of the doctoral thesis *Languages and tools for Optimization of Large-Scaled Systems* and is developed using the JastAdd compiler construction framework. [7]

3.3.1 The compilation process of a Modelia model

When a Modelica model is compiled, the JModelica.org compiler creates abstract syntax trees that represent the model. The nodes in the AST:s represent the different Modelica language constructs that make up the model: classes, components and equations to only name a few. The JModelica.org compiler first creates a source AST from the source code being compiled. It

then creates an instance AST which has its root in one of the class declarations of the source AST. The differences between these two ASTs that are relevant to this thesis are as follows. First, the source AST is where the annotations are stored. However, since every node in the instance AST contains a link back to its corresponding node in the source AST, the annotations of instance nodes can be accessed anyway. This does mean that annotations that reference variables in a class can not be correctly parsed, though, since the values of the variables are not known unless the class is instantiated. Secondly, the *redeclare* construct in the Modelica language causes some problems. In Modelica, developers can allow for the types of parameters and components in the class to be changed by declaring parameters and components *replaceable*. The types are changed by using the *redeclare* construct in the declaration. This means that the class of a component is not always known until the class that contains the component is instantiated. Since annotations are evaluated in the source AST, the annotations of *redeclared* parameters or components will not be possible to evaluate. Because instantiating classes is quite computationally expensive, the conditions for this project has been to render icons out of the information stored in the annotations without knowledge of the information in the instance AST.

3.3.2 The AST representation of a Modelica model

In the source AST, the most relevant nodes that represent a Modelica model are called *ClassDecl, ExtendsClause* and *ComponentDecl.* The inheritance of a class is represented by the *ExtendsClause* node and a component is represented by the *ComponentDecl* node. The *ClassDecl* contains the node *Annotation* that stores the annotation of the Modelica class. The source AST does not contain a link between an *ExtendsClause* and the *ClassDecl* that represents its class. The same is true for a *ComponentDecl* and the *ClassDecl* that represents the class of the component. However, during this project the JModelica.org team added attributes to the aspect *SimpleLookup* which made it possible for us to reach the *ExtendsClause'*s and the *ComponentDecl*'s *ClassDecl* nodes. The corresponding nodes in the instance AST are called *InstClassDecl*, *InstExtends* and *InstComponentDecl.* In contrast to the source AST the instance AST does contain links between these nodes.

In the Modelica language there is a short way of declaring a model that extends another model, and only makes small adjustments to the model that it extends. This *short class* declaration raises a *ShortClassDecl* in the source AST.

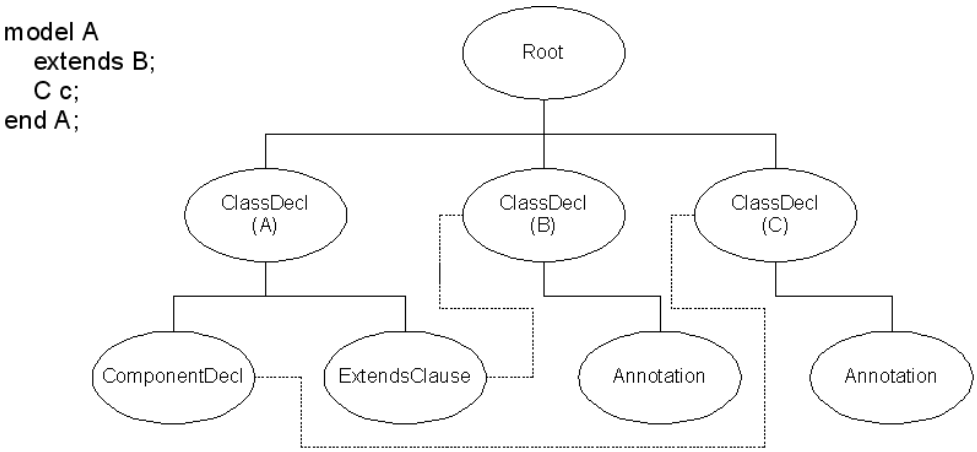


Figure 6: A simplified model of the *source AST*. The dotted lines represent *ClassDecl* nodes that are returned by calls to attributes on the *ExtendsClause* and *ComponentDecl* nodes defined in the the aspect *SimpleLookup*.

In the aspect AnnotationAPI the class *AnnotationNode* forms a tree out of the Annotation node which makes it possible to iterate over the annotation and extract the information stored in the annotation.

3.4 JModelica IDE

Included in the JModelica.org distribution is the integrated development environment JModelica IDE. JModelica IDE uses the JModelica.org compiler and is implemented as an Eclipse plugin [4]. The JModelica IDE supports syntax highlighting, brace matching, code outline, error markup and code folding. In the present situation JModelica is a textual editing enviroment, but the purpose is to also include graphical editing. JModelica IDE has several outlines, *Source Outline, Instance Outline* and *Class Outline*. *Source Outline* and *Instance Outline* are source AST and instance AST represention of the file in the current editor. *Class Outline* is a source AST representation of all Modelica files in the current project plus the Modelica files in imported libraries. The icons used in the outlines are pre-rendered images. There is a default image for a package, a model and a component.

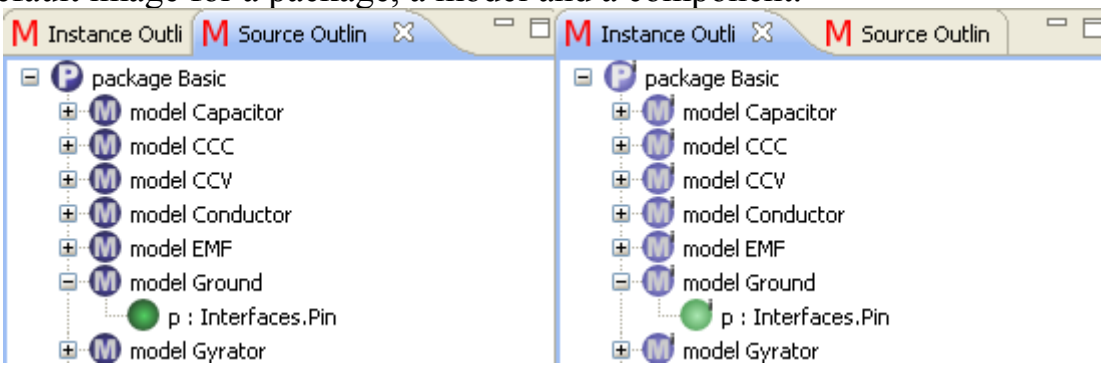


Figure 7: Outline views in JModelica IDE using default icons.

When a user uses the JModelica IDE to edit a Modelica file, the file being edited is continuously parsed by Eclipse using the JModelica.org compiler. Each AST node generated from the classes in the Modelica file is sent to the outline view in the JModelica IDE along with a default icon image.

4 Implementation

We have implemented a module in the JModelica IDE that renders icons from the annotations stored in the Modelica source code. The module consists of a java package named *org.jmodelica.icons* to represent a Modelica model as a java data structure (an *Icon)*, a drawing package named *org.jmodelica.icons.drawing* to render an *Icon* to an image and a package with exceptions named *org.jmodelica.icons.exceptions*. The module also consists of two aspects for adding and creating *Icons* and adding rendered *Icons* for the AST nodes. In the outlines of JModelica IDE, the default icons are replaced by these rendered icons.

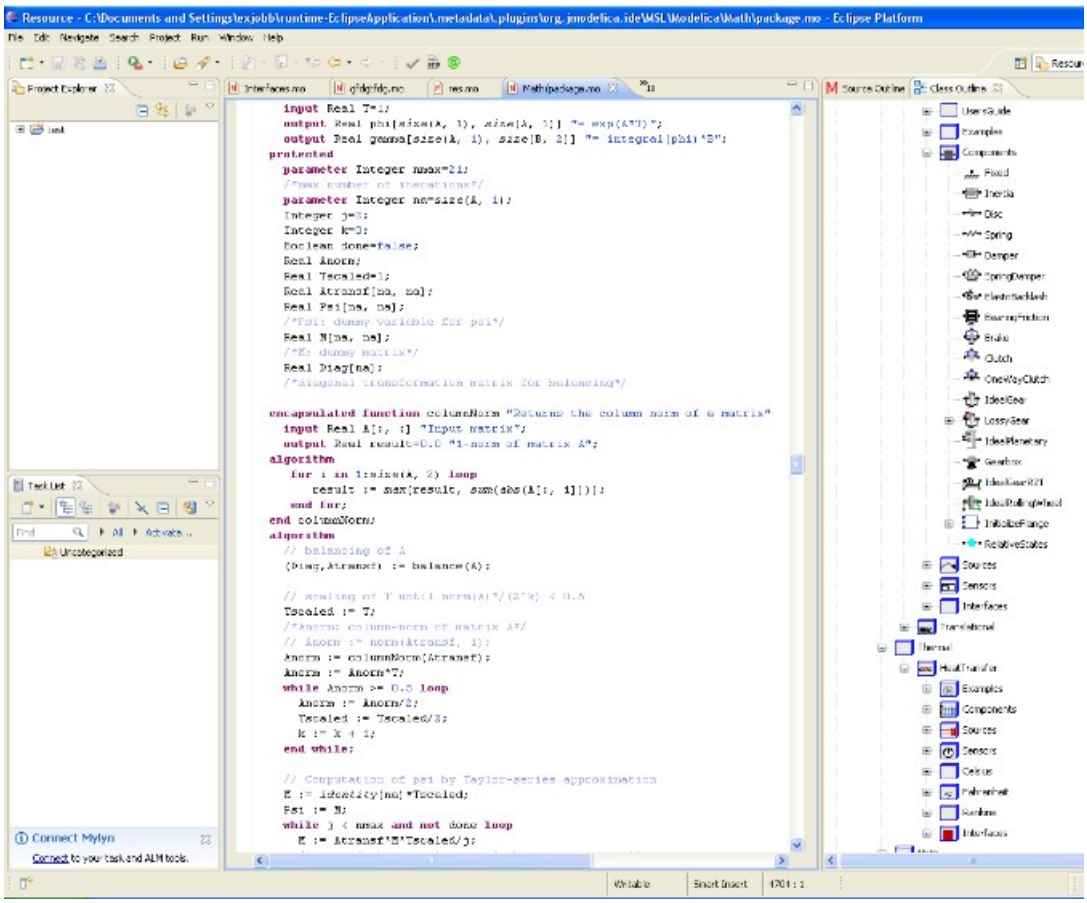


Figure 8: A screenshot of the JModelica IDE using the icon rendering implemented in this project.

The following sections describe how we have implemented the icon rendering. 4.1 describes the icon data structure, 4.2 describes the icon drawing, 4.3 describes the exceptions used in the implementation and 4.4 describes the aspects.

The icon rendering software is included in the distribution of

JModelica.org. The implementation described here is revision 2664 [8], later revisions may include changes.

4.1 The icons java package

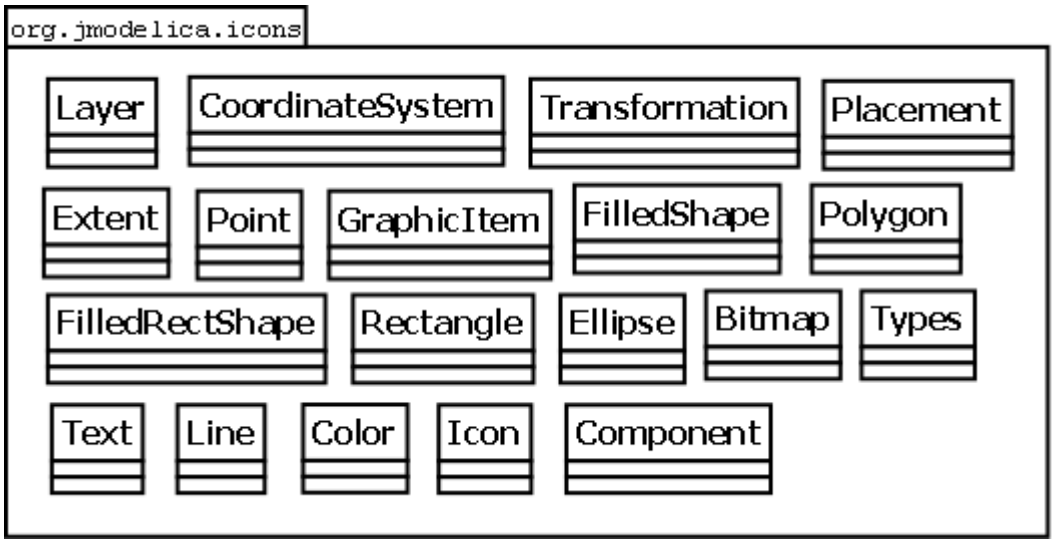


Figure 9: Classes in the package org.jmodelica.icons.

The package *org.jmodelica.icons* contains classes that form the graphical representation of a Modelica class and are implemented according to the *Modelica Language Specification* (MLS) [5]. Appendix A contains a summary of the chapter in MLS that describes this structure.

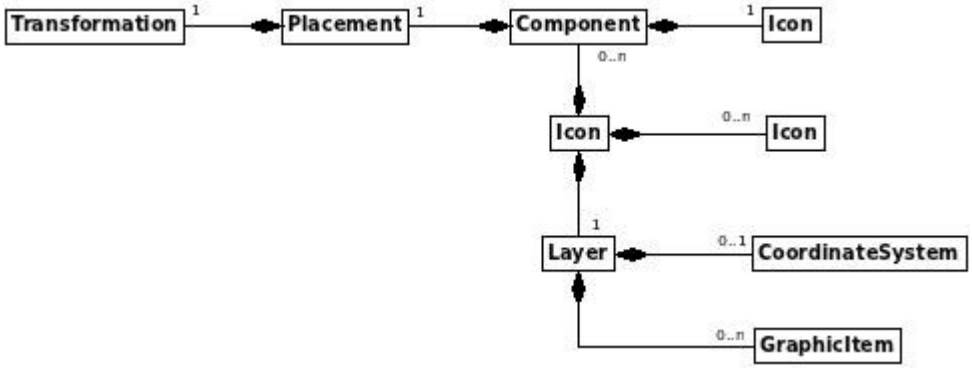


Figure 10: An object diagram of the icon class structure.

The class *Icon* is the graphical representation of a single Modelica class. The class *Icon* also stores references to the inherited classes and the components of the specific class. The inherited classes of the models are stored as a list of *Icons* and the components are stored as a list of *Components*. The graphic representation consists of a *Layer* which contain a *CoordinateSystem* and a list of *GraphicItems*. A *Component* consists of an *Icon* and a *Placement*. *Placement* contains a *Transformation*. *Transformation* stores the data concerning how to place the component in relation to the

enclosing class's coordinate system. Since only one layer is used at a time the graphical representation of a Modelica model is equivalent with the *Layer*. One approach was to represent a model as a *Layer* with two lists of *Layers* for inheritance and components. That would both reduce the number of classes and let the program structure stay closer the MLS. On the other hand, the classes *Icon* and *Component* make the class structure more clear. We considered a clear class structure to be more important than to stay close to the MLS.

The *Extent* class describes an area defined by two *Points*. We have added methods for calculating the width and height of the extent, as well as for determining the *Point* in the middle of it. There are no rules for how the *Points* of an *Extent* are written in Modelica annotations (other than that they have to cover a rectangular area). This means that the an Extent that starts at the point (-10, -10) and ends at (10, 10) can be expressed as "extent = {{-10, -10}, {10, 10}}", but might as well be expressed as "extent = {{-10, 10}, {10, -10}}". In some cases this difference is insignificant, such as when declaring the *Extent* of a *Rectangle* primitive. In other cases, however, it is highly significant. For example, if the second example was given as the *Extent* of a *Transformation* inside a *Placement* record, it would mean that the icon of that particular component should be drawn with all of its primitives flipped along the X axis. This has prompted us to implement a method for returning the "fixed" version of an extent, that is a version of the extent where it is guaranteed that the coordinates of the first point are less than or equal to those of the second point. Critically, many methods in the graphics library we are using requires that rectangular areas are expressed in this way.

Another feature that we have had to implement in the *Extent* class is the method *contain*. It takes a second *Extent* as its parameter and returns a third *Extent* that is large enough to contain both the first and the second *Extent*. This method is critical in calculating the total space that an icon covers, including all of the icon's graphical primitives as well as the graphical primitives of the icon's super classes and components. The total area of the icon is used to calculate how much all coordinates should be scaled so that the primitives fit inside the icon image.

4.1.1 Graphic items

GraphicItem is the basic class that all graphical primitives inherit. Due to the need to calculate the total size of an icon, we have deemed it necessary to be able to calculate the rectangular bounds of any graphical primitive. Since the method of calculating the rectangular bounds of a primitive varies between different primitives, we have added an abstract method *getBounds* in GraphicItem that must be overidden by the subclasses so that every primitive calculates its bounds in its own way. We introduced the class *FilledRectShape*

to reduce redundancy for primitives which are handled like rectangles (rectangles, ellipses and texts). *FilledShape* inherits *GraphicItem*, which is not the case in MLS. The reason for this is that Modelica allows for multiple inheritance while Java does not.

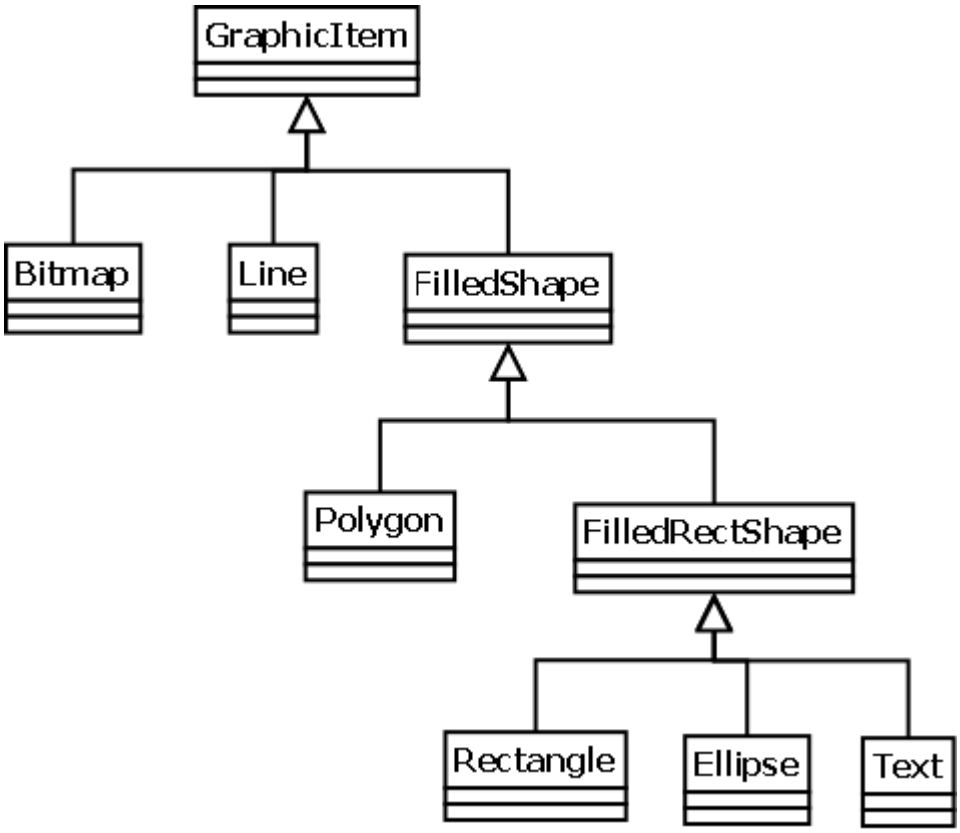


Figure 10: A class diagram displaying the class structure of the classes representing the graphical primitives.

The *Bitmap* primitive represents a bitmap image placed somewhere in the icon. It is one of few icon-related features described in the MLS that are not yet fully implemented in the JModelica IDE. According to the MLS, Modelica developers should be able to specify source data for bitmap images in several different ways. First, it should be possible to specify file paths using URI schemes. The regular file URI scheme should be supported, as well as a special "Modelica URI scheme" described in the MLS. Additionally, developers should be allowed to specify their bitmap image as a text string that is encoded using the base 64 encoding scheme [9]. All of these features are fully supported by the JModelica IDE except for the "Modelica URI scheme" method for specifying file paths.

Line primitives in the Modelica graphics annotation system are specified by a list of two or more *Points*. Beyond specifying various preferences such as line pattern, line thickness and color, the MLS says that developers should be

able to indicate whether or not a *Line* should have *Arrows* at one or both of its ends. We have chosen to implement this feature by storing the arrows on the *Line* object as an array containing two line-shaped *Polygon* objects. If one of the arrow values is "Arrow.None", then that arrow polygon is assigned the value *null*.

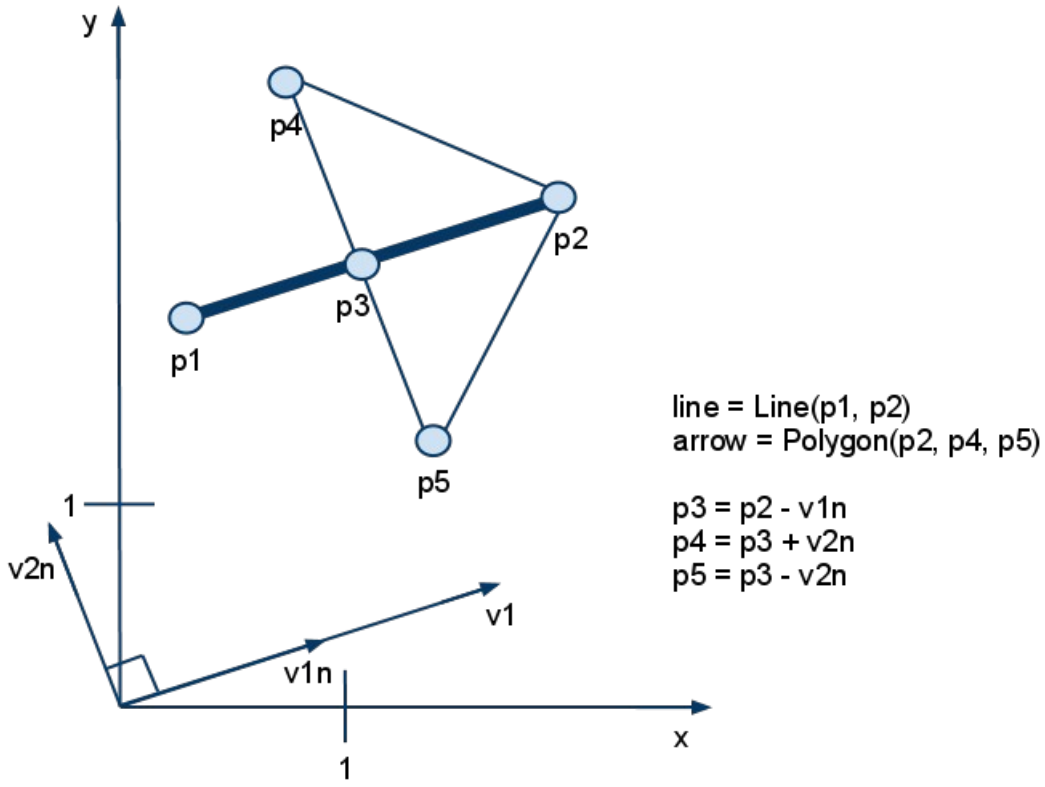


Figure 11: The vector mathematics used to calculate the coordinates of the arrows. For simplicity, it is assumed that *arrowSize* is equal to 1. Since a *Line* primitive is allowed to have any number of line segments, the process described above is first executed with the first point of the *Line* as p2 and the second as p1, and finally with the second to last point as p1 and the last point as p2.

In the *Line* class, we have also added a method for calculating the rectangular bounds of the line. These bounds are given as the smallest rectangular area that contains all of the *Line*'s points.

4.1.2 Color

Our color representation consist of three integers representing the red, green and blue components of a color. We have added methods for returning a darker or lighter version of the same color. This is useful when drawing the border pattern of a rectangle, making the edges brighter or darker to make the rectangle appear sunken or raised.

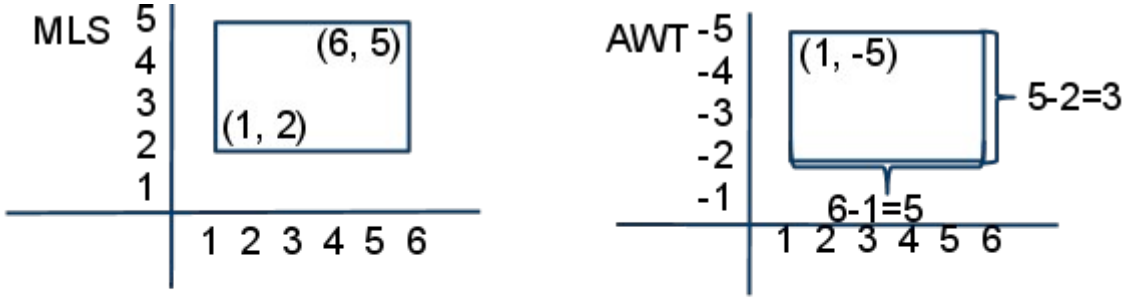
4.2 The drawing

The icon drawing is handled in *AWTIconDrawer*. An instance of this class is created for each icon to draw, with the current icon as a parameter to the constructor. *AWTIconDrawer* contains methods for drawing all of the different graphical primitives that icons can contain. The graphical primitives are drawn on an image, using the AWT (Abstract Window Toolkit) graphics library.

The drawing methods of the *AWTIconDrawer* class are called by the *draw* method in the *Icon* class, so that all of the *Icon*'s graphical primitives are drawn correctly. This method also calls the *draw* methods of the *Icon*'s super classes and components, making sure that these are drawn in the correct order.

4.2.1 Drawing the graphical primitives

When drawing MLS primitives which extend *FilledShape*, we take advantage of AWT's *Shape* construct which allows shapes to be filled and drawn in a powerful but simple manner. For this purpose there are *createShape* methods in *AWTIconDrawer* that construct the AWT objects that represent the MLS primitives. These are called every time a MLS primitive is drawn. Since AWT's coordinate system extends downwards and to the right - as opposed to those in MLS which extend upwards and to the right - it is necessary to make adjustments to all coordinates before they are passed to the AWT drawing methods, or all icons will be drawn upside-down. The *createShape* methods is where we choose to do this.




```
return new java.awt.geom.Rectangle2D.Double(
    (extent.getP1().getX()),
    -(extent.getP2().getY()),
    (extent.getP2().getX()-extent.getP1().getX()),
    (extent.getP2().getY()-extent.getP1().getY())
);
```

Figure 12: The Shape object of a Rectangle is created. Note the Y-coordinate passed to the Rectangle2D constructor.

Figure 12 shows the conversion between a MSL *Extent* (two points) and an AWT *Rectangle2D* (a point, a width and a height). This expression creates the *Shapes* of *Rectangle* and *Text* primitives from the MLS. Similar adjustments are made to the MLS *Ellipse* and *Polygon* primitives.

For *Line* primitives, the y coordinates are inverted in the *drawLine* method right before they are drawn.

4.2.2 Transformations

For the transformations we have used the AWT class *AffineTransform*. An instance of *AffineTransform* contains a transformation matrix and a *transform* method that transforms a *Point2D* object with the matrix. To change the matrix, there are methods in the class named *translate*, *scale* and *rotate* which take numbers as arguments and execute the correct mathematic operations on the matrix. Using *AffineTransform* makes it easier to keep track of the matrix data structure. It also means that it is not necessary to manually manipulate the matrix when translations, scaling operations and rotations need to be applied to it. The developers only need to call the corresponding methods on the *AffineTransform* object.

The first transformation that is applied when drawing the icon of a model has to do with fitting the contents of the icon inside the icon image. Creating this transformation involves calculating scale factors and translation increments. The reason that this transformation is needed has to do with the fact that there are no restrictions on the coordinate system of a Layer record in a Modelica graphical annotation. On top of that, the coordinates of primitives can extend outside the coordinate system, and should still be visible in the icon image. The drawing coordinates passed to the methods in the graphics library must be in the range of the image's first pixel (0) to the image's last pixel (currently 20). The goal of this transformation is to make all coordinates fit into that range (see figure 13).

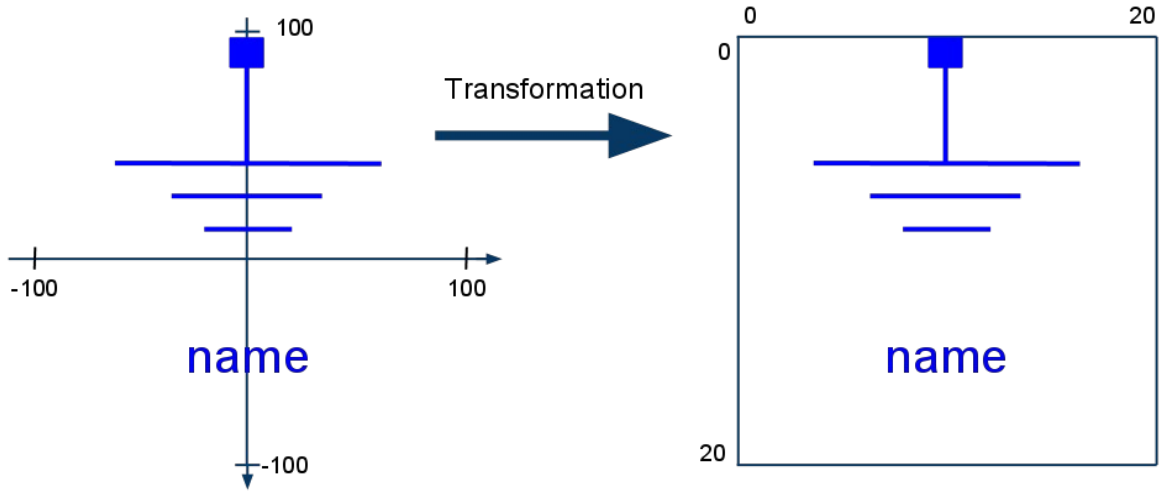


Figure 13: The initial transformation applied when drawing the icon of a model. Note that the graphical annotations in Modelica contain coordinate systems which extend upwards and to the right, while the coordinates of the image extend down and to the right. However, this difference is accounted for at a later stage, when creating the graphic objects passed to the AWT graphics library, so there is no need to handle this in the transformation (see section 5.3).

First of all, the total width and height of the icon must be calculated. This width and height is aquired by calling the *getBounds* method of the icon, with the icon's coordinate system as its parameter. The *getBounds* method in *Icon* is a method that for every graphical primitive calls the method *getBounds* declared in *GraphicItem*. The returned extent is guaranteed to be large enough to contain the coordinates of the primitives of the main class as well as those of its super classes and components, and is larger than or equal to the coordinate system extent that was passed as an argument.

This width and height must then be compared to the width and height of the image that the icon will be drawn on. The result of this comparison is the two scale factors, which every coordinate from then on must be multiplied by. There is one scale factor for the x-coordinates and one for the y-coordinates, since both the icon size and the size of the final icon image could potentially be of any proportions. As a side note, we currently preserve the aspect ratio of all square icons by extending their size equally in all directions when it needs to be extended. The aspect ratio of non-square icons is not correctly preserved in our implementation, unfortunately.

Once the scaling operation has been done on the transformation matrix, the translation increments are calculated as half of the image's width and half of the image's height, respectively. Unfortunately, this method of calculating the translation increments doesn't take into account whether or not the origin is centered in the icon's coordinate system. This means that icons with origins that are not centered in their coordinate systems are not correctly translated.

When the transformation described above has been applied, the *draw* method of the icon is called. The *Icon* object recursively draws its own super classes and components, and thus makes sure that the correct transformations are applied when drawing these. When the component of a class is drawn, we must make sure that the icon of that component is drawn in the correct place, as described by the *Placement* annotation of that component declaration. Applying the transformation of a component means taking into account the *Placement* of the component, the *CoordinateSystem* of the class of the component and the enclosing class *CoordinateSystem* when calculating translation increments and scale factors.

```
model VariableResistor
  ...
  Modelica.Blocks.Interfaces.RealInput R
    annotation(
      Placement(
        transformation(
          origin={0,110},
          extent={{-20,-20},{20,20}},
          rotation=270
        )
      )
    );
  ...
end VariableResistor;
```

Figure 14: The component transformation. The icon for VariableResistor contains a scaled, translated and rotated version of the icon for RealInput, as specified in the component declaration's Placement annotation.

The *Transformation* record in the *Placement* annotation has the attributes *origin* (which is a *Point*), *extent* and *rotation*. To calculate the translation increments, the *origin* point is added to the point at the middle of the *extent*, which is added to the middle point of the coordinate system of the enclosing class. The x- and y-components of the resulting vector make up the translation increments of the transformation with which the component's icon should be drawn. The x-scale factor is determined by dividing the width of the *Placement Transformation* record's *extent* by the width of the coordinate system of the component's class, with the y-scale factor being calculated in the corresponding way. After this, all that is needed for the component's icon to be drawn correctly is rotating the coordinates by the amount written in the annotation, after converting the angle to radians and multiplying it by -1 (because of AWT's inverted y-axis).

4.2.3 Antialiasing

All drawing is done using antialiasing. Antialiasing affects the way that pixles are colored when only a fraction of the pixel is covered by a graphical primitive.

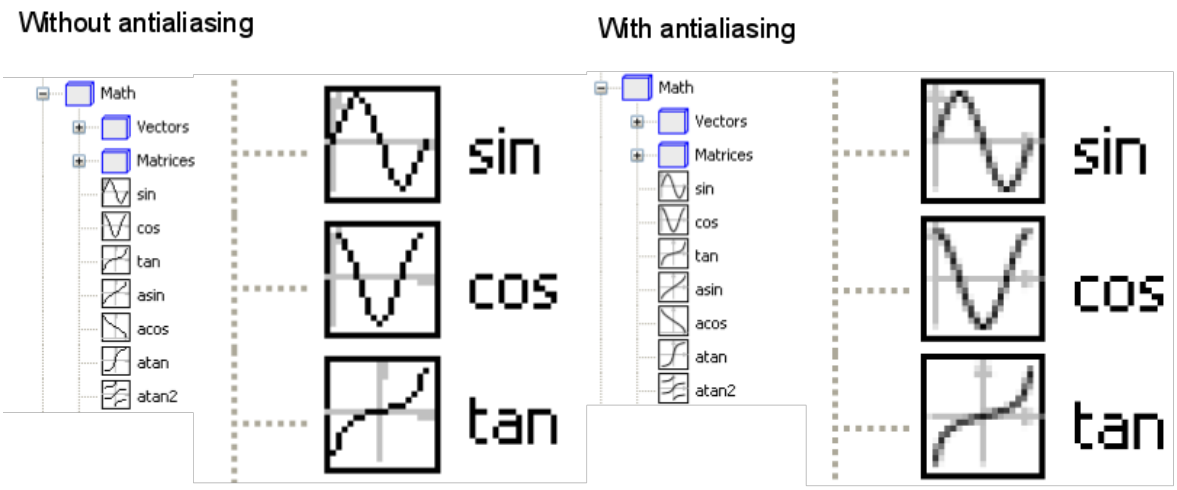


Figure 15: A comparison between the same icons drawn with and without antialiasing. Both examples are from the JModelica IDE, showing icons for models in the Modelica Standard Library.

Without antialiasing, pixles that have more than half of their surface covered by a primitive get the color of the primitive while all other pixles get the background color (or their original color before the current drawing operation). With antialiasing, a pixel that has a fraction of its surface covered by a primitive gets a color that is a mixture between the primitive's color and the pixel's original color. The more of the pixel's surface that is covered by the primitive, the closer the pixel's color will be to the color of the primitive. This results in a much smoother appearance, especially when drawing small primitives, such as the ones in the outline icons that are the focus of this project.

4.2.4 Border patterns

Border patterns are decorations at the edges of rectangles, specified by the attribute *borderPattern* in the annotation for *Rectangle*. The possible values for the attribute are *None*, *Sunken*, *Raised* and *Engraved*. There are no descriptions in the MLS of how these patterns are meant to be drawn, but we have implemented the drawing of border patterns for the values *Sunken* and *Raised*. *Engraved* has not yet been implemented.

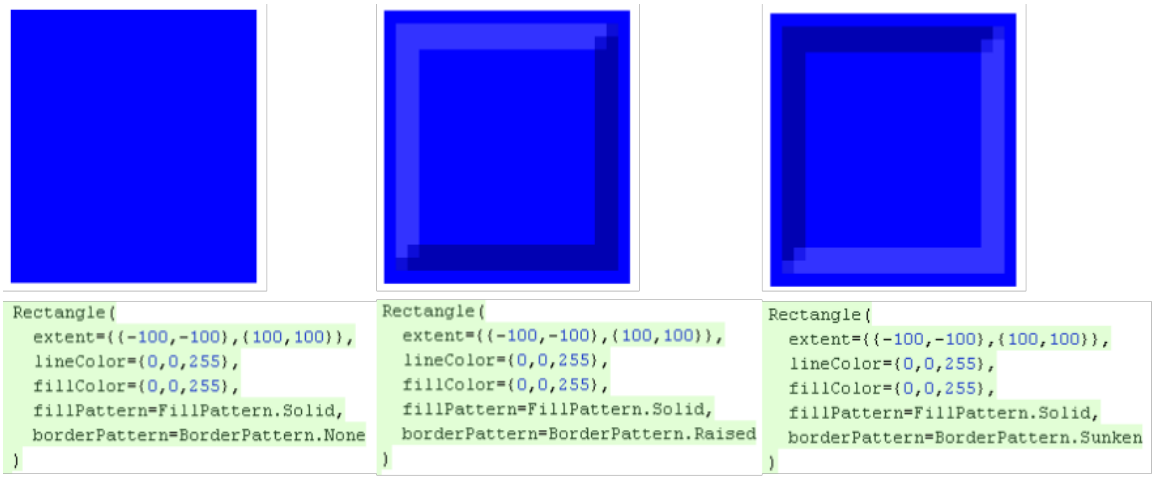


Figure 16: Rectangle primitives displaying the three currently implemented values for borderPattern, along with the annotations that declare them.

Our implementations of Sunken and Raised involves creating polygons at the edges of the rectangle and filling them with brighter and darker versions of the rectangle's fill color to give the appearance of the rectangle being raised or sunken.

4.2.5 Fill patterns

The *fillPattern* attribute of the *FilledShape* primitive describes how the primitive should be filled. As far as our implementation goes, there are three categories of fill patterns: regular fill patterns, gradients and texture fill patterns.

The regular fill patterns are *None* and *Solid*. The implementation of these is trivial: *Solid* means that the primitive should be completely filled with its fill color and *None* means that it should not be filled at all. Using the *Shape* construct of AWT, any shape can be filled with a single method call.

Gradient fill patterns include *Horizontal*, *Vertical*, *HorizontalCylinder*, *VerticalCylinder*, *Sphere*. Filling an area with a gradient means that the color should progressively shift between one color and another. In our

implementation - and as specified in the MLS - gradients go from the primitive's line color to its fill color. Again, thanks to the *Shape* construct of AWT, as well as comprehensive and well-documented classes for gradients in the AWT library, filling a shape with a gradient is not complicated. The class we have used to implement the gradients is *java.awt.GradientPaint*.

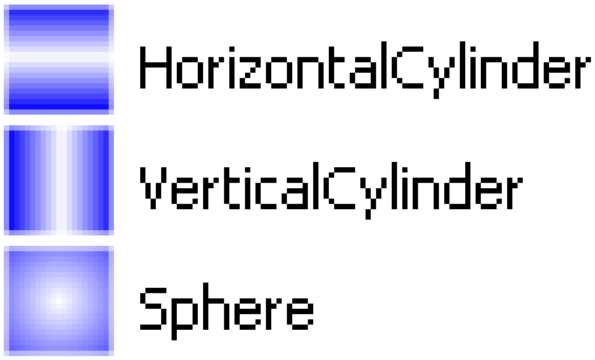


Figure 17: Shapes filled with the three gradient fill patterns.

The final category, texture fill patterns, consists of *Horizontal*, *Vertical*, *Cross*, *Forward*, *Backward* and *CrossDiag*. These patterns consist of stripes drawn in the primitive's line color over the primitive's fill color. We have implemented these patterns by creating a small image, drawing the lines on it, and then using the image as a texture to fill the primitive with. The class in AWT that best supports this is *java.awt.TexturePaint*. In AWT, the colors that the *Graphics* object draws with is determined by its *Paint* object. After setting the *Paint* object to an instance of *TexturePaint*, with the image loaded as its texture, the *Graphics* object will use the image as a texture to fill any shapes that are passed to its *fill* method.

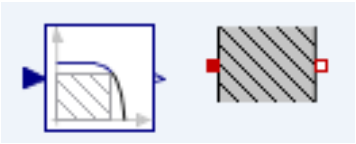


Figure 18: Icons containing shapes filled with the *Backward* texture fill pattern.

4.2.6 Line features

The style in which *Line* primitives and the outlines of other shapes are drawn is possible to specify by changing the *pattern* and *lineThickness* attributes of the *FilledShape* annotation, or the *pattern*, *thickness* and *smooth* attributes of the *Line* annotation. In AWT, the *Stroke* object of the current instance of the *Graphics* class determines in what style lines and outlines are drawn, and by creating a *java.awt.BasicStroke* with the correct attributes and setting it as the current *Stroke*, we make sure that the correct drawing style is engaged.

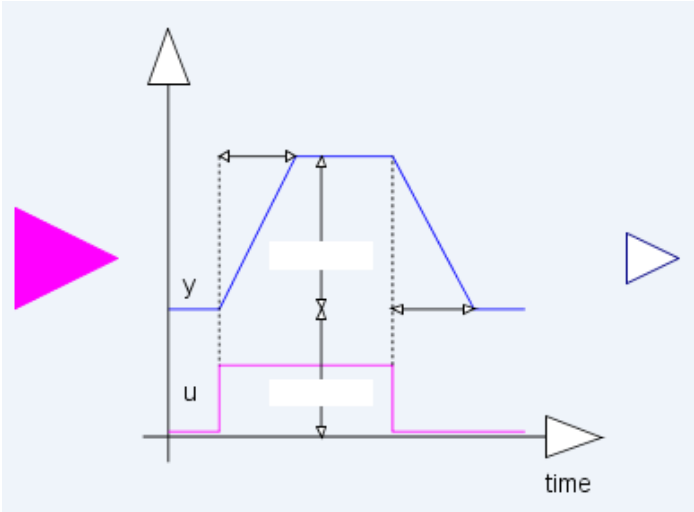


Figure 19: An icon containing lines with the *Dot* line pattern attribute.

The line thickness is simply passed as an argument for the *BasicStroke* constructor. For setting the line pattern, we use a constructor of the *BasicStroke* class that takes an array of *float* values called *dash* as a parameter. These *float* values determine how many separate dashes that the line pattern should consist of, and the length of these. The different line patterns that can be specified in graphical annotations are *None*, *Solid*, *Dash*, *Dot*, *DashDot* and *DashDashDot*. We have implemented these by creating a Java enumeration class called *LinePattern* with a float array as its attribute, to store the dash values for each enumeration type. For example, *LinePattern.DOT* has {0.2f} as its dash value, *LinePattern.DASHDOT* has {2.0f, 4.0f} and *LinePattern.SOLID* has *null*. This way, it is easy to construct a *BasicStroke* with the correct line pattern activated, by calling the *getDash* method of the *Line* object's *LinePattern* attribute and passing the result as an argument to the *BasicStroke* constructor.

The *smooth* attribute of the *Line* primitive specifies whether the line should be drawn as a series of quadratic Bezier curves, or as a regular line. Figure 20 displays two lines with the same coordinates, the first declared with *Bezier* as its *smooth* attribute and the other with *None*. [5]

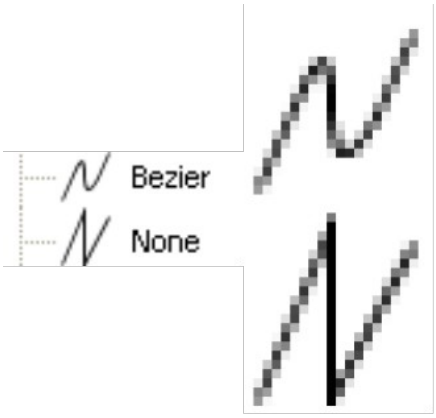


Figure 20: A comparison between the *Bezier* and *None* smooth attributes.

4.2.7 Rendering of text strings

Text strings that are too small to read should not be drawn on icons. Because of this, drawing a *Text* primitive involves making a few different calculations with the aim of making sure that the text string is sized and aligned in the correct way, as well as determining whether to draw the text or not.

First, the font size of the text string is determined. If the graphical annotation that represents the text string has a *fontSize* value of zero (the standard), then the size of the text is scaled to fit the *extent* attribute of the *Text* primitive. This is done by first increasing the font size until the text is wide enough to cover the entire extent horizontally, and then decreasing the font size until the text fits inside the extent vertically. Next, the x coordinate of the text is determined so that it gets correctly aligned in the extent, as specified by the *horizontalAlignment* value of the *Text* primitive. Finally, the actual font size of the text - meaning the font size of the text that will actually be drawn on the screen given the calculations described above and the current transformation - is calculated to check if the text is large enough to be drawn. This is done by multiplying the font size by the scale factor of the current transformation. If the resulting font size is larger than a previously determined value (currently 9), then the text is drawn.

4.2.8 Creating the SWT image

Since the icon to be put to the outline of JModelica IDE must be a SWT *Image*, a method converting AWT *BufferedImage* to SWT *Image* was needed. An icon in the Eclipse outline view is defined as a *model object* and require empty pixels in bottom and on the left of the icon. [10] This is handled by the converting method, *getImage*, of AWTIconDrawer.

The conversion from *BufferedImage* to *Image* starts with extracting the *BufferedImage* object's *ColorModel* attribute. The color model describes how the color data of all of the pixles of the image are stored in the *BufferedImage* obejct. Our *BufferedImages* use a direct color model, which means that all of the ARGB data (alpha, red, green and blue, where alpha represents the level of

opaqueness contra transparancy) of a pixel is represented by a single number, in our case an 32-bit integer. A SWT *PaletteData* object is also created. The PalletteData object is created from the *ColorModel* and is needed to translate the ARGB values of the *BufferedImage* into values that SWT can use. Next, a SWT *ImageData* instance is created. One of the parameters to its constructor is the amount of bits used to represent the ARGB value of each pixel. This number is calculated by calling the color model's *pixelSize* method. The ARGB data for every pixel in the *BufferedImage* is then transferred to the corresponding pixel in the *ImageData* object. This is done by first calling *getRGB* on the *BufferedImage* to obtain the ARGB value of the current pixel, and then calling the *PaletteData* instance's *getPixel* method with the red, green and blue components of the ARGB value. Since each color component is 8 bits long, the four components are read by right shifting the ARGB value 24 times, 16 times, 8 times and zero times respectively, and each time reading the right-most 8 bits by using the bitwise AND operator with the number 255 (eight ones). The color components of the current pixel of the *ImageData* instance are set with the method *setPixel* and the alpha value is set with the method *setAlpha*.

4.3 The *icons.exceptions* java class package

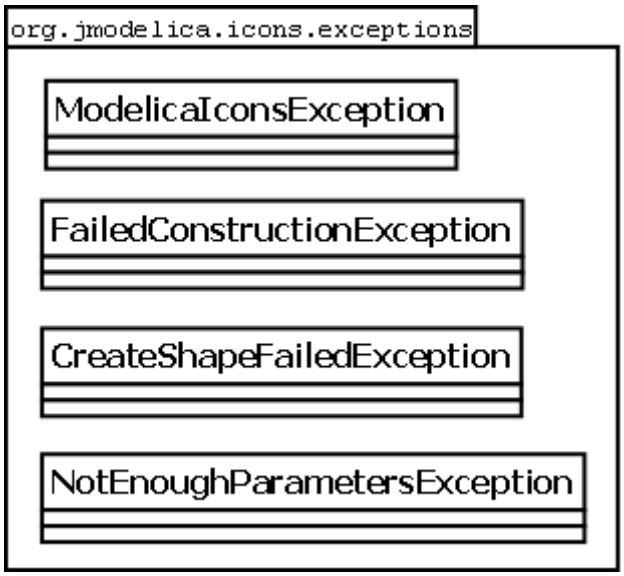


Figure 21: The exceptions used in the project.

The exceptions *NotEnoughParametersException* and *FailedConstructionException* are thrown and caught while iterating over the *AnnotationNode*, whenever the annotation parsing module fails to parse a given annotation record. *CreateShapeFailedException* is thrown when creating shapes of primitives in *AWTIconDrawer,* and are caught when

drawing the shape. They all inherit the *ModelicaIconsException* which in turn inherit *ModelicaException* used in JModelica.org.

4.4 The aspects

In the aspect *ModelicaIcons* the attributes *icon* and *contentOutlineImage* are added to the AST nodes. The aspect *AnnotationParsing* supplies *icon* with values for the graphical elements.

4.4.1 The *AnnotationParsing* aspect

In the aspect *AnnotationParsing* we iterate over the annotations in order to extract the values of the graphical elements. The methods are all private except from *createIconLayer*, *createDiagramLayer*, *hasPlacement* and *createPlacement*. The public methods are called from the aspect *ModelicaIcons* when creating *Layer*s and *Placement*s while the private methods are used inside *AnnotationParsing* to create items needed to build *Layer*s and *Placement*s, such as *CoordinateSystem*, graphical primitives and graphical properties. Most items needed to create a *Layer* or a *Placement* have one or more attribute that must be set while others are optional, i.e they have a default value. When creating an item we initialize an empty object where the default values are set. Then we iterate over the annotation to extract the attributes' values. It would be safer and more intuitive to initialize an item with the attributes as parameters in the constructor. Furthermore the nestled calls when iterating over the annotation are, for some items many and hard to follow. But, since we can not know beforehand which attributes the actual item consists of, initializing an item with the attributes as parameters would require a lot of unnecessary variables to store while iterating over the annotation.

The way to handle a non-existing attribute node and a non-valid value of an attribute depend on whether the attribute for the creating item has a default value or not. For example, the method *createGraphics* returns a list of graphic items and is called when creating a *Layer.* Finding the word "Line" results in calling the method *createLine* which initializes a *Line* with default values and then iterates over the line's attributes. If "color" is found the method "*realVector"* in *AnnotationAPI* is called to return value of the node

- If *realVector* does not contain three doubles or if their value are out of bounds an exception is thrown.
    - If *color* has a default value in *Line* the exception is caught in *createLine* and Line's default value for color is unchanged.
    - If *color* does not have a default value in Line the exception is not caught in *createLine*. The *createLine* is determined and the

exception is send further to *createGraphics* where it is caught and no item is added to the list of graphic items.

- If the values of *color* are correct, then the method *setColor* in *createLine* uses the returned *color* value as a parameter to change the *color* attribute in *Line*.

4.4.2 The *ModelicaIcons* aspect

The attribute *icon* is declared as a synthetic lazy attribute. This means that when an icon has been created for a node, a reference to that icon object is stored on that node. The next time that the icon for that node is needed, the icon object reference is returned. Since Modelica is constructed in an object-oriented way where models are composed of components and the same components exists in several models, the lazy declaration saves memory as well as computing effort. The *icon* attribute is defined as an equation and returns an object of the class *Icon*. If the annotation of the node does not exist there is nothing to be rendered but the icon is created anyway to be able to store inheritance and components. The icon's layer is in this case an empty layer i.e the coordinateSystem and the list of graphic items are null. If a component is added to the icon the empty layer is replaced by a layer with a default coordinateSystem defined in the Modelica Language Specification. The coordinateSystem is needed when the component's transformation is calculated.

Within the attribute the inherited classes and components are added by the two methods *addSuperclasses* and *addComponents*. To prevent the icon rendering threads from entering infinite loops, in the event that models extend themselves or use themselves as components we have added a flag that signifies whether the icon of the *ClassDecl* is currently being calculated, called *visitingDuringIconRendering*. Whenever the icon of a inherited class or component is about to be calculated for a *ClassDecl*, the flag is checked for the *ClassDecl* of the inherited class or component. If it is true, the icon is not created. Models that extend themselves or use themselves as components are not allowed in Modelica, but declaring such models should not make the icon rendering module crash, and this is our way of making sure that this is avoided.

As described in 3.3.2 the node that stores the annotation is the *ClassDecl.* The *ClassDecl* of the current icon stores a list of *ComponentDecls* where each *ComponentDecl* is representing a component. To create the icon of a component we find the *ClassDecl* of the *ComponentDecl* through the aspect *SimpleLookup.* We then make sure that an *Icon* of that *ClassDecl* is created – if the visibility and the restriction of the component respective the class of the component correspond to the conditions described in 3.1.2. A reference to the created icon is then added to the current icon's list of components.

To add inherited classes there is no restriction or visibility to consider. The inherited classes of the current icon are stored in the icon's *ClassDecl* as a list of *ExtendsClause*. For each *ExtendsClause* we get the inherited class's *ClassDecl.* A reference to the icon for that *ClassDecl* is then added to the list of super class icons in the same way as for components.

The *ShortClassDecl* has its own definition of the icon attribute since both the annotation of the *ShortClassDecl* and the annotation of the model that it extends are taken into account.

For the instance AST nodes the icon attribute is added in a similar way as for the source AST nodes.

The *contentOutlineImage* attribute, which returns a rendered icon as an SWT *Image* object, is added to the nodes. To render an icon, an instance of *AWTIconDrawer* is created with the *Icon* object that was created earlier for the node as a parameter. The *contentOutlineImage* attribute contains methods for caching images and icons, as well as mechanisms which allow for the icon images to be created in separate threads in the Eclipse framework, which improves the user experience since it means that the system doesn't freeze while the icons are being created. This functionality was developed by the JModelica.org development team in parallel with our project.

5 Development

This section describes some of the challenges we have encountered during the implementation of the icon rendering.

5.1 Choosing a graphics library

The scope of this thesis when it comes to graphical rendering is fairly limited. We are only concerned with drawing graphical primitives on images, preferably with a simple way of applying transformations for translating, scaling and rotating coordinates without too much trouble. There are several different graphics libraries available for the Java programming language that fully support this. The one we settled for is the Abstract Window Toolkit (AWT) by Sun Microsystems.

Since the start of this thesis, it has been a goal of the thesis project that the software that results from the project will contribute to the JModelica.org project as much as possible and and that it should be possible to reuse our software for various purposes without spending much time on adapting the code. Because of this, it is very important that the graphics library does not put restraints on which platforms the software can be used on. Since the AWT library is built into Java, we decided that it fulfills these needs very nicely.

The strongest competitor to AWT among graphics libraries is - in our eyes - the Standard Widget Toolkit (SWT) by IBM (now maintained by the Eclipse foundation). SWT largely fulfills our needs when it comes to drawing primitives just as well or better than the AWT library. Another obvious reason in favor of choosing SWT for our project would be that all of our software is currently executed in the context of the JModelica IDE, an Eclipse plugin. More specifically, in order to display images in the outlines of the Eclipse IDE, the images need to be passed as instances of IBM's own *Image* class, which is part of SWT. Using SWT as the main graphics library would make this very easy. However, making the icon rendering dependant on SWT would force anyone who wants to render icons in other applications using our code to interact with the SWT library, or rewrite much of the icon rendering code. SWT needs to be distributed with any software that uses it, and additionally lacks support on many platforms. We quickly decided that the benefits of platform independence and easier distribution provided by choosing AWT outweighs the inconvenience of having to convert the AWT image objects into SWT counterparts before passing them to the Eclipse outlines.

5.2 Handling transformations

Making sure that primitives are drawn in the correct size and with the proper rotation and translation of their coordinates, requires different transformations to be made to the coordinates of each primitive before they are drawn on the icon image. Managing these transformations turned out to be one of the greater challenges of the development of the icon drawing software.

Most – if not all – graphics libraries have ways of simplifying coordinate transformations. Developers rarely need to implement the mathematical operations needed to create the matrices which contain the transformations and which coordinate vectors are multiplied by in order to be transformed. Typically, there are methods available in the graphics interface that construct these matrices. This is the case in the AWT library. For example, the AWT *Graphics* class has a method called *rotate* which takes an angle as its parameter and alters the current transformation matrix so that it includes the specified rotation. This means that the act of executing transformations is quite straightforward. The challenge lies in deciding what transformations are needed and in what order to execute them.

Figuring out the correct transformation to apply in order for the icon to fit inside the image was quite a difficult task. As described in chapter 4.2.2, the transformation consists of scale factors and translation increments. There have been problems with the calculation of both of these.

One problem was that we calculated the width and height of coordinate system extents in an incorrect way. A coordinate system that extends from the point (-10, -10) to the point (10, 10) could intuitively be considered to have a width and height of 20 (10 - (-10) = 10 + 10 = 20). However, when a MSL model uses this extent for its coordinate system, it means that both -10 and 10 should be valid x or y coordinates for primitives in the model. This gives a width and height of 21. Since the scale factors are calculated by dividing the icon image's width and height with the width and height of the icon being drawn, this problem caused the scale factors to become too large which meant that some primitives ended up outside of the edges of the icon and were thus not visible. The problem was fixed by adding 1 to the icon width and height as the scale factors were calculated.

Another problem that caused primitives on the edges of icons to not be seen had to do with the rounding of non-integer coordinates passed to the graphics library's drawing methods. From the experiments that we conducted, it appeared that the AWT methods did not use a consistent rounding strategy. Numbers that were rounded down when we ran the program on one computer were rounded up when we ran it on another computer. This phenomenon likely has to with Java2D's conversion of coordinates between *user space* and *device space* [11]. User space is the coordinate system in which the

coordinates of graphical primitives are passed to the AWT drawing methods. Device space, on the other hand, is the coordinate system of the output device such as a monitor, which of course differs between different systems. The conversion between these coordinate spaces is done automatically and is beyond the programmer's control. The way that we solved this problem is that instead of simply applying the transformation to the AWT *Graphics* object and pass the coordinates so that transformation and rounding was done automatically, we kept the transformation object separate from the *Graphics* object and used the transformation to transform the coordinates "manually". This allowed us to access the transformed coordinates and round them ourselves in any way we wanted. Then we could pass the rounded coordinates to the AWT drawing methods, with complete control over how the rounding was done.

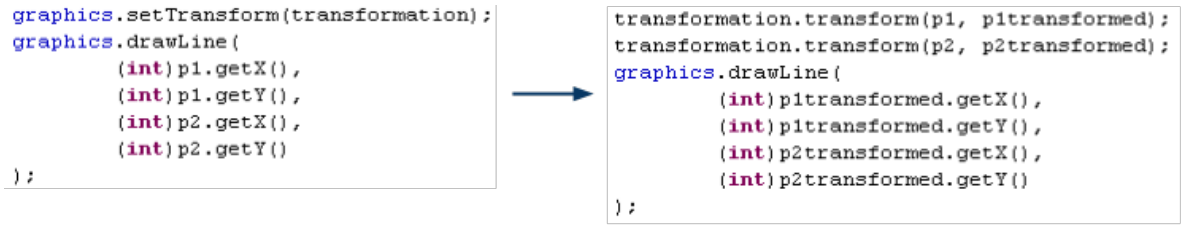


Figure 22: This image shows two different ways of drawing a line between two points represented by Point2D objects (p1 and p2), with the coordinates transformed according to the transformation represented by an AffineTransform object (transformation). In the second example, the transformation is done explicitly in the code instead of implicitly by the Graphics object.

Activating antialiasing requiered half a pixel on the edges of the image. This "margin" made the drawing much less sensitive to rounding errors and differences between the graphics context of different systems and solved the rounding problems. We kept the adjustment – described above – of transforming coordinates manually instead of letting the *Graphics* object transform them as they are passed to the drawing methods. This is a good method of drawing primitives because it allows the programmer direct access to the transformed coordinates, which is very useful when debugging.

5.3 The representation of graphical primitives

When it comes to the graphical primitives specified in the MLS, we decided early on that the AWT *Shape* concept would be useful for representing those of the primitives that extend *FilledShape*: *Rectangle*, *Ellipse*, *Polygon* and *Text*. As explained in chapter 4.2.1, *Shape* is an interface that classes can implement in order to make use of the powerful filling and drawing capabilities of Java2D. There are standard *Shape* implementations for the three geometrical shapes mentioned above (*Text* has a rectangular shape). It seemed logical that the classes for the primitives should have their corresponding

AWT *Shape* as an attribute, and that each primitive should know how to create its *Shape*. This *Shape* object would only have to be created once for every primitive instance, and since it belonged to the icon data structure, it would be cashed along with the rest of the icon representation because of the *icon* attribute being declared as *lazy*. There is a problem with this approach, however. We want our icon rendering software to be as usable by other developers as possible. This includes the ability to make different choices than us regarding which graphics library to use. With this in mind, placing *Shape* objects – which are only useful in the context of the AWT graphics library – on the classes that represent the basic graphical primitives makes little sense (chapter 5.5 further discusses the issue of modularizing the icon drawing). We ended up creating the *Shape* object for a primitive whenever that primitive is drawn, with the logic for creating the different *Shape* objects (and all references to them) being located in our AWT drawing class (AWTIconDrawer).

Another problem we encountered was deciding which *Shape* implementations to use for our primitives. We started out with the most intuitive solution: using java.awt.*Rectangle* for *Rectangles* and *Texts*, *java.awt.Polygon* for *Polygons* and *Ellipse2D* for *Ellipses*. However, we discovered some unpredictable behaviour in the *draw(Shape)* and *fill(Shape)* methods in the *Graphics2D* class. The area that was actually drawn or filled seemed to vary depending on which *Shape*-implementing class that the *Shape* object passed was an instance of, even when it represented the same area. For example, filling a rectangle-shaped *java.awt.Polygon* object would sometimes result in a slightly smaller area being filled compared to when filling a java.awt.*Rectangle* object with the exact same coordinates. For this reason, we ended up making all shapes into polygons before drawing them. Turning a rectangle into a polygon is trivial since rectangles are in theory a subset of polygons. Making polygons from ellipses is only possible by approximating their circumference in line segments. While learning how to do this, we found the method *getPathIterator* in *Shape*, which does exactly this. The method returns an object that iterates over all of the points that make up the shape, approximating the path in case it is a curve. Discovering this method had two added benefits. First, *getPathIterator* can take an *AffineTransform* object as a parameter, which transforms all of the coordinates returned by the iterator with the specified transformation. This was useful when solving the last problem discussed in chapter 5.2 (the unpredictable rounding). Second, the *getPathIterator* can also take a real number argument called *flatness*, which affects the amount of line segments used when approximating curved paths. By tweaking this parameter, we managed to achieve ellipses that looked smoother than the ones resulting from calling *Graphics2D.fill* with *Ellipse2D*

objects as the argument.

The problem with a rectangle and a rectangle-shaped polygon being drawn differently may also have had to do with the fact that we used the class *java.awt.Rectangle* to represent our rectangles. That class stores its coordinates as integer values, while the class we used for polygons stores coordinates as *double*s. This logically should not make a difference, since all coordinates are integers until they are transformed, but we could not rule out that this may have affected the unpredictable drawing behaviour.

5.4 The order of drawing classes

One interesting challenge was to find a way to draw the icons, super classes and components so that their transformations were correctly set in place, at the same time as they were drawn in the correct order. The order in which classes are drawn is significant because primitives are drawn "on top of" each other, in the sense that if two non-transparent primitives have the same coordinates, only the one that is drawn last will be visible.

There are no explicit guidelines in the MLS for which graphical primitives should be drawn first. However, a rather intuitive policy is that super classes should be drawn before the class that inherits them, and that components should be drawn after the class that contains them. Furthermore, the components of a super class should be drawn after the class that inherits the super class.

```
draw(Icon icon) {
      for (Icon superIcon : icon.getSuperclasses()) {
            draw(superIcon);
      }
      for (Icon componentIcon : icon.getComponents()) {
            applyTransformation(componentIcon);
            draw(componentIcon);
            resetTransformation();
      }
      drawPrimitives(icon);
}
```

Figure 23: An early draft of the icon drawing algorithm.

Figure 23 shows pseudocode that describes one of our first algorithms for drawing icons of classes. Using this algorithm, the transformations for the components get correctly applied. However, the primitives of the main class are drawn after the components of the class. This means that the last primitives drawn are the primitives of the main class, which is incorrect since components should be drawn on top of the class that uses them as components. One might be tempted to correct the problem by simply drawing the primitives before drawing the component. However, this would also be

incorrect. The components of the super class would still end up underneath the primitives of the main class.

```
draw(Icon icon) {
      drawClass(icon);
      drawComponents(icon);
}

drawClass(Icon icon) {
      for (Icon superIcon : icon.getSuperclasses()) {
            drawClass(superIcon);
      }
      drawPrimitives(icon);
}

drawComponents(Icon icon)
      for (Icon superIcon : icon.getSuperclasses()) {
            drawComponents(superIcon);
      }
      for (Icon compIcon : icon.getComponents()) {
            applyTransformation(compIcon);
            draw(compIcon);
            resetTransformation();
      }
}
```

Figure 24: The icon drawing algorithm used in the JModelica IDE.

Pseudocode for the algorithm that we currently use is shown in figure 24. It ensures that all of our requirements are met concerning the order of primitives drawn, at the same time as all classes are drawn with the correct transformations in place.

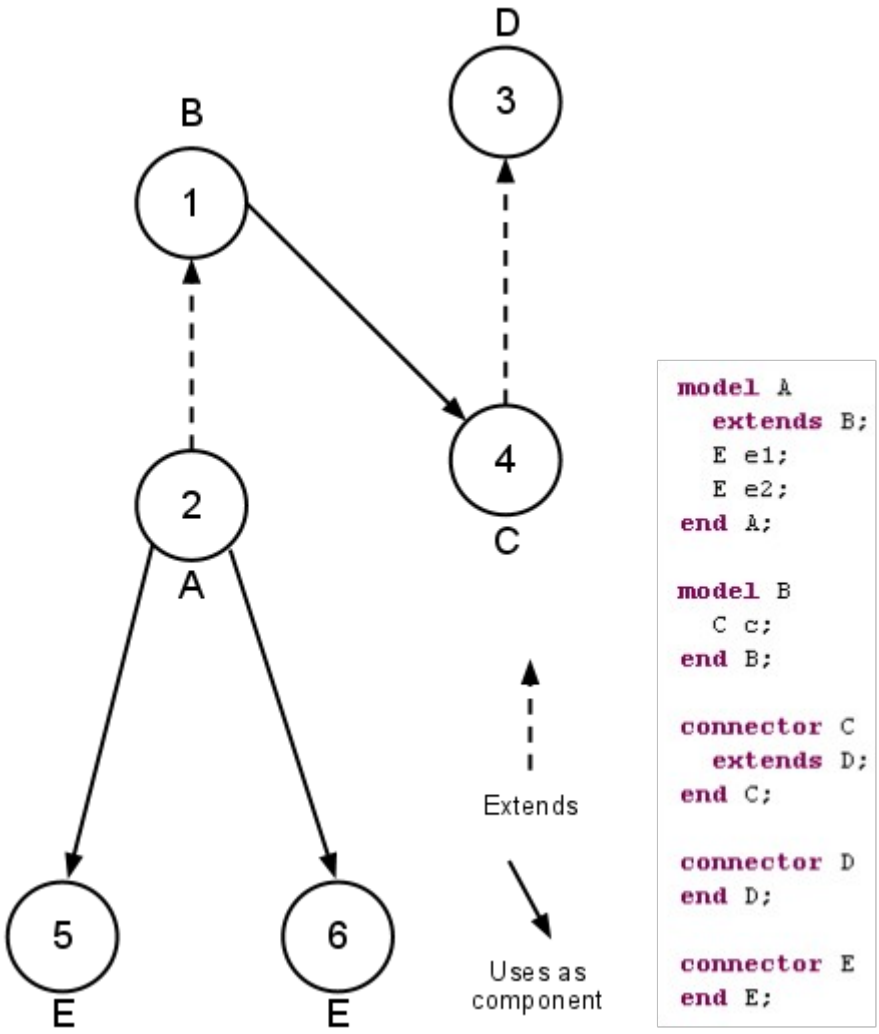


Figure 25: The drawing algorithm illustrated. The Modelica code to the right of the diagram contains the declarations of all the models in the diagram, with everything except for component and inheritance declarations omitted. Numbers signify the chronological order in which the graphical primitives of the models are drawn.

5.5 Creating a graphics interface

As stated before, a major goal with the drawing software was to make it as platform independent as possible, in order to increase the likelihood that future developers can use our code without too much adaptation. A good way of doing this – other than choosing a graphics library like AWT, that runs on many platforms – is to modularize the software and keep the calls to the graphics library isolated in as small a module as possible. We made an attempt to do this by creating a graphics interface class, *GraphicInterface*. The idea is that future developers who wish to use a different graphics library should only have to change a small part of the program in a well-defined way.

As shown in figure 26, our *GraphicsInterface* has methods for drawing the graphical primitives specified in the MLS, as well as for setting the

drawing color and handling transformations. This means that developers who wish to define their own graphical drawing methods only need to implement these.

```
public interface GraphicsInterface {
    public abstract void drawLine(Line l);
    public abstract void drawText(Text t, Icon icon);
    public abstract void drawShape(FilledShape s);
    public abstract void drawBitmap(Bitmap b);
    public abstract void setTransformation(Component comp, Extent extent);
    public abstract void saveTransformation();
    public abstract void resetTransformation();
    public abstract void setColor(Color color);
    public abstract void setBackgroundColor(Color color);
}
```

Figure 26: The JModelica IDE graphical interface.

The *GraphicsInterface* as it currently stands is far from perfect. Ideally, we would have wanted the methods in the interface to be at a lower level, that is to say that they should only draw basic graphical primitives like lines, text strings and basic shapes, and take coordinates as parameters instead of drawing the MLS primitives as they do now. Simply put, the logic that turns the MLS primitives into basic drawing operations should not be AWT dependant. This would decrease the amount of code that any future implementing classes will need to contain. As it is now, large amounts of quite complex logic (see 4.2) will need to be rewritten if someone decides to create a new graphics drawing class that implements our interface.

One example of a method that should ideally look quite different is *drawShape*. The reason that we have one method for drawing shapes instead of one each for ovals, rectangles and polygons is that early on in the development of the drawing code, we realized that the AWT concept of *Shapes* fit very nicely together with MLS. When using AWT, developers can create geometrical shapes of all different types and still draw them all with the same method call to the *Graphics2D* object, as long as the geometrical shapes implement the AWT interface *Shape* [12]. This means that they can be drawn with different border patterns or filled with colors, gradients or textures, in a very simple manner. Since MLS specifies very similar requirements for the *FilledShape* record, it seemed like a very good idea to make use of this construct. All of the features (filling with different patterns, plotting the outline with patterns) could hypotetically be implemented using primitive drawing operations, but it would take a considerable amount of time and effort.

The reason why we haven't gotten as far as we would have liked in this area, is that we had already written most of the drawing code when we realized that making a graphic interface was needed. By the time that we made

the decision to refactor the drawing code, the effort needed to fully do so was too large to feasibly have time for at such a late stage in the project, and other more pressing issues took priority. If we had started out developing the icon drawing code with this need in mind, we would likely have gotten a better result.

5.6 Issues with antialiasing

As described in chapter 4.2.3, we currently use antialiasing for all the drawing on our icon images. We activated antialiasing quite late in the project. The reason for this is that while we experimented with it, we did not manage to achieve the visual results that we were hoping for. The main issue that we had was that when we activated antialiasing, all of the colors of the icons were brighter than normal, giving the icons a blurred look. Since the purpose of antialiasing is to make lines and primitives smoother, we assumed that this might be a problem with AWT's implementation of antialiasing – that it simply worked too well – and that nothing could be done about it short of developing our own implementation. However, we later learned that the problem had to do with the line thickness being too small. The issue was that activating a transformation that has a scaling component affects the line thickness by scaling it with the same amount that all coordinates are scaled. This caused the line thickness to be too small. The reason that we only noticed this problem when we activated antialiasing was of course that with antialiasing disabled, the thinnest line possible is one pixel thick, so if a line thickness smaller than that is activated, it doesn't make any difference. With antialiasing enabled however, lines that are thinner than one pixel result in pixles that get a proportionally smaller part of the line's color. In other words: if the background is white, thinner lines get brighter. After realizing this, we initially fixed the bug by multiplying the line thickness by the inverse of the current scaling factor of the transformation every time we drew primitives. Later however, we eliminated the problem by transforming the coordinates manually, and drawing primitives with no transformation enabled in the *Graphics* object. That way, the correct line thickness was used.

5.7 Determining the line thickness

One issue with the line thickness was that we had a difficult time deciding whether it should be scaled, and if so, in what way it should be scaled. One argument for not scaling the line thickness is that the MLS specifies that line thickness is given in millimeters. This may be interpreted as meaning that a *Line* primitive in a model should always be drawn as thick as is specified in the annotation, regardless of whether the model is actually used as a smaller component inside another model, and regardless of the size of the icon image

that the icon is drawn on. In other words: one could argue that the line thickness is absolute. Upon observing other icon rendering software for Modelica models, it was obvious that this was a common interpretation. However, the problem with this way of handling the line thickness is that it simply does not look good for many models. How good an icon looks is sometimes somewhat of an aesthetic judgement to make, which makes it difficult to form an objective opinion. In other cases, a high line thickness on a small icon simply distorts the icon. For example, if too lines are drawn close together, and their thickness is too high in relation to the distance between them, they will be impossible to distinguish from one another. If this is the case, an icon that is meant to represent one specific model may suddenly look like it represents another model. This is clearly a case of lost value of the icon. Because of cases like this, we decided on the following rule for determining the line thickness: line thickness in annotations should be interpreted as absolute, but there should be an upper limit on the final line thickness which means that lines will never be so thick that they distort the icon that they are drawn in.

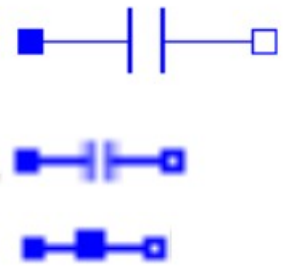


Figure 27: Three versions of the same icon (a capacitor). The upper one is the original version, the middle one is scaled down with the upper limit on line thickness taken into account. The lower version displays what happens if the upper limit on line thickness is ignored.

5.8 Developing with JastAdd

In the final implementation of this project we handle the AST nodes both for extracting information from the annotations and to add the created icons to the nodes. This was not the case from the beginning. Between the aspect *ModelicaIcons* and the icon data structure we had a Java file for creating icons. By putting the attributes to the nodes we did not only get rid of a useless class, we could also derive the advantages of cashing. An advantage with using a static Java class in between the aspects the rest of the program is that it makes some parts of the development easier. For example, debugging JastAdd code can be quite cumbersome. Using a debugging program with breakpoints is very common, and is something that we have done extensively during this

project. The JastAdd structure makes this slightly more complicated however, since breakpoints can not be placed right in the JastAdd aspects files. In order to place a breakpoint at a certain line in an attribute declaration, the developer has to find the Java file that the attribute affects and place the breakpoint at the correct line in that file. This isn't always an easy task, especially since the JModelica.org compiler front end that we have been working with contains hundreds of JastAdd-generated Java files. Using a Java file with static methods that are called from the aspects and does the actual work, gives the developer a convenient place to put such breakpoints. Another property of JastAdd that complicates debugging is the fact that in order for changes made in aspect files to take effect, the Ant script that does this must be run which means that all Java files must be generated anew. While this does not take particularly long (currently about 30 seconds in the case of the JModelica.org compiler), when debugging it is often tempting to make many small changes rapidly and see the result (such as adding data printouts), and this is harder to do if it involves waiting in between each change. Again, debugging by calling methods in a static Java file makes this easier, since Java files are compiled almost instantly.

Since the aspect for *SimpleLookup* was not fully implemented when we started this project we started to work with the instance AST. When we had found out how a Modelica model is build up by inheritance and components and the implementation of the *SimpleLookup* aspect was completed we went over to implement the icon rendering for the source AST nodes.

6 Methodology

6.1 Revision control

In this project we have been working with the Trac system of JModelica.org, connected to a SVN repository. We first worked with a repository of our own, and later on in the JModelica repository. The procedure with Trac is to create a ticket for bugs or things to be implemented. The tickets are visible in a timeline. When a SVN code commit is done, it is customary to include a reference to the ticket that the commit concerns. Conversely, when a ticket has been worked towards, ticket comments can be written which can contain references to the the changesets (commits) that concern the ticket. This way, cross references exist between tickets and commits which have made it easier for us to track changes. Also, working with the Trac system has been helpful in the work of narrowing down the project to realistic problems.

6.2 Source criticism

During the course of the project, we have taken care to make sure that all of the sources that we have worked with have been reliable. For information on how a specific technology works, we have consistently used documents produced by the company or organization that is responsible for the development of that technology. In addition to this, as sources for information on the Modelica language and JModelica.org, we have used two theses written by people who are closely linked to the JModelica.org project. We consider all of our sources to be reliable.

7 Conclusions

7.1 Results

In this thesis we have presented a complete icon rendering process in the context of the JModelica IDE. The process handles all of the steps between graphical annotations (as they are represented by the JModelica compiler) and icon images, ready to be put in the JModelica IDE outlines. In chapter 2, we posed the following question: is it possible to render icons in the outline of JModelica IDE from annotations in Modelica models? As this thesis shows, it is indeed possible to render icons in such a way. The result when it comes to icon quality is comparable to an established commercial product.

Our implementation is available in the distribution of JModelica.org and will be included in the Modelica Workshop for Physical Modeling from Modelon AB.

7.2 Future work

Even though we focused on the rendering of the icons for models, we did put some work into making sure that the *diagram* attribute in our *ModelicaIcons* aspect worked as well as the *icon* attribute. This means that practically all of the graphical rendering of classes that is needed for a full graphical interface is already implemented through this project. We also took other measures to make future work easier, such as trying to modularize our drawing module, as discussed in chapter 5.5.

8 Dictionary

AST: Abstract syntax tree, a data structure used to represent program code as a tree. The nodes in the tree are different constructs found in the source code. ASTs are commonly used by compilers.
AWT: Abstract Window Toolkit. A GUI toolkit that is part of the Java Foundation Classes.
Eclipse: An open-source IDE. All features of Eclipse except for the core functionality consists of small modules called ”plugins”. By creating new plugins, users can easily add new functionality to Eclipse. The JModelica IDE is such a plugin.
IDE: Integrated development environment.
MLS: Modelica Language Specification. A document produced by the Modelica association, defining different aspects of the Modelica language, including its syntax.
MSL: Modelica Standard Library, an extensive and regularly updated library of Modelica classes, free to use by Modelica developers to build models.
SVN: Subversion, an open source program used for revision control.
SWT: Standard Widget Toolkit. A GUI toolkit maintained by the Eclipse Foundation.
Trac: An open-source bug tracking system.

References

[1] JModelica.org, 2011. http://www.jmodelica.org (2011-08-15).

[2] Modelica Association, 2011. http://www.modelica.org (2011-08-15).

[3] Modelon AB, 2011. http://www.modelon.com (2011-08-15).

[4] Jesper Mattsson. *The JModelica IDE: Developing an IDE by Reusing a JastAdd Compiler.* Department of Computer Science, Lund University, Sweden, October 2009.

[5] Modelica Association. *Modelica - A Unified Object-Oriented Language for Physical Systems Modeling Language Specification Version 3.2.* Modelica Association, 2010. https://www.modelica.org/documents/ModelicaSpec32.pdf (2011-08-15).

[6] JastAdd, 2011. http://www.jastadd.org (2011-08-15).

[7] Johan Åkesson. *Tools and Languages for Optimization of Large-Scale Systems*. PhD thesis, Department of Automatic Control, Lund University,Sweden, November 2007.

[8] JModelica.org, 2011. http://trac.jmodelica.org/browser?rev=2664 (2011-08-30).

[9] Josefsson. *The Base16, Base32, and Base64 Data Encoding.* The Internet Engineering Task Force, 2006. http://tools.ietf.org/html/rfc4648 (2011-08-15).

[10] Nick Edgar, Kevin Haaland, Jin Li, Kimberley Peter. *Eclipse User Interface Guidelines Version 2.1.* International Business Machine Corporation, 2004. http://www.eclipse.org/articles/Article-UI-Guidelines/Contents.html (2011-08-15).

[11] Oracle 2011. http://download.oracle.com/javase/tutorial/2d/overview/coordinate.html (2011-08-15).

[12] Oracle 2011. http://download.oracle.com/javase/1.4.2/docs/api/java/awt/Graphics2D.html (2011-08-15).

Appendix A: Graphical annotations in Modelica

Below is a summary of the specification found in the MLS, chapter 17.5, of the graphical annotations in Modelica.

Types

The following table contains the types that are used to express values in the graphical annotations.

String	A text string - a series of characters.
Boolean	A boolean value (true or false).
Real	A real number.
DrawingUnit	The basic (real number) measurement unit, in millimeters.
Point	A two-dimensional point.
Extent	An area defined by two points.
Color	A basic RGB representation of a color.
LinePattern	Describes the pattern of a *Line*, or the pattern of the outline of a shape. Can have the values None, Solid, Dash, Dot, DashDot or DashDashDot.
FillPattern	Describes the pattern that a shape is filled with. Can have the values None, Solid, Horizontal, Vertical, Cross, Forward, Backward, CrossDiag, HorizontalCylinder, VerticalCylinder or Sphere.
BorderPattern	Describes decorations drawn on the border of a *Rectangle*. Can have the values None, Raised, Sunken or Engraved.
Smooth	Specifies whether or not a *Line* should be drawn as a Bezier curve. Can have the values None or Bezier.
Arrow	Describes an arrow at the end of a *Line*. Can have the values None, Open, Filled or Half.
TextStyle	Can have the values Bold, Italic or UnderLine.
TextAlignment	Can have the values Left, Center or Right.

Records

These are the records that make up the graphic annotations.

```
partial record GraphicItem
    Boolean visible = true;
    Point origin = {0, 0};
    Real rotation(quantity="angle", unit="deg")=0;
end GraphicItem;

record CoordinateSystem
    Extent extent;
    Boolean preserveAspectRation = true;
    Real initialScale = 0.1;
    DrawingUnit grid[2];
end CoordinateSystem;

record Icon
    CoordinateSystem coordinateSystem(extent={{-100,-100},
        {100,100}});
    GraphicItem[:] graphics;
end Icon;

record Diagram
    CoordinateSystem coordinateSystem(extent={{-100,-100},
        {100,100}});
    GraphicItem[:] graphics;
end Icon;

record FilledShape
    Color lineColor = Black;
    Color fillColor = Black;
    LinePattern linePattern = LinePattern.Solid;
    FillPattern fillPattern = FillPattern.None;
    DrawingUnit lineThickness = 0.25;
end FilledShape;

record Transformation
    Point origin = {0, 0};
    Extent extent;
    Real rotation(quantity="angle", unit="deg")=0;
end Transformation;

record Placement
    Boolean visible = true;
    Transformation transformation;
    Transformation iconTransformation;
end Placement;

record IconMap
    Extent extent = {{0, 0}, {0, 0}};
```

```
    Boolean primitivesVisible = true;
end IconMap;

record DiagramMap
    Extent extent = {{0, 0}, {0, 0}};
    Boolean primitivesVisible = true;
end IconMap;

record Line
    extends GraphicItem;
    Point[:] points;
    Color color = Black;
    LinePattern pattern = LinePattern.Solid;
    DrawingUnit thickness = 0.25;
    Arrow arrow[2] = {Arrow.None, Arrow.None};
    DrawningUnit arrowSize = 3;
    Smooth smooth = Smooth.None;
end Line;

record Polygon
    extends GraphicItem;
    extends FilledShape;
    Point points[:];
    Smooth smooth = Smooth.None;
end Polygon;

record Rectangle
    extends GraphicItem;
    extends FilledShape;
    BorderPattern borderPattern = BorderPattern.None;
    Extent extent;
    DrawingUnit radius = 0;
end Rectangle;

record Ellipse
    extends GraphicItem;
    extends FilledShape;
    Extent extent;
    Real startAngle(quantity="angle", unit="deg")=0;
    Real endAngle(quantity="angle", unit="deg")=0;
end Ellipse;

record Text
    extends GraphicItem;
    extends FilledShape;
    Extent extent;
    String textString;
    Real fontSize = 0;
    String fontName;
    TextStyle textStyle[:];
    TextAlignment horizontalAlignment = TextAlignment.Center;
end Text;
```

```
record Bitmap
    extent GraphicItem;
    Extent extent;
    String fileName;
    String imageSource;
end Bitmap;
```