MOTOROLA LABS

TITLE:

User Guide for the Graphical Model Editing Framework

SOURCE:

Networks and Infrastructure Research Lab

AUTHORS:

Name	Organization
Kabe VanderBaan	Networks and Infrastructure Research Lab, Illinois
Scott Brodie	MSU Capstone
Jerrid Matthews	
April Noren	
Aman Rastogi	

DATE:

December 11, 2006
Version 2.0

Revision History

Version	**Date**	**Change Description**
1.0	11/27/2006	Rough outline of User Specifications.
1.1	11/29/2006	Added information to the Implementation of Diagram Editor section
1.2	12/03/06	Removed code examples in Implementing a Diagram Editor since it was not needed.
1.3	12/07/06	Added information to the handler area
1.4	12/08/06	Added installation steps.
2.0	12/11/06	Final Release Version

1 Executive Summary

This document describes the user specification of the Graphical Model Editing Framework (GMEF). GMEF is comprised of a set of libraries providing a standard set of tools to enable developers to associate multiple graphical editors with a specific type of model without the overhead producing complex algorithms to handle the modification of a model between editors, and the repetitive cut and pasting of code to reference a model.

GMEF acts as a bridge to associate model types with any editor designed to represent a model graphically. The graphical representations are modifiable and are reflected in the underlying model. GMEF accomplishes this by monitoring the editor for changes. When objects are changed in the graphical editor, it notifies the framework and the framework updates the model with the current changes. This document will define, in detail, the GMEF Framework starting from the defined logic to handle model references.

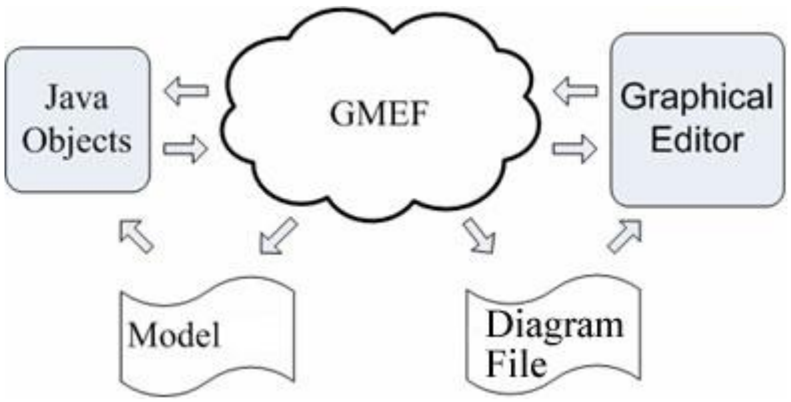


Figure 1 High-level view of GMEF

As Figure 1 outlines, the major functionality of GMEF is described below:

1. Allows users to create a graphical editor using custom shapes to edit or create a model.
2. GMEF enables the graphical editor to access information about the model.
3. The framework relays information between the model and the editor. When a shape is modified in the editor, listeners notify the framework of these changes then decide what action to follow. Finally, the framework analyzes the changes made and invokes a handler that updates the Java object representation of the referenced model. Upon a save, the model changes are saved into a file.
4. The framework is model-agnostic, allowing it to work with any model type that is properly parsed and registered with it.
5. GMEF exports a separate file for storing the graphical data needed to represent a model.

2 Installation

GMEF is an Eclipse plug-in that can be added to any developers project environment. This installation guide includes steps on how to install the GMEF plug-in in the Eclipse environment.

2.1 Eclipse Environment

2.1.1 Java 1.5.06

Download and install J2SE™ Development Kit 5.0 Update 6 (Select JDK 5.0 Update 6, not the NetBeans version)
http://java.sun.com/j2se/1.5.0/download.jsp

2.1.2 Eclipse 3.2

1. Create a c:\proj directory, this will be your workspace for Eclipse
2. Download and install Eclipse 3.2
   http://www.eclipse.org/
   Extract all files to c:\eclipse (i.e., eclipse.exe should be located at c:\eclipse\eclipse.exe)
3. Create a shortcut on your desktop to Eclipse
   Right Click on the desktop, select -> New -> New Shortcut
   Location of Item: C:\eclipse\eclipse.exe -data C:\proj -vm "C:\Program Files\Java\jdk1.5.0_06\bin\javaw.exe"
   Name: Eclipse

2.1.3 Eclipse Configuration

Setup Eclipse to recognize the correct compiler settings
From within Eclipse, select: Window ->Preferences… -> Java -> Compiler -> Compiler Set Compliance Settings to 5.0

2.1.4 Install Graphical Editing Framework (GEF)

1. Within Eclipse, Select Help -> Software Updates -> Find and Install...
2. Select "Search for new features to install" -> Next
3. Select "Callisto Discovery Site" -> Finish -> OK
4. Under "Graphical Editors and Frameworks", select "Graphical Editing Framework 3.2.0.v20060626"
5. This should also install the Draw the Draw2d plug-in.

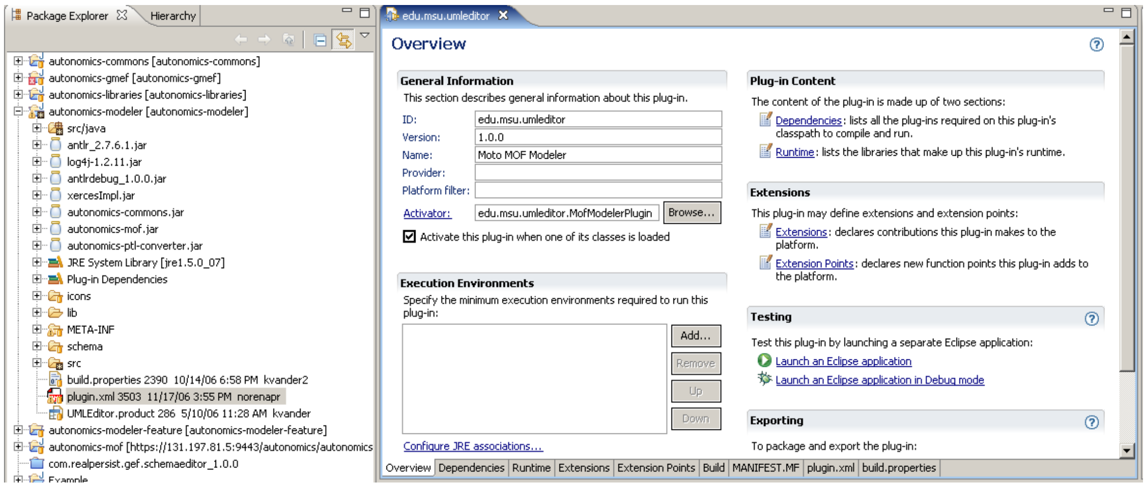
2.2 GMEF Plug-in Installation

To install the GMEF Plug-in:

If your editor is a plug-in:

1. Place the *autonomics.gmef(1.0.0).jar* GMEF Plug-in in the “Plugins” folder of your eclipse installation.

2. Open the plugin.xml file in the “Plug-in Manifest Editor” located under the project folder by double clicking on the Plugin.xml file.
    a. If the manifest editor is not open, right click the plugin.xml file and select “Open With” “Plug-in Manifest Editor”.



```
autonomics-commons [autonomics-commons]
autonomics-gmef [autonomics-gmef]
autonomics-libraries [autonomics-libraries]
autonomics-modeler [autonomics-modeler]
    src/java
    antlr_2.7.6.1.jar
    log4j-1.2.11.jar
    antlrdebug_1.0.0.jar
    xercesImpl.jar
    autonomics-commons.jar
    autonomics-mof.jar
    autonomics-ptl-converter.jar
    JRE System Library [jre1.5.0_07]
    Plug-in Dependencies
    icons
    lib
    META-INF
    schema
    src
    build.properties 2390  10/14/06 6:58 PM  kvander2
    plugin.xml 3503  11/17/06 3:55 PM  norenapr
    UMLEditor.product 286  5/10/06 11:28 AM  kvander
autonomics-modeler-feature [autonomics-modeler-feature]
autonomics-mof [https://131.197.81.5:9443/autonomics/autonomics
com.realpersist.gef.schemaeditor_1.0.0
Example
```

General Information
This section describes general information about this plug-in.

ID:	edu.msu.umleditor
Version:	1.0.0
Name:	Moto MOF Modeler
Provider:	
Platform filter:	
Activator:	edu.msu.umleditor.MofModelerPlugin

☑ Activate this plug-in when one of its classes is loaded

Execution Environments
Specify the minimum execution environments required to run this plug-in:
Configure JRE associations...

Plug-in Content
The content of the plug-in is made up of two sections:
Dependencies: lists all the plug-ins required on this plug-in's classpath to compile and run.
Runtime: lists the libraries that make up this plug-in's runtime.

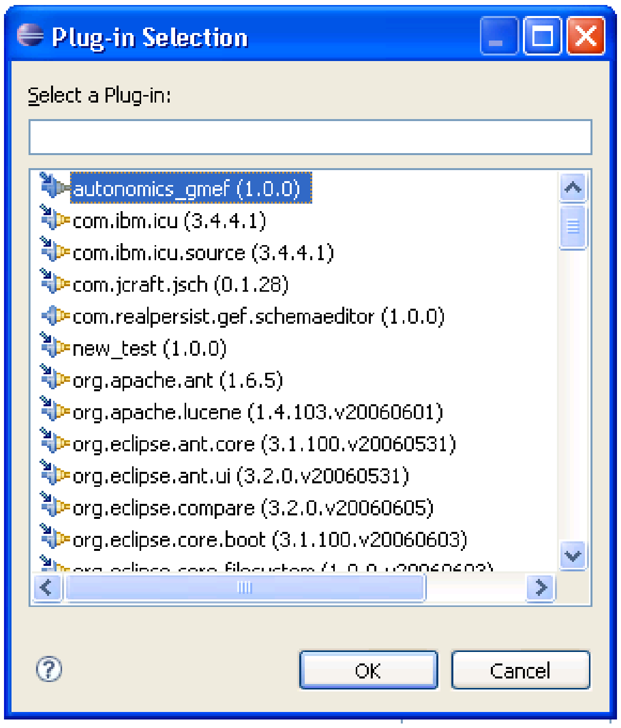
Extensions
This plug-in may define extensions and extension points:
Extensions: declares contributions this plug-in makes to the platform.
Extension Points: declares new function points this plug-in adds to the platform.

Testing
Test this plug-in by launching a separate Eclipse application:
Launch an Eclipse application
Launch an Eclipse application in Debug mode

Exporting
To package and export the plug-in:

Figure 2: Plug-in Manifest Editor

3. Next, select the “Dependencies” tab within the editor, and click “Add” to add the GMEF plug-in into the project via the “Plug-in Selection” tab.



Select a Plug-in:

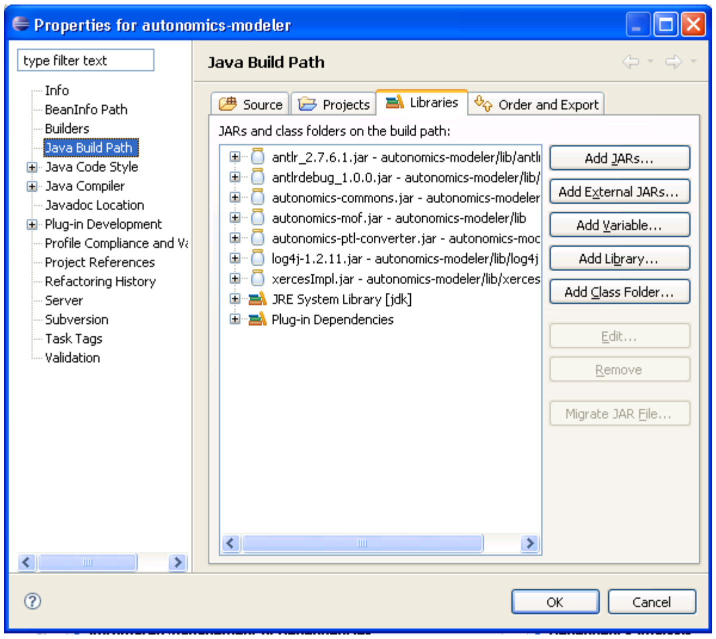
```
autonomics_gmef (1.0.0)
com.ibm.icu (3.4.4.1)
com.ibm.icu.source (3.4.4.1)
com.jcraft.jsch (0.1.28)
com.realpersist.gef.schemaeditor (1.0.0)
new_test (1.0.0)
org.apache.ant (1.6.5)
org.apache.lucene (1.4.103.v20060601)
org.eclipse.ant.core (3.1.100.v20060531)
org.eclipse.ant.ui (3.2.0.v20060531)
org.eclipse.compare (3.2.0.v20060605)
org.eclipse.core.boot (3.1.100.v20060603)
```

Figure 3: Plug-in Selection

4. FINISHED!!!! You are now ready to start using the GMEF Framework plug-in.

<u>If your editor is a Java Project (Not a plug-in project):</u>

1. Place the *autonomics.gmef(1.0.0).jar* GMEF Plug-in in the "Plugins" folder of your eclipse installation.
2. Right click on the desired java project folder, located under the "Package Explorer" tab. Select "Build Path" from the Menu tab. Then click "Configure Build Path" to open the "Properties" editor.



JARs and class folders on the build path:

```
antlr_2.7.6.1.jar - autonomics-modeler/lib/antl
antlrdebug_1.0.0.jar - autonomics-modeler/lib/
autonomics-commons.jar - autonomics-modeler
autonomics-mof.jar - autonomics-modeler/lib
autonomics-ptl-converter.jar - autonomics-moc
log4j-1.2.11.jar - autonomics-modeler/lib/log4j
xercesImpl.jar - autonomics-modeler/lib/xerces
JRE System Library [jdk]
Plug-in Dependencies
```

Figure 4: Java Build Path

3. Select the "Libraries" tab, and then click the "Add External Jars" button. Next, select the *autonomics.gmef(1.0.0).jar* located in the "Plugin" folder of your Eclipse installation.
4. Click OK. FINISHED!!!! You are now ready to start using the GMEF Framework plug-in.

3 Implementing a Custom Diagram Editor

This section describes the classes that are needed to create a custom diagram editor for GMEF. These classes allow the user to add in specific elements that are used to represent specific models. For information on implementation of functions within these classes please refer to the Java Documentation (Java Doc) of GMEF. Additionally, there is an example editor available that can be accessed through the following webpage, https://cse498t05s.cse.msu.edu/.

3.1 Diagram Editor

To create a new editor using the GMEF framework, the developer needs to create a new class that extends `DiagramEditor`. A customized palette can be created by overriding the generic Java palette API's that extend the `DiagramEditor`.

To associate specific model references and figure relationships with an editor, the developer will have to create a model handler that implements GMEF's handler interfaces. In our implementation the class is called `MOFDiagramEditor`.

The following sections will describe how to associate specific handlers with a custom editor.

3.2 Model Handler Factory

To reference a specific model with an editor, the developer must first create a factory that creates instances of model handlers for a type of model. This factory must implement the `IModelHandlerFactory` interface. After creating the model handler factory, the user must then register it to the `DiagramHandler` instance, which is available through the `DiagramEditor` by the protected variable *diagramHandler*.

In our implementation this class is named `MOFModelHandlerFactory` and exists as a Singleton. It is designed this way to allow the instance to be referenced in the `DiagramHandler` class. This instance is registered to the `DiagramHandler` via the method "*addModelHandlerFactory()*".

3.3 Model Handler

Next, the developer will need to create a model handler to allow the GMEF Framework to handle instances of a specific model type. This class must implement the `IModelHandler` interface. To associate the class with GMEF, a model handler factory must exist that creates instances of this model handler.

In our implementation this class is called `MOFModelHandler`. This class is instantiated by the `MOFModelHandlerFactory`, which is accessible through the `DiagramHandler`.

3.4 Figure Handler

To create viewable figures in an editor, the user must first create a figure handler for the type of figure to be represented visually. Figure handlers will be used to create instances of viewable objects in an editor given an element of its defined type from a model and a location to draw the figure in the editor.

To create a figure handler, the user must create a class that implements `IFigureHandler`. A figure must also exist for the `FigureHandler` to use.

In our implementation this class is a singleton, which means only one instance of this class exists at a time. The methods that implemented `IFigureHandler` were used to cast the object into a specific implementation type that is dependent on the model. In the case of the Class Diagram editor using a MOF Model, objects were cast into a Class implementation. The *getFigure()* method not only gets the figure, but it also creates it by setting an `IFigure` to a new `CustomFigure`, which was `ClassFigure` in our implementation.

3.5 Figure

The GMEF Framework contains generic components to build any type of figure representation of a model element in an editor. These components are designed only for figure elements that contain compartments. For example, a UML class contains three compartments, which are Class label, attributes, and operations. Within these compartments are 0 to *n* sub compartments for various elements.

To develop a visual figure in the editor, the user must use the classes within GMEF to develop a Figure. This is done using the hierarchical structure of compartments within GMEF.

In our implementation, a figure was developed to represent a class element from a model. The name of our Figure representing a class within a MOF model was `ClassFigure`. This class was developed first by creating three compartments. One compartment was created for the title of the figure. There were two other compartments created for the attributes and operations of the class.

<u>To create the Title compartment:</u>

1. The `CompartmentFigure` class was extended to create a `TitleCompartmentFigure` class for formatting the layout of the class label compartment.
2. After creating the `TitleCompartmentFigure` class, this class is the first compartment created within GMEF to hold the class name. After creating the title compartment was created.

3.6 Connection Handler

Connection handlers are used to create instances of relationships in the diagram editor given a type of relationship from a model, and a location to draw the relationship in the editor. The handler is designed as a single instance, which will be used by an edit part to instantiate instances of a type of relationship from a model. All connection handlers must be registered to each `ConnectionTypeEditPart` instance for visualization of a relationship in the diagram.

In our implementation, a connection handler was developed called `ConnectionFigureHandler` to create figure representations of a type of relationship in a model. One type of figure relationship instance that is instantiated by a handler is an aggregation relationship. These figure instances are created through the `AggregationRelationshipFigure` class. The `AggregationRelationshipHandler` creates instances of the `AggregationRelationshipFigure` for every instance of this type of relationship in the model.

To create a connection handler;

1. The user must implement the `IConnectionHandler` interface as a singleton class, defining custom logic to create instances of the figure representation of the relationship desired.
2. After implementing the interface to create the connection handler, the user must register the connection handler to the `DiagramEditPartFactory` `via the` *addConnectionHandler()* `method`. The `DiagramHandler` is accessible via the protected data member "*diagramHandler*", located in the `DiagramEditor` class.

   Example:
   diagramHandler.getEditPartFactory().addConnectionHandler (*YourHandler*.getHandlerType(), *YourHandler*.getInstance())

3.7 Connection

A Connection is an element of the Diagram Model that visually displays relationships between figures within a model. An example of a connection within a model that would be referenced is an inheritance relationship between two classes.

The GMEF Framework contains generic components to build the figure representation of relationships within a model. To develop a visual relationship in the editor, the user must first extend the `PolylineConnection` class, and then create custom logic to define the visual representation of the relationship.

To create a Connection figure:

1. The user must implement a class that extends GEF's `PolylineConnection` class.
2. Next, define custom logic to create the desired graphics (e.g.: dotted line relationship, solid line relationship, etc.) and decorations for the type of relationship figure.

3.8 Drop Handler

The purpose of the Drop Handler is to allow developers to specify what types of objects are to be dropped on the editor and how to handle those elements. The class that is created to perform this task needs to implement `IDropHandler`. For GMEF to use this handler, it must be added in the developers' custom editor.

In our implementation, this class is a singleton. For the MOF diagram editor, we used the class `MOFDropHandler` to handler the objects that were dropped in the editor. This class checked the object that was dropped and if it were a Class the `MOFDropHandler` would cast it appropriately.

4 Diagram File Format

GMEF uses a custom file format called MOF Model Diagram (mmd). The mmd file format holds xml formatted data that any editor extending GMEF can read and save to.

An example diagram file might look like the following:

```
<?xml version="1.0" encoding="UTF-8"?>
< gmef:diagram>

<figure gmef:elementReferenceID="434" gmef:figureTypeID="06804">
<gmef:point gmef:x="0" gmef:y="0"/>
<gmef:model gmef:type="MOF" gmef:path="C:\DemoModel.mof"/>
</gmef:figure>

<figure gmef:elementReferenceID="123" gmef:figureTypeID="25073">
<gmef:point gmef:x="0" gmef:y="100"/>
<gmef:model gmef:type="MOF" gmef:path="C:\DemoModel.mof"/>
</gmef:figure>

<diagramRelationship gmef:connectionTypeID="67865">
<gmef:source>06804</gmef:source>
<gmef:target>25073</gmef:target>
</diagramRelationship>
```

Explanation:

Each file contains a **<diagram…>** tag that encapsulates all objects contained within it. The figures are denoted by the **<figure… >** tag. Each figure contains its location within the diagram, as well as information about the model that it comes from.

There may also exist a relationship between the two figures, which is denoted in the **<diagramRelationship…>** tag. A relationship holds the information about the connection between any two figures within the diagram. Relationships are not always present within a diagram file, however, as many types of relationships can be derived directly from the Figure data by itself. Typically a relationship is only present if it contains specialized drawing information, such as bend points or special decorations. See the GMEF Technical Specification Document for further information on how GMEF handles this information.

5 Editor Feature Outline

This section highlights the major functionality included in an editor that properly extends GMEF. The purpose of this section is to demonstrate how to access and use the features that GMEF offers.

5.1 *Drag and Drop*

GMEF allows for Drag and Drop of elements from a model navigator to the editor diagram. This is done by selecting the model element, clicking on it, and dragging it across the screen and releasing on the editor. Upon drag and drop of the element, the user also has the ability to Undo or Redo the action.

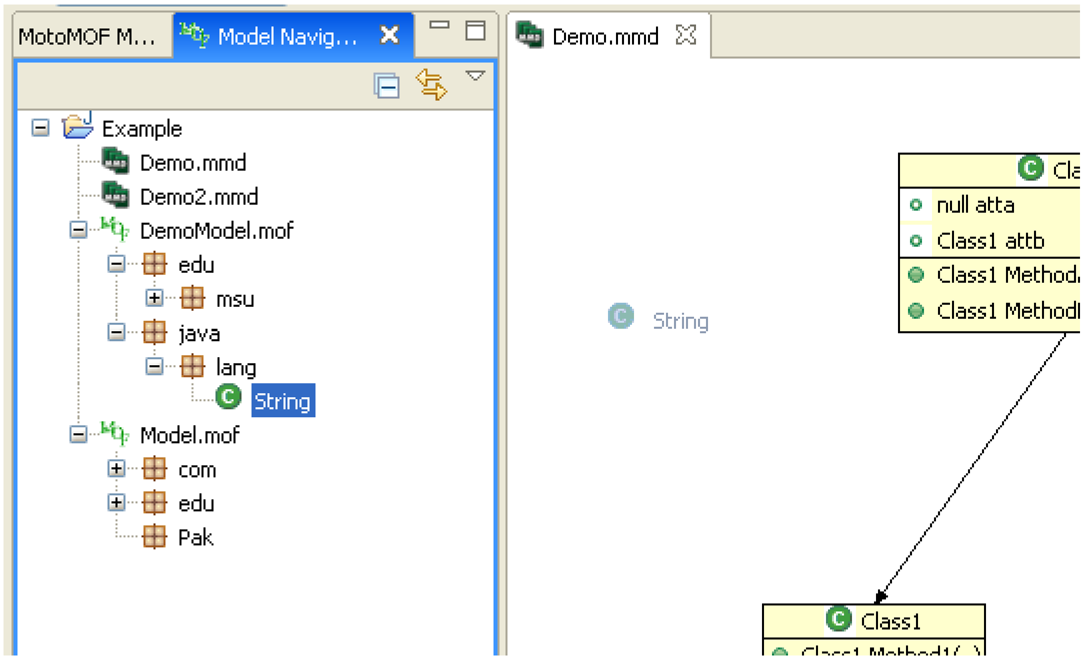


Figure 5: Drag and Drop

5.2 *Palette Tools*

Any editor extending GMEF comes equipped with a FlyoutPalette, which is used to hold tools for manipulating a diagram. While the palette itself is customizable, GMEF initially supports the following basic tools:

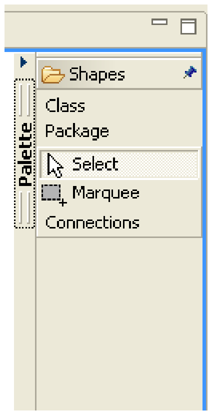


Figure 6: Palette

- Folder – A container used for organizing palette tools.
- Selection – Place the focus on a particular figure within the diagram by simply left clicking within the figure's boundary.
- Marquee – Place the focus on one or more figures within the diagram by clicking and dragging a bounding box around the figures boundaries.
- Connection – custom implementation needed.

5.3 Movable Figures

Figures that are created on the diagram have the ability to be selected and moved by the user. This allows for easy placement and manipulation of diagrams. After moving the figure, GMEF has the ability of Undo and Redo of the previous state.

5.4 Deletion of Figures

Figures can be deleted from the diagram in a variety of ways. After the figure object is selected, the user can press the "Delete" key on the keyboard, select the red "X" on the top toolbar, or select "Delete" from the drop down menu.