CHAPTER 4

Meta-Modeling of UML Techniques

The development of a CASE tool supported by UML techniques is done by using two types of metaCASE tools, MetaEdit Personal 1.2 and MetaEdit+ 2.5. This CASE tool is named as UMLCASE. This chapter is divided into two parts, the first part describes the meta-modeling process using MetaEdit Personal 1.2 and the second part describes the meta-modeling process using MetaEdit+

6.1 Method Modeling Using MetaEdit Personal 1.2

In MetaEdit Personal 1.2, the meta-modeling has to go through three steps, as follows:

1. Design of an OPRR model for each UML technique. In this stage the OPRR model will be generated to form the method definition file (*.mta file). Further elaboration and discussion will be explained in details in the following part.

2. Modify the *.mta file by adding the meta-model header, shape definition and symbol definition. After compiling using a method compiler, the *.mof file for the method to be used will be produced.

3. Test the *.mof file to check whether it has fulfilled the notation of UML. Otherwise, the *.mta file needs to be changed and recompiled.
4.2 The Structure of the Meta-Metamodel of MetaEdit

Figure 4.1 shows that MetaEdit is capable of defining three different levels: the meta-metamodel, the meta-model and the model. MetaEdit itself has a meta-metamodel which has capability to define the syntax and the semantics of a meta-model supported by the MetaEdit. The meta-metamodel level provides the flexibility and extendability of MetaEdit. The meta-model in this case, can be UML, OMT, Booch method, or other methods. The model is used to draw a diagram using one of the techniques that have been provided by the meta-model such as a Class Diagram of UML.

![Diagram of MetaEdit](image)

Figure 4.1 – The Meta-Metamodel of MetaEdit [9]

4.3 The OPRR Modeling Process

In creating a new technique or method, the understanding of the conceptual contents of the technique to be modeled is imperative as the first step. As such, the objects and their relationships need to be identified.
For instance, in a Use Case Diagram, actors and use cases are the objects of the Use Case Diagram. For the relationships between these objects, a communication relationship between an actor and a use case, and a uses or extends relationship between use cases are needed. Then the representation of the technique needs to be specified by using the OPRR notations. The OPRR notations have been described in Chapter 2. The modeling file will be saved as *.opr file as given below:

\[\text{C:/Metaedit/models/}.opr\]

### 4.3.1 Meta-Model of the Use Case Diagram

The OPRR modeling shown in Figure 4.2 describes the meta-model of Use Case Diagram. An Object, such as an actor or a use case is represented by a rectangle notation and a Relationship by a diamond. A Role is used to link between an object and a relationship. It helps to clarify how an object participates in a relationship. The concept is the same when linking an object to another object such linking an actor to a use case. A Role is represented as a circle. For each object, an identifier that has a unique value is needed. In the diagram above, the property [identifier] is represented as a double ellipse linking to a thick solid line. This identifier is unique and can only be used for that object. The two main objects in this meta-model are actor and use case. The relationships involved in this technique are communication, extends, and uses.
Figure 4.2 – The OPRR Modeling of the Use Case Diagram

4.3.2 Meta-Model of the Class Diagram

The OPRR modeling described in Figure 4.3 shows a part of the meta-model of Class Diagramming technique. A complete OPRR modeling specification can be referred in Appendix A at the end of this thesis. In the Class Diagram, the main objects are class, package, interface, metaclass, and object. Each object is linked to various types of relationships, such as association, dependency, composition, aggregation, and generalization. Every object has its own properties that attached to it.
4.3.3 Meta-Model of the Sequence Diagram

The OPRR modeling shown in Figure 4.4 describes the notation of the Sequence Diagram. The objects are objectbox, activation and termination. The relationships involved in this technique are transition and interaction. The transition relationship occurs between the activation and the interaction relationship occurs between the objectbox and the activation.
6.3.4 Meta-Model of the Collaboration Diagram

Figure 4.5 describes the elements and their relationships that form the meta-modeling of the Collaboration Diagram. The main objects are classrole, messageflow, multiobject and actor. A few types of relationships that link the classrole and messageflow, are procedure call, control all and asynchronous call. An association relationship is required to link an actor with the classrole while the composition relationship is used between the multiobject and the classrole.
4.3.5 Meta-Model of the State Diagram

The OPRR modeling shown in Figure 4.6 describes the elements that form the meta-model of the State Diagram. The main objects are state, final state, initial state and synchronization bar. The State Diagram uses a transition relationship to link the objects.
4.3.6 Meta-Model of the Activity Diagram

The meta-model of the Activity Diagram is shown in Figure 4.7. The main objects are activity, start, end and synchronization bar. Similar to the State Diagram, the relationship used is a transition relationship.

![Activity Diagram](Image)

Figure 4.7 – The OPRR Modeling of the Activity Diagram

4.3.7 Meta-Model of the Component Diagram

The meta-model of the Component Diagram is shown in Figure 4.8. The main objects are component and interface. This technique only uses one type of relationship, which is dependency relationship. Each object has a few properties that help to describe the object.
3.8 Meta-Model of the Deployment Diagram

The OPRR modeling for a Deployment Diagram shown in Figure 4.9 is quite similar to that of Component Diagram. The main objects are node, component, object and interface. The objects such as component and node can link to some other nodes using a dependency relationship. The art_of_relationship is to define the instances of objects. The double ellipse is used to describe the identifier for each object.
Figure 4.9 – The OPRR Modeling of the Deployment Diagram

4.4 The Method Definition Tools

MetaEdit Personal 1.2 provides the following two alternatives for producing method definitions.

1. Writing all the definitions manually.
2. Modeling the conceptual structure of a technique using the OPRR notations.

In the first alternative, the declaration for the shapes, symbols, properties, objects and relationships must be defined from the initial state and these definitions are not based on the OPRR modeling. Since the objective of this project is to apply the meta-modeling approach, the second alternative has been used. Therefore, all the techniques modeled are using OPRR notation which will be automatically generate the method definitions. This can be done by using a special report called MethodDefinition. Even though the output or the produced code is syntactically correct but the additional definitions are required to run the techniques successfully. As such, some other definitions for graphical representation such as shape definitions and symbol definitions need to be added. The code produced will be named as a *.rpt file, and saved as a
The mta file. This step is necessary in producing the corresponding method in the *.mof file generated by using the method compiler. The method definition given in the Figure 4.10 is incomplete as it is based on a Use Case Diagram which has been generated from the OPRR specification (use.opr). This method definition is lack of a meta-model header, shape definitions and symbol definitions. The MethodDefiniton tool however, is only capable of generating a partial part of a report from the OPRR modeling which are property types, object types, role types and relationship bindings. The header, shape and symbol definitions must be added to this method definition in order to ensure the techniques run successfully.

    property type "Actor name"
    { datatype String;
      values unique;
    }

    property type "Use case"
    { datatype String;
      values unique;
    }

    property type "Documentation"
    { datatype Text;
    }

    property type "Label"
    { datatype String;
    }

    object type "Actor"
    { symbol "Stickman";
      duplicates allowed;
      explodes;
      properties("Actor name","Documentation")
    }

    object type "Use case"
    { symbol "Ellipse";
      duplicates allowed;
      explodes;
      properties("Use case","Documentation")
    }

    relationship type "Communicates"
    { directed;
      line type solid;
      line width 1;
      colour black;
    }
Figure 4.10 – The Incomplete Method Definition
Figure 4.11 gives a complete method definition that can be compiled as a .mof file and to be used as a new method in MetaEdit. The method definition includes shape definitions for all shapes used in the Use Case Diagram and symbol definitions for the shapes. The method definitions (*.mta file) of other UML techniques can be referred from Appendix D.

```
meta-model "Use Case Diagram UML"
extension "USE"

shape "Rectangle"
{ shape (0@0, 200@0, 200@200, 0@200, 0@0);
  connection points (0@0, 200@0, 200@200, 0@200, 0@0);
  line type solid;
  line width 3;
  colour darkgreen;
}
shape "Line"
{ shape (100@100,107@100);
  line type solid;
  line width 2;
}
shape "Circle"
{
  shape (130@130);
  line type solid;
  line width 2;
  colour black filled black;
  connection points (50@100, 50@89, 60@70, 70@59, 90@50, 110@50,
  130@59, 139@70, 150@89, 150@100, 150@109, 139@129, 130@139,
  110@149, 90@149, 70@139, 60@129, 50@109);
}
shape "Straightline"
{ shape (100@130,100@180);
  line type solid;
  line width 2;
}
shape "Handline"
{
  shape (80@140,120@140);
  line type solid;
  line width 2;
}
shape "Legline"
{
  shape (70@190,100@180,130@190);
  line type solid;
  line width 2;
}
shape "Invisiblerectangle"
{ shape (0@85,200@85,200@120,0@120);
  line type invisible;
```
connection points (0@85,200@85,200@120,0@120);
}

shape "Ellipse"
{ shape (200@150);
  line type solid;
  line width 2;
  colour red;
  connection points (100@150, 40@145, 5@120, 0@100, 5@80, 40@55, 100@50, 160@55, 195@80, 200@100, 195@120, 160@145);
}

shape "Arrow"
{ shape (100@100,90@95,90@105);
  line type solid;
  line width 0;
  colour blue filled white;
}
symbol "Stickman"
{ shapes ("Circle", "Straightline", "Handline", "Legline"); scale 0.5;
  labels
  { "Name" at (200 180 180 200) centered;
  }
}
symbol "Rectangle"
{ shapes ("Rectangle");
  scale 2.0;
  labels
  { "System Name" at (0 10 190 0) centered;
  }
}
symbol "Ellipse"
{ shapes ("Ellipse");
  scale 0.4;
  labels
  { "Name" at (20 60 180 140) centered;
  }
}
symbol "Arrow"
{ shapes ("Arrow");
  scale 1.0;
  labels {"Label" at (10 30 100 100) left;}
}
symbol "Line"
{ shapes ("Line");
  scale 0.4;
}
symbol "Invisiblerectangle"
{ shapes ("Invisiblerectangle");
  scale 0.4;
  labels {"Note label" at (5 85 195 120) left;}
}
property type "Note label"


```
{ datatype String;
    values unique;
 }

property type "Label"
{ datatype String;
 }

property type "Name"
{ datatype String;
    values unique;
 }

property type "System Name"
{ datatype String;
    values unique;
 }

property type "Description"
{ datatype Text;
 }

object type "Note"
{ symbol "Invisiblerectangle";
    duplicates allowed;
    properties ("Note label","Description");
 }

object type "Box"
{ symbol "Rectangle";
    duplicates not allowed;
    explodes;
    properties ("System Name","Description");
 }

object type "Actor"
{ symbol "Stickman";
    duplicates allowed;
    explodes;
    properties("Name","Description")
 }

object type "Use case"
{ symbol "Ellipse";
    duplicates allowed;
    explodes;
    properties("Name","Description")
 }

relationship type "Communicates"
{ directed;
    line type solid;
    line width 2;
    colour black;
 }

relationship type "Extends"
{ directed;
    line type solid;
    line width 1;
    colour blue;
 }

relationship type "Uses"
{ directed;
 ```
Figure 4.11 – The Complete Method Definition

4.1 The Method Compiler

The method compiler is a text editor that is capable of compiling its contents into a binary method definition. The compiler consists of two parts as follows:
- The editing area that is used to write or edit method definitions.
- The menu-bar which contains the action that can be taken for processing the definitions.

The *mta file is the output of the method compiler. The MOFF menu that used to check syntax and consistency of the method definitions is attached on the menu bar. Whenever the compilation runs, a message window will appear and the status of information related to the compilation will be displayed continuously. An error message will be displayed on the message window for any error encountered. The *mta file must be saved so that any modification can be done to the same file. For this project, the class.mta file is saved in a directory as follows:

C:\Metaedit\methdev\class.mta

The class.mta file can be compiled once again into a class.mof file and saved in a particular directory as follows:

C:\Metaedit\methods\class.mof

4.2 Notation

In order to create a new method or technique, it is important to know the notation that must be used in the method definition.

User Defined Names

All user-defined names must be enclosed in quotes, such as symbol "Rectangle". The method compiler will not recognize the name without the quotes.

Colour Definition

MetaEdit Personal 1.2 supports several types of colours and it can be used to define the presentation of concept. The colours are:
Points

Points are important in defining a particular shape for any concept, such as for a use case, the shape will be an ellipse. To create an ellipse, the points must be represented using integers and @ sign as a separator between x value and y value in the form of a coordinate.

Example: \( x@y \), \( x \text{ value is } 100 \text{ and } y \text{ value is } 120 \), the points will be 100@120.

Points are mapped onto 200 x 200 coordinate system. The center of the coordinate is 100@100 as shown in Figure 4.12.

![Figure 4.12 – The Point Coordinates](image)

The coordinates that form the shapes for the techniques must be defined. For instance, the shape definition in Figure 4.13 shows the points for an ellipse shape. Other definitions are line type, line width, colour type and connection points. The connection points are the points that will define the shape to which the connectors can be attached. In MetaEdit Personal 1.2, all shapes used in the techniques must be defined using points.
shape "Ellipse"
{
  shape (200@150);
  line type solid;
  line width 2;
  colour red;
  connection points (100@150, 40@145, 5@120, 0@100, 5@80, 40@55,
  100@50, 160@55, 195@80,
  200@100, 195@120, 160@145);
}

**Figure 4.13 – The Method Definition of Shape**

4. Line Types

A few line types are supported, such as:

- **Solid line**

- **Dash line**

- **Dot line**

- **DashDot line**

- **DashDotDot**

- **Invisible line**

These line types can be used to define the types of relationships such as, *dependency relationship* is represented by a dashed line.
4.4.3 The Ordering of Definitions in Method Definition Language

The ordering of definitions in the method definition is very important. The method definition must be defined in the following order:

1. Meta-model header

The meta-model header must be defined on the first line of the method definition (*.mta file). It states a long name for the method definition. The next line is the declaration of the extension of the method definition. This extension is used as a default and must be specified in three characters, "USE", as shown below:

```
meta-model "Use Case Diagram UML"
extension "USE"
```

2. Shape definitions

The definition in Figure 4.14 shows the shape of a use case is an ellipse. In this case, only one point of coordinate is needed for an ellipse shape.

```
shape "Ellipse"
{
  shape (200@150);
  line type solid;
  line width 2;
  colour red;
  connection points (100@150, 40@145, 5@120, 0@100, 5@80, 40@55, 100@50, 160@55, 195@80, 200@100, 195@120, 160@145);
}
```

**Figure 4.14 – The Shape Definition**

This correct coordinate for a shape must be configured. Some examples of shape definition are given below:
shape (200@150); This shape defines an ellipse.

shape (0@0, 200@0, 200@200, 0@200, 0@0); This shape defines a rectangle.

type solid; The line type can be any of the line types that is supported by

    MetaEdit

line width 2; The line width must be a non-negative value. It can be started with 0

    and above.

colour red; The colour for the shape.

connection points (100@150, 40@145, 5@120, 0@100, 5@80, 40@55, 100@50,

    160@55, 195@80, 200@100, 195@120, 160@145);

This connection points are the points that define the polygon to which connectors

    can be attached.

3. Symbol definitions

For each shape definition, its symbol needs to be defined. For instance, the symbol can be scaled

    between 0 and 1 to define the ellipse shape. This is to determine the size of an object. As for the

    labels, the symbol can have a list of labels that define the property types with their values written

    in the object.

    symbol "Ellipse"

    {
        shapes ("Ellipse");
        scale 0.4;
        labels
        {
            "Name" at (20 60 180 140) centered;
        }
    }

    Figure 4.15 – The Symbol Definition of Ellipse Shape

The symbol definition given below defines the shape of a “stickman”. The “stickman” is a

    combination from several shapes such as Circle, Straightline, Handline and Legline. These

    shapes are defined separately in the shape definition.
symbol "Stickman"
{"shapes ("Circle","Straightline", "Handline", "Legline");
 scale 0.5;
 labels
{ "Name" at (200 180 180 200) centered;
}
}

Figure 4.16 – The Symbol Definition of Stickman

4. Property types

The property definition can be generated automatically from the OPRR modeling. The two property definitions are given in Figure 4.17.

property type "Name"
{ datatype String;
 values unique;
}

property type "System Name"
{ datatype String;
 values unique;
}

Figure 4.17 – The Property Definition

In the property definition, there are a few types of datatypes supported by MetaEdit 1.2, such as Integer (Int), String, Boolean, Character (Char), Text, list of strings and multilist, reference and editlist. The value is unique because this property is used as an identifier and so it can only be used by the first object. It is represented as a double ellipse in the OPRR modeling.

Object types

The object type definition is also automatically generated from the OPRR modeling. An example of an object definition is given below:
object type "Actor"
{ symbol "Stickman";
duplicates allowed;
explodes;
properties("Name","Documentation")
}

Figure 4.18 – The Object Definition

The name of the object type must be put in quotes such as “Actor”. The symbol for an actor is declared in the object type. The symbol definition must use the same symbol defined in the object type definition. In the definition, it is specified duplicates allowed, this means that the object can be used as many times as the user wishes. If duplicates not allowed is specified, the object can only use once in a model. The object types also have their own properties such as name and documentation.

5. Relationship types

The relationship type definition defines the type of relationship that exists between the objects. The relationship can be directed or not directed. If it is specified as directed, then the user must elect objects for its roles in a certain order. The line type for the relationship also needs to be defined, in order to ensure the notation is correct. The line width of the relationship can be specified using the line width. The colour is used to define the colour of the line. A relationship can also have a property. An example of a relationship definition is given below:

relationship type "Communicates"
{ directed;
  line type solid;
  line width 2;
  colour black;
}

Figure 4.19 – The Relationship Definition
7. Role types

A few examples of role type definitions are given in Figure 4.20. A role type can have a symbol that can be attached to an object. The connectivity definitions (0,M) define that an object is bound to this role type must have at least 0, but not exceeding M number of roles of this type. Character “M” is used to define the maximum connectivity.

```
role type "Communicate From" of "Communicates"
{ connectivity (0,M);
}

role type "Communicate To" of "Communicates"
{ connectivity (0,M);
}

role type "Extend To" of "Extends"
{ symbol "Arrow";
connectivity (0,M);
properties ("Label");
}
```

Figure 4.20 – The Role Definition

8. Relationship Bindings

In a relationship binding, the relationship type is bound together with its role types and the object types. According to the example below, the actor is communicating with the use case. A few objects that relate to the relationship can be added if needed.

```
bind relationship "Communicates"
as "Communicate From" ("Actor"),
as "Communicate To" ("Use case")
```

Figure 4.21 – The Relationship Binding Definition
4.5 The Method Modeling Using the MetaEdit+ 2.5

MetaEdit+ provides a repository to store predefined projects that have been developed using various types of methods. Some of the methods supported by MetaEdit+ are OMT, Booch, Fusion, MOSES, and UML. The repository also contains information about the methods such as the elements and properties. In this case, the MetaEdit+ provides a method engineering tool (Method Workbench) that allows the user to modify a method and its concepts, its graphical representations, the dialogs and the reports.

Any changes to the method definition can lead to changes to the models that are based on that method. As such, all techniques are automatically updated once the method definition is changed.

As for UML, MetaEdit+ has several predefined techniques, such as Class Diagram, Use Case Diagram, Collaboration Diagram, State Diagram, Component Diagram and Deployment Diagram but not the Activity Diagram and the Sequence Diagram. Unfortunately all these techniques are incomplete, and a few elements need to be added to fulfill the notation of UML.

4.5.1 The Environment Management Tools

![Figure 4.22 - The Startup Launcher](image)

When MetaEdit+ is started, it begins loading the Startup Launcher. The user needs to login before using the application. MetaEdit+ provides a default user name and a password. A new login account can be created in the repository. After login into the system, the Launcher appears on the screen as shown in Figure 4.22.
Figure 4.23 shows a Launcher which consists several components related to the meta-modeling. The main tools used in the modeling process are Property, Object, Relation, Role and Graph tool. The third level describes the editors which are, Diagram editor, Matrix editor and Table editor.

![Launcher](image)

**Figure 4.23 – Launcher**

The Diagram editor is a tool for creating, managing and maintaining graphs as diagrams. To open the Diagram editor, click on the Diagram icon of the Launcher. Figure 4.24 shows the Use Case Diagram. The steps to create a diagram are given as follows:

1. Click the Diagram Editor icon.
2. Select the item ‘Create New Graph’.
3. Choose the method such as UML.
4. Fill in the properties and the Diagram Editor will open the new graph.

![Diagram Editor](image)

**Figure 4.24 – Diagram Editor**
The Matrix editor is a tool used to handle graphs represented in the form of matrices that contain two axes and related cells. Each element, either in an axis or in a cell can be changed. The modification of the elements will be stored in the repository. Figure 4.25 shows the matrix representation for the Use Case Diagram. The types of elements can be chosen from the menu bar. The steps to open a graph with the Matrix editor are given below:

1. Click the Matrix icon.
2. Select the graph, such as Use Case Diagram.
3. Select the representation from the dialogue that appears. To create a new representation, select ‘Create new Matrix’.

![Figure 4.25 – Matrix Editor](image)

The Table editor provides a tabular or form-based view on the objects in graphs, which are stored in the repository. The objects of the graph are represented as rows and the properties of the objects form the columns. Figure 4.26 shows a table that describes the ‘Actor’ and its properties. The steps to open a Table Editor are given below:

1. Click the Table icon.
2. Select the graph, such as the Use Case Diagram.
3. Select a representation from the dialog that appears. To create a new table, select ‘Create new Table’.
5.2 The Method Engineering Tools

The Method WorkBench of MetaEdit+ provides a set of tools that can be used to customize the methods supported by MetaEdit+. The tools are Object, Relationship, Role, Property and Graph Tool.

· Object tool
The Object tool as shown in Figure 4.27 is used to create, view and edit the object types in the system. Examples of object types are actor, activity or class. The creation of a new object must be for a particular project. Before the creation of new object, the Ancestor and Project need to be specified. For instance, to create a new object such as an actor, all the functions about it can be described using the tool. The new definition of actor will be updated and stored in the repository after the Generate button is clicked. The symbol for actor can be defined by using the Symbol Editor.
2. Property tool

All the objects, relationships, roles and graphs have their own properties that specify the information related to them. The Property tool is used to create and modify these properties. It can be opened by clicking the Property icon in the Launcher. Figure 4.28 shows the structure of Property tool. It contains Name, Ancestor, Description, Datatype and Widget fields. A new property can be created by filling the name field and specifies the data type. Clicking the Generate button will generate the new property. Figure 2.29 shows a pop-menu which lists the existing properties that can be used to define a new object.
3. Relationship tool

The *Relationship tool* also works in a similar way as the *Object tool*. It contains a few fields represented as buttons labelled *Name*, *Ancestor* and *Project*. As an example, to create an *aggregation relationship*, a name needs to be written in the *name* field and the properties related to the relationship such as *label* can be set. A relationship does not have any symbol. Figure 4.30 and Figure 4.31 show the structure of the *Relationship tool* and the pop-up menu that lists all the existing relationships defined in the UML.

![Figure 4.30 – Relationship Tool](image1)

![Figure 4.31 – The pop-up menu](image2)

4. Role Tool

The *Role tool* is used to specify the component attached at the end of a relationship which is connected to an object. The roles can have properties and a symbol such as an arrow. In a technique such as the Use Case Diagram, the roles for *extends relationship* can be defined as *extend to* and *extend from*. The *extend to* can have a symbol of hollow triangle. A line width and colour of the line can also be defined for the role. Figure 4.32 shows a display of the *Role tool*.
Figure 4.32 – Role Tool

The Graph tool provides the functionality to access, view and modify the method specification at the graph level such as the Use Case Diagram. The difference between the Graph tool and other tools is that the tools, such as the Object tool and Role tool, only handle individual components of a method. The Graph tool connects the pieces together in order to form the techniques.

In Figure 4.33, there are six buttons at the bottom of the window. The Generate button is used to generate the changes or the definitions of a new graph. The Types button functions in a similar way as the Type tool. It can be used to select the Object, Relationship and Role types of the method. The Bindings button is a binding tool, which is used to define the bindings of the Object, relationship and Role types within a Graph tool. Each element has a pop-up menu that allows the user to Add, Delete, Edit Type and Set Cardinality. A cardinality is a range with maximum and minimum values. In MetaEdit+, the cardinality is set from 0 to N (N is a maximum value). The Explosions and Decompositions buttons are the tools that provide a menu to Add or Delete the list of available graph types. This selection is made from the Object type to be exploded. A new graph such as an Activity Diagram can be created by filling the name field. The properties, types and bindings for the graph can be defined using the tool.
5.3 Symbol Editor

The Symbol Editor as shown in Figure 4.34 is used to draw a symbol for a particular object such as an actor or class. It also has a Symbol menu which allows the user to save the symbol and browse the library, to add the symbol from the library. Add to library, is to add the current symbol to the default project. The Edit menu enables the user to edit the symbol. As for the Connections menu, it allows the user to Generate defaults, to generate four default connections to form a rectangle around the symbol. This is important to ensure that any relationship occurs will attach directly to the object. The Add point is to add extra connection points and Join points, is used to temporarily connects the connection points.

A symbol such as an actor can be developed by clicking the circle and drag into the drawing board. A “stick man” can be formed by attaching two lines to the circle. A colour can be applied to make the symbol more interesting. As shown in the Figure 4.34, four small circles are attached to the symbol to define the connection point. To put a label to the symbol the properties of the object need to be defined.
5.4 Repository

All information in the MetaEdit+ environment is stored in the Object Repository. This includes all the methods, diagrams, matrices, objects, properties and others. The repository provides security such as locks and backup. A lock is used to prevent any changes by an unauthorized user and as well as to prevent two users from making any changes directly to the same method at the same time. A backup is used to make another copy of the repository. This is to prevent any destruction made to the original copy.

5.5 The Outcome

The outcome from the meta-modeling using MetaEdit+ environment is quite similar as in the MetaEdit Personal environment. A case study related to the Course Registration System has been done via UMLCASE. A complete case study can be referred in Appendix C. Since MetaEdit+ has come out with a set of engineering tools, most of the elements can be created easily without writing their method definition completely. From a completed diagram, its semi-completed source code in the form of C++, Smalltalk, Java, Object Pascal (Delphi) and SQL languages can be generated.
4.5.6 The Code and Report Generation

MetaEdit+ supports a few languages and as such, the code in C++, Smalltalk, Java and Delphi can be generated through UMLCASE. The code generation is performed by using the Report Browser. The steps are given below:

1. Choose a graph such as Class Diagram.
2. Click on Graph|Reports|Run from the pop-up menu.
3. Select the language from the dialogue that appear on the window.
4. Choose to generate the header file (*.h) or *.cpp files.

An example of the report generation of Use Case Diagram is defined in Figure 4.35. It is a report of object participation in the Use Case Diagram.

LIST OF OBJECTS AND THEIR CONNECTIONS (graph: c)

'Add and Drop courses' <Use case [UML]>
Takes part in relationship(s)
'Participation' with object(s), 'Student' <Actor [UML]>

'Student' <Actor [UML]>
Takes part in relationship(s)
'Participation' with object(s), 'Add and Drop courses' <Use case [UML]>
'Participation' with object(s), 'Register courses' <Use case [UML]>

'Setup Curriculum' <Use case [UML]>
Takes part in relationship(s)
'Participation' with object(s), 'Registrar' <Actor [UML]>
'Extends [UML]' with object(s), 'Log on' <Use case [UML]>
'with object(s), 'Log on' <Use case [UML]>

'Lecturer' <Actor [UML]>
Takes part in relationship(s)
'Participation' with object(s), 'Request course schedule' <Use case [UML]>

'Register courses' <Use case [UML]>
Takes part in relationship(s)
'Extends [UML]' with object(s), 'Pay study fee' <Use case [UML]>
'with object(s), 'Log on' <Use case [UML]>
'Participation' with object(s), 'Student' <Actor [UML]>

'Request course schedule' <Use case [UML]>

92
Takes part in relationship(s)
'Participation' with object(s), 'Lecturer' <Actor [UML]>

'Pay study fee' <Use case [UML]>
Takes part in relationship(s)
'Extends [UML]' with object(s), 'Register courses' <Use case [UML]>

'Log on' <Use case [UML]>
Takes part in relationship(s)
'Extends [UML]' with object(s), 'Setup Curriculum' <Use case [UML]>
" with object(s), 'Setup Curriculum' <Use case [UML]>
" with object(s), 'Register courses' <Use case [UML]>

'Registrar' <Actor [UML]>
Takes part in relationship(s)
'Participation' with object(s), 'Setup Curriculum' <Use case [UML]>

Figure 4.35 – Report Generation of the Use Case Diagram

C++ header files of the class diagrams can be generated according to the following rules:

- One header file generated for each class.
- For multiple inheritance, use visibility and virtually defined inheritance.
- Documentation of class is included as comments.
- Attributes and operations are ordered by their access level: public, private, protected.
- Comments are given for attributes with undefined data types, for filling in later.
- Undefined return types are treated as void.
- Constraints of attributes and operations are included as comments.
- Aggregation structures between classes are included as comments, for possible adding as attributes.

C++ function definitions of the class diagrams can be generated according to the following rules:

- One *.cpp file generated for each class.
- Include statements for header files.
- Undefined return types are treated as void.
• Each operation is defined by a specific name.
• Documentation of operation is included as comments.
• Operation body is included into the function declaration, properly indented.

The *.h file of the C++ code generation based on the Course Registration System is listed in Figure 4.36.

```cpp
// From MetaEdit+, graph: register
// Class definition (.h file)

class Course
{
    public:
        // Please specify data type name;

    private:

    protected:
};

class Schedule
{
    public:
        // Please specify data type date;
        // Please specify data type credit;

    private:

    protected:
        void update ( );
};

class Student
{
    public:

    private:

    protected:
};

class RegistrationForm
{
    public:
        // RegistrationManager myRegistrationManager; Please add here right access level and variable name!

    private:

    protected:
};
```
class Professor
{
    public:
        // Please specify data type name;

    private:

    protected:
};

class RegistrationManager
{
    public:
        void add_student(course, studentinfo) ( );

    private:

    protected:
};

Figure 4.36- The C++ Code Generation (*.h file)

Figure 4.37 provides a listing of *.cpp file from the C++ code generation.

//From MetaEdit+, graph: register
// function definition (*.cpp file)

void Schedule::update ( )
{
}

void RegistrationManager::add_student(course, studentinfo) ( )
{
}

Figure 4.37- The C++ Code Generation (*.cpp file)