Version 5.0

The Graphical Metamodeling Example

MetaCase Document No. GE-5.0

First Printing, 2nd Edition, September 2012.

MetaCase
Ylistönmäentie 31
FI–40500 Jyväskylä
Finland

Tel: +358 14 641 000
Fax: +358 420 648 606
E-mail: info@metacase.com
WWW: http://www.metacase.com

Preface

The goal of this example is to demonstrate graphical metamodeling in MetaEdit+. Graphical metamodeling is added to MetaEdit+ to support the early stages of language creation: designing the basic metamodel that covers the key language concepts and related rules. The created language design can be imported into MetaEdit+ Workbench and extended with the metamodeling tools of MetaEdit+ Workbench.

Technically, the metamodeling language is defined as one of the many domain-specific languages in MetaEdit+ Workbench. Now the domain is designing modeling languages and generating language definitions to be executed in MetaEdit+.

This example focuses on using the metamodeling language and generating the created language definition into an XML file. The generated XML will be imported back into MetaEdit+ Workbench as a modeling language. Please note that certain parts of the example require you to work “hands-on” to ensure the best understanding of the subject matter.

For exploring the metamodeling example thoroughly, you need to have MetaEdit+ installed on your machine. If you wish to extend the created language further − add notational symbols, additional constraints, generators or by modifying dialogs and toolbars for modeling tools − you should use MetaEdit+ Workbench or the evaluation version, which is available for download from www.metacase.com.

For further information about MetaEdit+, please refer to the ‘MetaEdit+ Users Guide’, ‘MetaEdit+ Workbench Users Guide’ or our web pages at http://www.metacase.com.

1 The graphical metamodeling example

The graphical metamodeling example presents a metamodeling language and its tool support specifically tailored for designing modeling languages. Strictly speaking, with the metamodeling language we focus on the basic metamodel, which covers a language's concepts, their properties, connections and rules as well as integration between several languages.

In this chapter we introduce the graphical metamodeling language and its usage scenarios. Chapter 2 then explains how to use the language via a small example in which we create a metamodel of a Use Case Diagram. Chapter 3 shows how the created language specification can be generated and imported into MetaEdit+ Workbench for inspection and further extension.

In this metamodeling example we will mostly discuss the use of the language for metamodeling. Because the metamodeling language is implemented similarly to other domain-specific modeling languages in MetaEdit+, it can be extended using the metamodeling tools of MetaEdit+ Workbench. This latter part, however, is not addressed in this example although modifications of this metamodeling language are possible.

Please note that walking through the metamodeling example requires a good knowledge of the metamodeling concepts of MetaEdit+ and basic knowledge on how to use MetaEdit+. A good starting point to get this knowledge is the Family Tree example in the 'Evaluation Tutorial'.

1.1 THE BASIC IDEA OF GRAPHICAL METAMODELING

A graphical metamodel usually covers the basic modeling concepts, their properties, connections and related rules. In terms of MetaEdit+, these concepts cover the following GOPRR metamodeling concepts: graphs, objects, properties, relationships and roles our modeling languages may have.

Graphical metamodeling is helpful at the early stages of language creation when we want to design the basic language structure and discuss about it with others. A graphical metamodel is also useful for getting an overview of the language.

During the later stages, language definers will find it becomes more effective to continue building the language via model examples rather than via the metamodel. This is simply because practical examples provide the ability to test the language and allow language users to understand it easier than when using the plain metamodel. Therefore, after having done the initial language design work graphically, it is normally better to continue by using MetaEdit+ Workbench's metamodeling tools to implement the complete language into MetaEdit+. This is achieved by importing the generated XML-based representation of the modeling language back into the MetaEdit+ Workbench. The MetaEdit+ Workbench is then used to extend the language further by defining notational symbols, additional constraints, generators or by modifying dialogs and toolbars for modeling tools.

We should note that graphical metamodeling is not particularly suitable for language evolution as it is separate from the models made so far. It doesn't allow so well to test language modifications with real models. Nor does it allow making changes into notations, generators,

dialogs or toolbar modifications. The metamodeling tools of MetaEdit+ Workbench, including the Symbol Editor, Generator Editor and Dialog Editor support the whole spectrum of language design and implementation.

1.2 AN EXAMPLE METAMODEL

Using the graphical metamodeling language, defining domain-specific languages is divided in three stages. You start by designing the basic concepts and rules with the graphical metamodeling language and then generate the metamodel into an XML file. Finally, you import this file into MetaEdit+ as a language definition and can apply it immediately. More about generating and importing XML type files in Chapter 3.

Figure 1-1 illustrates a sample language specification with the graphical metamodeling language. This diagram specifies the metamodel of a Data Flow Diagram. The Data Flow Diagram consists of three basic elements: 'External', 'Store' and 'Process'. These are specified using the Object concept in the metamodeling language. As these three objects have common properties and similar connections in data flow, an object type called 'Abstract' has been added to the language. This is a supertype of the other concepts. The 'Abstract' object is also marked as an abstract concept in the language by showing the text '{0}'.

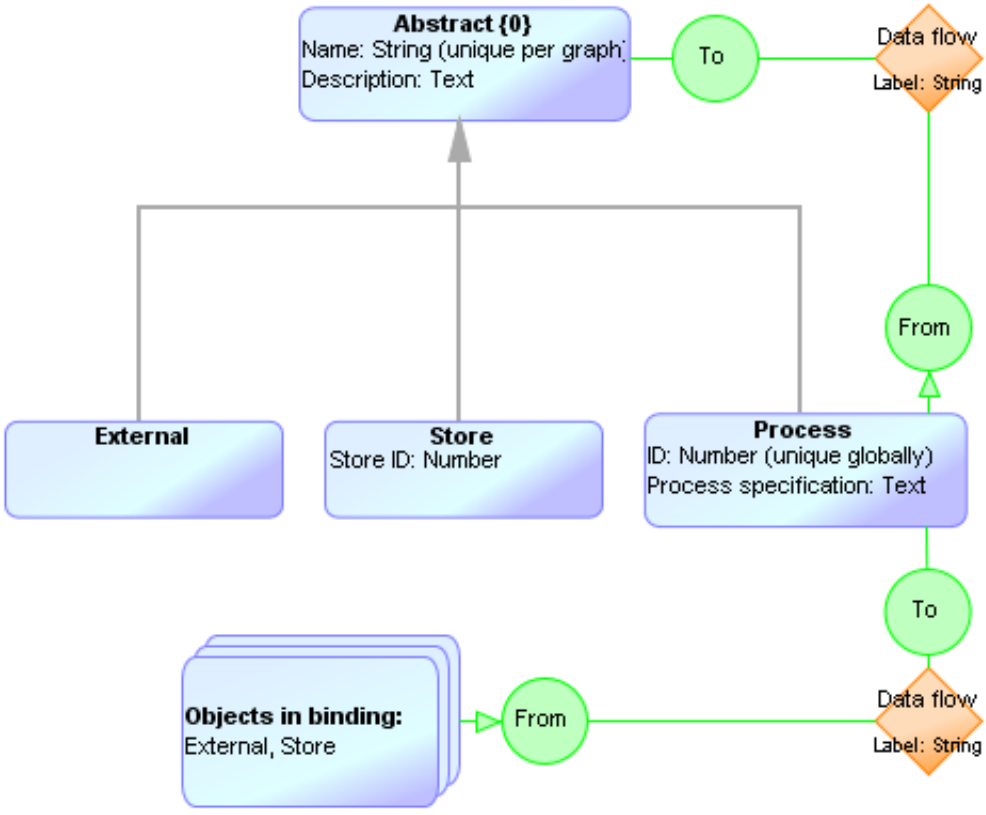


Figure 1-1. Metamodel of Data Flow Diagram

The metamodel in Figure 1-1 shows also properties of the modeling elements. For example, 'Abstract' has two properties: 'Name' which is a string that has a unique value within a data flow diagram and 'Description' which data type is text. The uniqueness rule for the 'Name' specifies that there can't be two instances of Store, External or Process in the same Data Flow Diagram with the same name.

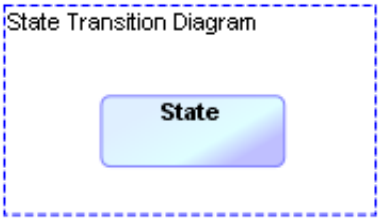
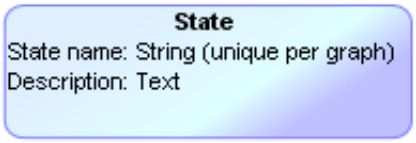
Other modeling elements have additional properties, like 'Process' has a property type 'ID'. This property is of number data type and its values must be unique among all data flow diagrams, as specified by the 'globally' keyword: it is not possible to have two different processes with the same ID in any of the diagrams.

If we inspect the metamodel further, we can see that there are two possible connection types between objects in the modeling language. These connections are called bindings in the metamodeling language (and in MetaEdit+ respectively). One binding is specified from 'Process' to any of the subtypes of 'Abstract'. The other binding can be drawn from 'External' or 'Store' to 'Process'. For the latter binding, the metamodeling language has a concept called 'Object set' to describe a collection of objects in a binding. This simplifies drawing the metamodel as there is no need to specify bindings for each object separately.

Finally, elements in bindings, namely relationships and roles, can also have properties. In this example, the relationship 'Data Flow', which is defined only once although presented twice in the diagram, has a string property type to enter a label for the flow. The metamodeling language includes a number of other rules but these are presented later.

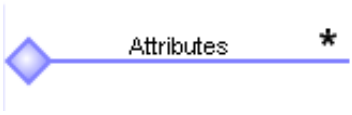
1.3 ABOUT THE GOPRR METAMODELING LANGUAGE

The graphical metamodeling language was made to work with MetaEdit+'s GOPRR datamodel. Therefore, all the modeling concepts used for graphical metamodeling come directly from MetaEdit+. These metamodeling concepts include:

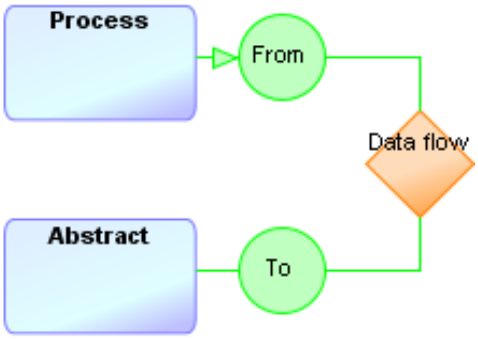
Language concepts	Representation of the concept
Graph specifies one modeling language, such as State Diagram and Use Case Diagram. Details of each language are modeled with a separate metamodel. Integration between languages, with explosions and decompositions, is specified in a metamodel for multiple graph types.	
Object describes the basic concepts of a modeling language. Objects are the main elements of your design. They are elements that you connect together and often reuse, such as Process, Message, Button and State.	
Relationship defines properties for the objects' connections, such as Inheritance, Message, Call and Transition. They are used to form bindings with objects and roles.	
Role specifies the lines and end-points of relationships, like the Superclass part of Inheritance relationships and the From part of State Transition.	
Property defines the attributes which characterize any of the previously mentioned language concepts. Properties can be of different data types	Properties are represented as part of other language concepts. Properties whose values are objects are shown like

(string, text, number, Boolean, collection etc.) and link to other modeling language concepts or to external sources, such as files, programs or webservices. Examples of properties are State name, Function identifier, Display type, and Data type.

this:



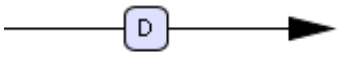
Binding connects a relationship, two or more roles, and for each role, one or more objects in a graph. Binding is further specified with multiplicity.



Object Set describes a collection of objects that can play the same role in a binding, for instance that External and Store can both be in the From role in a Data Flow relationship.

Inheritance allows creating subtypes of other language concepts, for instance External is a subtype of Abstract.

Decomposition allows objects to have subgraphs, for instance a Process can decompose to another Data Flow Diagram.



Explosion allows objects, relationships, or roles to be linked to other graphs, for instance the detailed structure of a Store in a Data Flow Diagram may be specified in an Entity Relationship Diagram.

To support generation from graphical metamodels back to MetaEdit+ the GOPRR metamodeling example includes a generator that creates MXT files. MXT file specifies metamodels in XML. The generated MXT file can be imported into MetaEdit+ and then applied as a modeling language. For further details of the MXT format see MetaEdit+ Workbench User's Guide.

2 Working with the metamodeling language

In this chapter, we discuss how to access the graphical metamodeling language and how to work with it, first by playing around with existing (meta)models and then by creating a new modeling language (metamodel) by using the graphical metamodeling language.

2.1 ACCESSING THE METAMODELING EXAMPLE

To access the metamodeling example, start MetaEdit+, choose the 'GOPRR' project from the list of available projects and login as usual into the demo repository. When MetaEdit+ has completed the login procedure, the metamodeling example can be accessed via the usual MetaEdit+ browsing and modeling tools like the Graph Browser and the Diagram Editor.

2.2 PLAYING AROUND WITH THE METAMODELING LANGUAGE

To start inspecting the metamodeling example, open any of the Graphs listed in the Graph Browser. A graph names 'Structured Analysis and Design' shows the integration among multiple languages. Double-click it from the list to open this integration metamodel. This metamodel is also presented in Figure 2-1.

The diagram provides an overview of multiple graph types, showing how four different modeling languages of 'Structured Analysis and Design' are integrated. In the diagram, each modeling language is defined by including it in a large rectangle that has a dashed line. This denotes to a graph type in MetaEdit+. The concepts of each language are then illustrated inside the graph type symbol. If a language element is not attached to be part of any language (be inside a graph type symbol), an error text is shown in the respective language element.

To access the properties of any model element, double-click it in the diagram or in the Diagram Editor sidebar. You may also access operations related to each model item by first choosing the element and then opening its pop-up menu.

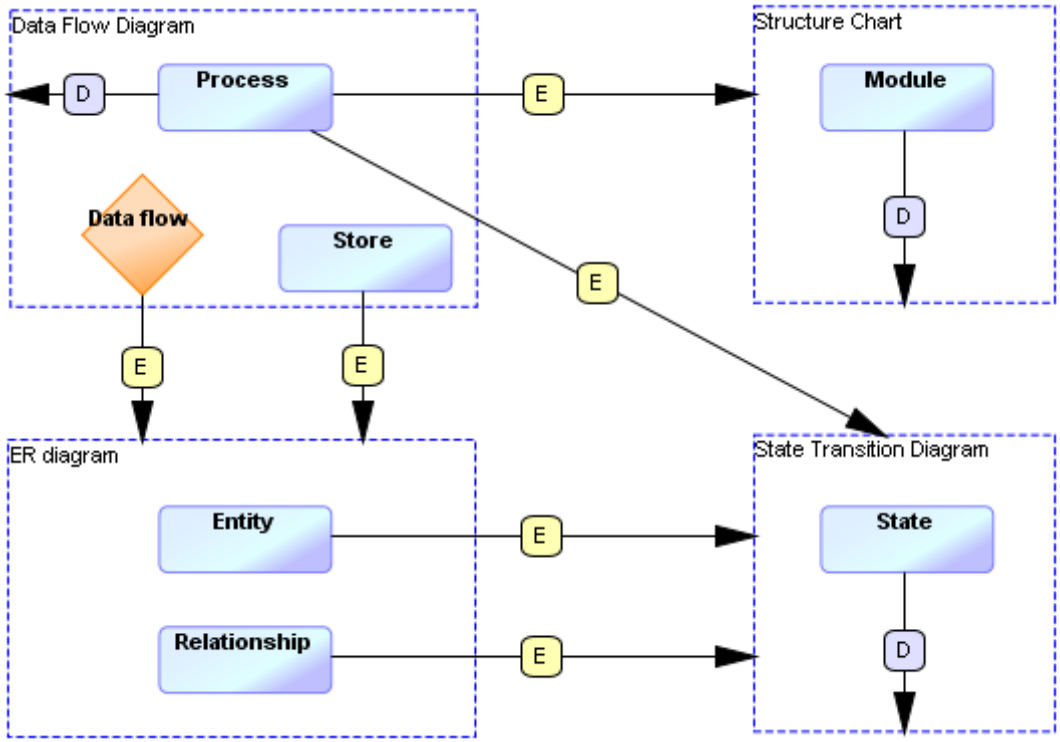


Figure 2-1. Metamodel for multiple graph types

Those language elements that have connections to other languages are linked with relationships. 'D' stands for decomposition and 'E' for explosion. According to the Structured Analysis and Design method, a 'Process' (Object) in a Data Flow Diagram (Graph) can be decomposed into another instance of Data Flow Diagram. 'Process' can also be exploded into one or more Structure Charts to specify its internal structure and to State Transition Diagrams to specify its behavior. In a similar manner, a schema of 'Store' identified in a Data Flow Diagram can be described using an Entity-Relationship (ER) Diagram. Other integration among the language concepts is illustrated similarly.

Details of each graph type are specified by using a different metamodeling language. To inspect these details, a separate diagram can be opened. To do so, you can simply double-click any of the graph types (symbols with dashed borders) while keeping CTRL pressed down. You may also choose a graph type in the diagram and select **Open Subgraph** from the pop-up menu to open the detailed metamodel. The pop-up selection **Manage Subgraphs…** allows modifying the subgraph links: removing or adding new ones.

To open the metamodel of the Data Flow Diagram, keep CTRL pressed and double-click the Data Flow Diagram symbol in the metamodel. This will open a new Diagram Editor to illustrate the same metamodel as presented in Figure 1-1. To inspect metamodels of other languages repeat this operation for other Graphs types.

2.3 CREATING A NEW METAMODEL

The next step in working with the graphical metamodeling language is to develop a new modeling language. In other words, we will use the metamodeling language to create a metamodel.

We will start here from scratch and make a graphical metamodel for the Use Case Diagram. We chose the Use Case Diagram because it is familiar to most and it has just a few concepts.

Because of this, we can create a metamodel of the language, generate the MXT file and import it back to MetaEdit+ to use the created Use Case language, all within half an hour.

2.3.1 Creating a new graph type

We start by first creating a new diagram for the Use Case metamodel. First, click the **Create Graph** button in the main window or choose the same operation from a pop-up menu that can be opened from the middle list of Graph Browser. You will be then asked for the type of metamodel you would like to create. As we define here only one language choose Metamodel [GOPRR] and press **OK**.

Next, enter the name for the graph ('Use Case Diagram') and add properties that you like to give for each diagram, like 'Model name' or 'Documentation'. To do this, open a pop-up menu in the Properties field and select **Add Element…**.This opens a new dialog for entering values for each property (see Figure 2-2). You can now enter the details of a property type, like its mandatory name and optional local name to be used in the modeling tool. You can also specify a data type for each property by choosing from a list of possible data types. These data types are described in detail in MetaEdit+ Workbench User's Guide. For our case of Use Case Diagram we can choose String data type for 'Model name' and Text data type for 'Documentation'.

The dialog allows also entering default values for the property and choosing uniqueness constraints. For 'Model name' we should choose uniqueness constraint 'globally' as it does not make sense to have multiple use case diagrams with the same name.

Finally, we may enter a description for each metamodel element. The description entered here is used in the created language and can be accessed during modeling in MetaEdit+ from the Help menu. The property dialog for the Model name should now look like Figure 2-2. Choose **OK** and close the dialog.

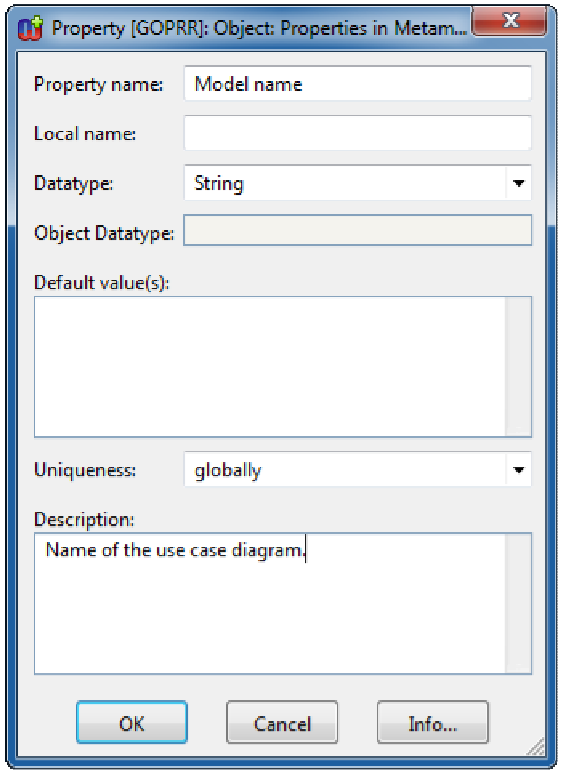


Figure 2-2. Dialog for adding a 'Model name' property for Use Case Diagram metamodel

Enter property type ‘Documentation’ in the similar manner to ‘Model name’ property and then close the dialog of Use Case diagram. This opens an empty Diagram Editor.

2.3.2 Adding a new object to the metamodel

Next, we need to specify the objects that we will use in our modeling language. In the Use Case Diagram they are ‘Use Case’, ’System’ and ’Actor’. Let’s start with the Actor concept. Choose Object [GOPRR] button from the toolbar or from Types menu and then click in the diagram. This opens a dialog to specify details of the object.

First we must give a name for the object: Enter ‘Actor’. Then we can specify the property types each actor may have. The property types such as ‘Actor name’ can be specified in a similar way as we did already for the properties types of graph type Use Case Diagram.

While defining the Actor concept we can also reuse already defined property types. For example, ‘Documentation’ text property type was already defined for Use Case Diagram. We can use it as a property type for the Actor too. To reuse it choose **Add Existing…** instead of previously used **Add Element…** menu item. This opens a dialog showing all available property types. Double-click the ‘Documentation’ item to add it to the list of selected elements to be added. Then press **OK** button to add the selected property type into the properties of ‘Actor’.

To finalize the definition of the ‘Actor’ we can enter a description to the modeling concept and choose the occurrence constraint. The default value ‘N’ means that multiple actors may exist in one use case diagram. Choosing the value to be ‘0’ will signify that the object is abstract, similar to the “Abstract” Object in our Data Flow Diagram example in Figure 1-1.

After you have defined the two properties, the dialog for specifying ‘Actor’ should look like Figure 2-3. Choose **OK** and close the dialog. This will add the created object to the diagram.

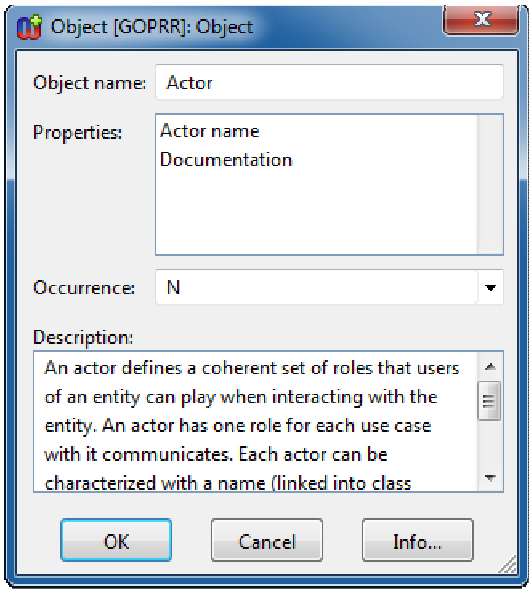


Figure 2-3. The specification of ‘Actor’ object

Next, we can continue by entering other modeling objects for the Use Case Diagram. Add the ‘System’ and ‘Use case’ objects in a similar manner. For ‘System’ we can define occurrence to be ‘1’, meaning that each use case diagram can show just one system. If you like to specify multiple systems within the same use case diagram, change this value to ‘N’.

To allow linking external files to use cases in the model remember to choose External Element as a data type for 'Documentation file' property type. After adding these two additional object types, the use case metamodel should look like Figure 2-4.

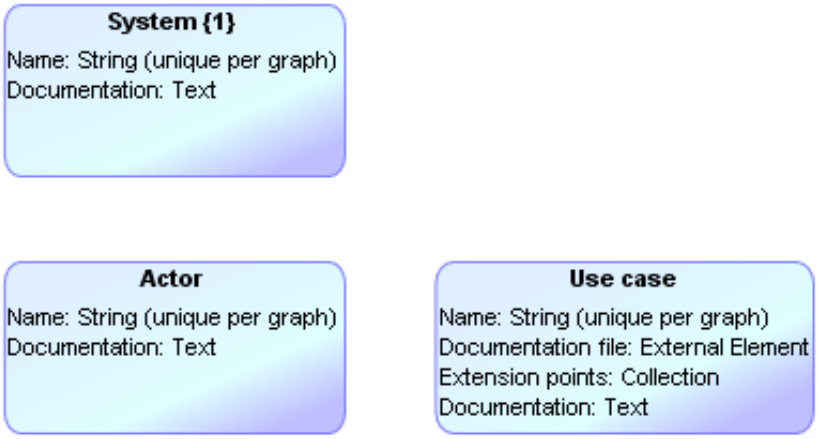


Figure 2-4. Modeling objects added to the metamodel of Use Case Diagram

2.3.3 Creating bindings between objects

Next, we need to define connections between the object types. We do this by creating bindings among the objects. To define an association relationship between use case objects and actor objects, we define a binding between them. Choose **Binding [GOPRR]** button from the toolbar (orange diamond) and connect the two objects by clicking them. Alternatively, you can start by choosing **Connect…** from the pop-up menu of an object and then click the other object or use any of the other connection creation possibilities as described in Diagram Editor Chapter of MetaEdit+ User's Guide.

Creating a binding relationship will open a dialog to specify details of the binding. Binding is the same concept as in the GOPRR datamodel of MetaEdit+. It relates objects with each other by defining their relationships and the role each object plays in that relationship. In the binding tab of the dialog we can specify the relationship type, its name and properties. This is done similarly to that of specifying objects. Choose **Attach New Object…** and enter at least a name for the relationship, like 'Association'. You need to specify at least the relationship name as it is a mandatory property. Optionally, you may also provide it with more properties, like for example 'Association name'.

For the binding we need to specify a minimum of two role types. We specify these in the next tabs of the binding dialog, as shown in Figure 2-5. In case of the Use Case Diagram, the 'Association role' can be specified similarly to 'Association' relationship. Click the tab with text 'First role' and specify 'Association role' that can have further properties like 'Role name'. Note that the name of the role is a mandatory property. For role types we need to specify also their cardinality constraints. The default values work here well as there normally can be only one use case and actor in the same association. Later we show other cardinality values for other bindings.

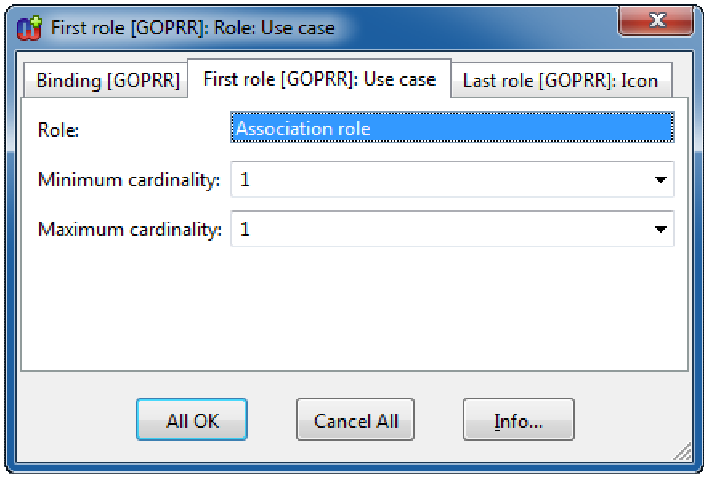


Figure 2-5. Creating binding between 'Actor' and 'Use case'

While specifying the second role type we can reuse the already once defined 'Association role' by selecting **Attach Existing Object…** from the pop-up menu and choosing the already defined 'Association role' from the list of available role types. Default values for cardinalities work also well here.

Finally, for the second role we can choose the direction in which the 'Actor' and 'Use case' can be connected. By default, the bindings are created as directed ones but if we choose the option **Can be drawn in both directions** then connections can be created in both ways: From actors to use cases and from use cases to actors. Choose this option and press **OK.** This creates a binding into the diagram.

In a similar manner you can also define a generalization relationship and a dependency relationship for the use cases. As described in Figure 2-6 the binding for 'Generalization' has 'Superclass' and 'Subclass' role types. The cardinality constraint for 'Subclass' role is '1,N' allowing to specify multiple subclasses with the same generalization relationship.

In Use Case Diagram, a dependency relationship allows specifying uses and extends connections between the use cases. To define this into our metamodel we add a 'Stereotype' property for the 'Dependency' relationship. This property type has a predefined list of values, namely 'use' and 'extends'. If we choose an overridable list as a data type for the 'Stereotype' property then the use case modeler can choose among the predefined values but can also enter own values. The first value entered in the list of predefined values is used as a default value. First value can be also an empty line. It allows defining dependency relationships into use case models which don't have any stereotype value.

To finalize our bindings for Use Case Diagram, we can also add 'Note' object to the language and add it to the association binding. This allows relating additional note elements in use case diagrams to specific associations. To add the created 'Note' object to the existing binding, select the 'Association' relationship. Then choose **Add a New Role…** from its popup menu and click the 'Note' object to create the connection. For the new role we can create a role called 'Note part'. This role definition should have '0' value as a minimum cardinality. This makes adding notes to associations optional.

We can repeat the same metamodeling operations and add optional role also for other bindings if we need them in use case modeling. The metamodel should now look similar to Figure 2-6.

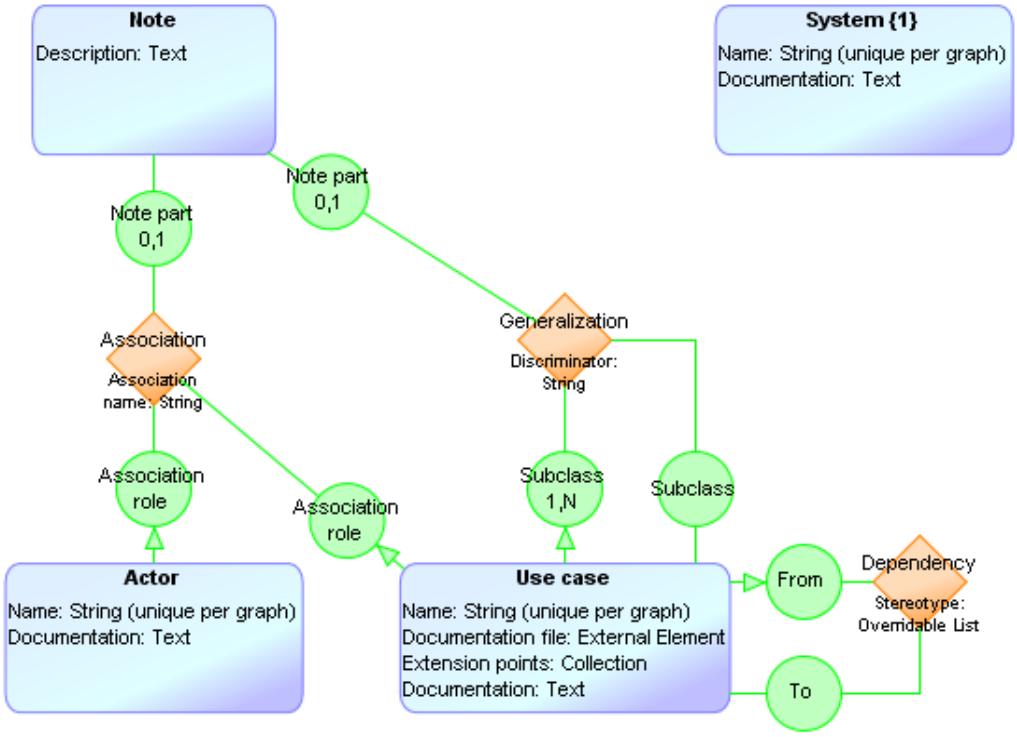


Figure 2-6. The graphical metamodel for Use Case Diagram

2.3.4 Adding objects as properties

To make the metamodel complete, we need to define the Attributes and Operations that a Use Case may have. This is especially needed if we want to specify a Use Case as a Class. For this purpose we can create two new Objects called ‘Attribute’ and ‘Operation’ and then connect them to the ‘Use case’ object with the property relationship. You can choose this connection from the ‘Property’ button of Diagram Editor toolbar (blue diamond with a short line).

For the property connection we can specify an optional local name and constraints. As a use case may have multiple attributes and operations we must mark them as Collections in the NonProperty of tab during relationship creation. This collection value is shown in the diagram with asterisk (*).

You can add additional properties for the ‘Attribute’ and ‘Operation’ objects if you want. Remember to mark these objects as abstract (choose Occurrence value ‘0’) so they are not used as the main modeling concepts but are available only via the ‘Use Case’. You can also make more complex modeling concepts by adding further objects as properties, like in case of Parameters for Operations.

Figure 2-7 illustrates the final metamodel for the Use Case Diagram. You will find it is similar to other metamodels available in the GOPRR project.

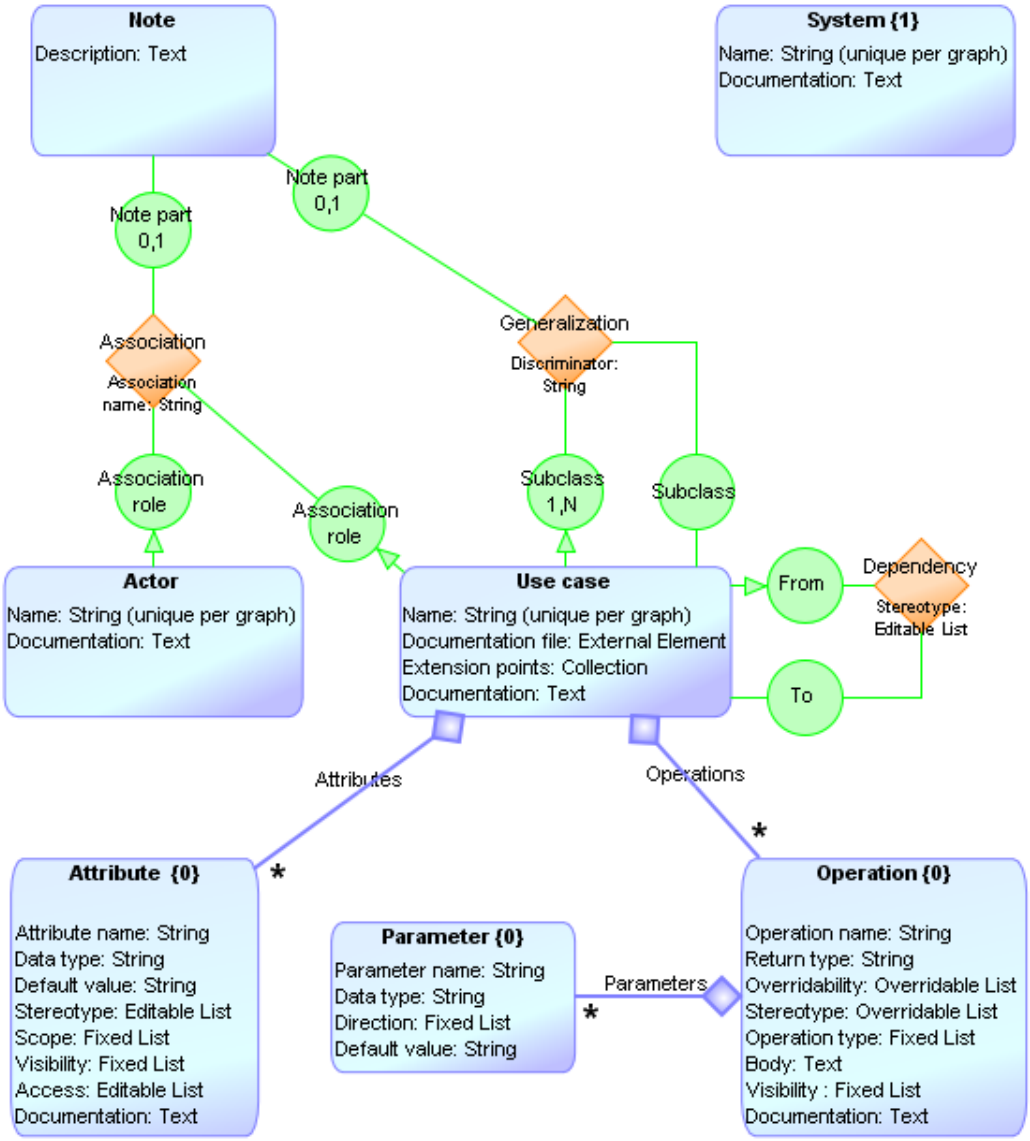


Figure 2-7. The complete metamodel for Use Case Diagram

2.3.5 Integrating languages and integrating individual graphs

Frequently, we need to integrate individual graphs with one another or integrate several languages together. A graphical metamodeling language allows for the creation of such explosion and decomposition links supported by GOPRRR. Figure 2-1 illustrates an example of this type of integration between four different languages.

To finalize our language design for Use Case Diagram we specify a language structure in which each System object can be described in one subgraph. This is specified as a decomposition. While this subgraph, target of decomposition, could be based on another language—as in Figure 2-1—we will create the link to the existing Use Case Diagram as follows:

First, click the **Create Graph** button in the main window and choose Metamodel for multiple graphs [GOPRR] and press **OK**. Next, enter a name for the graph (e.g. 'MyLanguage') and press **OK**. This opens an empty Diagram Editor providing a slightly different language than

the one used to specify the metamodel of Use Case Diagram. Here you can add all the languages to be integrated by selecting Graph [GOPRR] from the toolbar and then clicking in the diagram. This adds a Graph symbol to the drawing area with instructions on how to integrate it with the existing language as described below.

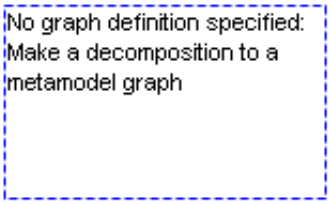


Figure 2-8. Graph added to the metamodel.

Next, this can be linked to the existing metamodel—Use Case Diagram in our case—by choosing **Open Subgraph** from the pop-up menu and then choosing Use Case Diagram from the list of available individual metamodels. Alternatively, you could start creating a new metamodel. After creating the link to a subgraph, the name for the metamodel will become visible for the Graph symbol. The pop-up selection also allows for the modification of the subgraph link: either by removing or adding links to other metamodels.

To specify a decomposition from System to Use Case Diagram we need to add the System concept to the diagram and place it inside the dotted graph symbol (see Figure 2-9). This denotes that the System concept is part of the Use Case Diagram. To do this you can recreate the System concept or simply add the existing System object that was previously defined. To do this you simply copy and paste the object from the subgraph into the top graph. The advantage this approach gives is that if you change the name of the System in one place it is changed everywhere immediately as you are reusing now the same object.

A slightly more powerful, but complex, way is to click the Object button in the type toolbar in the top graph and then shift-click in the desired place, which will prompt you in a dialog to select an existing Object type. (**Add Existing...** from the pop-up menu in the drawing area with no selection does the same.) The dialog list will initially show the objects already used in the graph, so you can press the Graph button to show all graphs, then choose the appropriate graph and double-click the right object type within it. This allows you to reuse several objects from various places at the same time. The path followed to get to the right object type will be remembered, so next time it can be selected from the Selection History pull-down list in that dialog.

If a language element (e.g. System in Figure 2-9) is not attached as part of the Use Case Diagram (i.e. inside a graph type symbol), an error text is shown in the Symbol element.

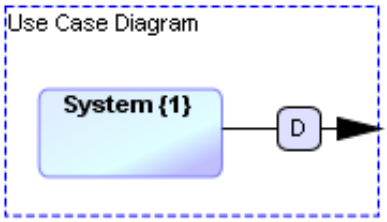


Figure 2-9. System is decomposed into another use case diagram.

To finalize the language structure, create a decomposition relationship from System object to Use Case Diagram: choose Decomposition [GOPRR] button from the toolbar and then connect System to Use Case Diagram as shown in the Figure 2-9. This structure now allows for the creation and maintenance of a hierarchy of use case diagrams.

3 Generating language definition into MetaEdit+

Now that we have defined our language, its concepts, properties, connections and rules, next we will generate our metamodel into MXT file format, so that we can import it into MetaEdit+ Workbench for further completing.

3.1 GENERATING MXT

MetaEdit+ supports XML-based importing and exporting of metamodels. The XML format for metamodels is called MXT (MetaEdit+ XML Types file). This metamodeling example uses the same format for producing XML files from the graphical metamodels.

To generate a MXT file from the graphical metamodel of Use Case Diagram we run the generator. In the Diagram Editor, generators can be executed by selecting **Graph | Generate…** and then choosing the required generator from the list that opens. A faster way is to press the generator button that is available in the toolbar of Diagram Editor.

For generating the metamodel into MXT, we provide three alternative generators:

- 'Export graph type to MXT file' (also the 'MXT' button in the toolbar of the Diagram Editor) produces the MXT file into the output directory. By default it is called 'reports' and its location is a subdirectory of the MetaEdit+ directory.
- 'Export and Open MXT' produces the MXT file and opens it in your default browser.
- 'Export and Build MXT' (also 'Build' button in the toolbar of Diagram Editor) produces the metamodel into MXT file and imports it as a metamodel into MetaEdit+. Please note that this requires that you have rights to import XML files into MetaEdit+. Read MetaEdit+ User's Guide for importing files.

The same generators are also available for the metamodeling language describing language integration. If you execute the generator from a metamodel that describes multiple languages then all the languages will be included into the same MXT file.

Next, you can run a generator to produce XML for the Use Case Diagram that we specified earlier. If you run the Build generator, no further actions are needed to import it to MetaEdit+ but if you executed other generators you will need to import them manually. To do this, press the Import button in the MetaEdit+ main window and choose the MXT file to be imported. MetaEdit+ User's Guide describes the procedures for importing files in more detail.

3.2 WORKING AND EXTENDING THE IMPORTED METAMODEL

After importing the metamodel, you can access it by using the metamodeling tools of MetaEdit+ Workbench, as described MetaEdit+ Workbench User's Guide. We can therefore next complete our use case implementation and add notation to it by using the Symbol Editor,

make possible generators for documentation and checking reports with the Generator Editor, modify dialogs in the Dialog Editor or customize toolbars used in Editors.

If we need to change the basic metamodel after importing it, we can either modify the metamodel directly using the metamodeling tools of MetaEdit+ or use the graphical metamodeling language. For the latter case we need to regenerate the MXT file and import it back into MetaEdit+.

3.3 EXTENDING THE MXT GENERATORS

The generators that we used to produce the MXT files are made in a similar way as other generators of MetaEdit+. You are free to modify them or create new generators to export the metamodels in other formats. To access these generators, open the Generator Editor by choosing **Edit Generators** from the **Graph** menu. Note that there are actually two main generators, one is for generating MXT for one graph and the other is for generating MXT from multiple graph types. To access these different generators you will need to open the generator for both metamodeling languages. For defining generators, please read the MetaEdit+ Workbench User's Guide for details.

4 Conclusion

In this example we have demonstrated graphical metamodeling. With the metamodeling language you can design the basic structure for a domain-specific language as well as its integration with other languages.

The created graphical metamodel can be generated into MetaEdit+ as a modeling language. MetaEdit+ provides then modeling tool support for it with various editors, browsers, multi-user support etc. The importing of metamodel is based on using the MXT format (MetaEdit+ XML Types). After importing the metamodel the language can be further extended by adding notation, generators and constraints or modifying dialogs and toolbars related to the language use. To complete the language definition you need MetaEdit+ Workbench.

The metamodeling language is tightly related to GOPRR metamodel used in MetaEdit+. However, it is implemented as any other modeling language in MetaEdit+. It is completely open and thus it can be freely extended to cover additional requirements of graphical metamodeling, such as cover ports or other metamodeling needs you find relevant. You are welcome to extend the metamodeling language as well as the generators further.