MapleMBSE 2021.1
Configuration Guide

MapleMBSE 2021.1 Configuration Guide

Contents

List of Figures

Introduction

MapleMBSE Configuration Guide Overview

MapleMBSE™ gives an intuitive, spreadsheet based user interface for entering detailed system design definitions, which include structures, behaviors, requirements, and parametric constraints.

The configuration file specifies the rules for how your data from your model is extracted and mapped into a table format along with how and where the extracted data is presented in an Excel spreadsheet.

In the following chapters, this guide will provide detailed instructions on working with configuration files and the configuration file language.

Related Products

MapleMBSE 2021 requires the following products.

- Microsoft® Excel® 2010 Service Pack 2, Excel 2016 or Excel 2019
- Oracle® Java® SE Runtime Environment 8.

Note: MapleMBSE looks for a Java Runtime Environment in the following order:

1) If you use the -vm option specified in **OSGiBridge.init** (not specified by default)

2) If your environment has a system JRE (meaning either: JREs specifed by the environment variables JRE_HOME and JAVA_HOME in this order, or a JRE specified by the Windows Registry (created by JRE installer)), MapleMBSE will use it.

3) The JRE installed in the MapleMBSE installation directory.

If you are using IBM® Rational® Rhapsody® with MapleMBSE, the following versions are supported: Rational Rhapsody Version 8.1.5, 8.3 and 8.4

- Teamwork Cloud™ server 18.5 SP3 or 19.0 SP4

If you are using Eclipse Capella™ with MapleMBSE, the following version is supported:

- 1.4.0

If you are using Eclipse™, the following version is supported:

- 2020-3

Note that the architecture of the supported non-server products (that is, 32-bit or 64-bit) must match the architecture of your MapleMBSE architecture.

Related Resources

Resource	Description
MapleMBSE Installation Guide	System requirements and installation instructions for MapleMBSE. The **MapleMBSE Installation Guide** is available in the **Install.html** file located either on your MapleMBSE installation DVD or the folder where you installed MapleMBSE.
MapleMBSE User Guide	Instructions for using MapleMBSE software. The **MapleMBSE User Guide** is available in the folder where you installed MapleMBSE.
MapleMBSE Applications	Applications in this directory provide a hands on demonstration of how to edit and construct models using MapleMBSE. They, along with an accompanying guide, are located in the Application subdirectory of your MapleMBSE installation.
Frequently Asked Questions	You can find MapleMBSE FAQs here: https://faq.maplesoft.com
Release Notes	The release notes contain information about new features, known issues and release history from previous versions. You can find the release notes in your MapleMBSE installation directory.

For additional resources, visit **http://www.maplesoft.com/site_resources**.

Getting Help

To request customer support or technical support, visit **http://www.maplesoft.com/support**.

Customer Feedback

Maplesoft welcomes your feedback. For comments related to the MapleMBSE product documentation, contact doc@maplesoft.com.

Copyrights

1 Getting Started

1.1 Introduction

The goal of this section is to introduce the elements of the configuration and template files and how they are connected together by defining a simple configuration file. The details about the elements are given in the following chapters.

A configuration file defines what data from a model is accessible and how it is presented in Excel. In order to do that, the configuration file must define the following elements.

- The content of the Excel workbook: how many and what types of worksheets it has.
- For each worksheet, define the area that is associated with the model data - the SyncView area and how it is displayed.
- For each SyncView area define what model data is displayed using a SyncTable.

1.2 Overview of MapleMBSE Mapping

The primary purpose of MapleMBSE is to map diagram-based models in UML into a table form that can be easily consumed and updated by an end user.

Mapping model information from diagram-based model form into table form requires a two step process.

First, a **SyncTable Schema** must be defined to convert the model to an intermediate table structure called a **SyncTable**.

A SyncTable Schema specifies how to find objects in a model starting with an object given by a **DataSource**. A pair of a DataSource and a SyncTable Schema defines one SyncTable.

Next, the **SyncView Layout** must be defined for how the SyncTable is displayed on a spreadsheet by specifying a layout and which columns of the SyncTable to include or omit. The resulting part of the spreadsheet displaying the SyncTable is called **SyncView**. The schematic flow of displaying a model in an Excel spreadsheet is shown in *Figure 1.1 (page 2)*.

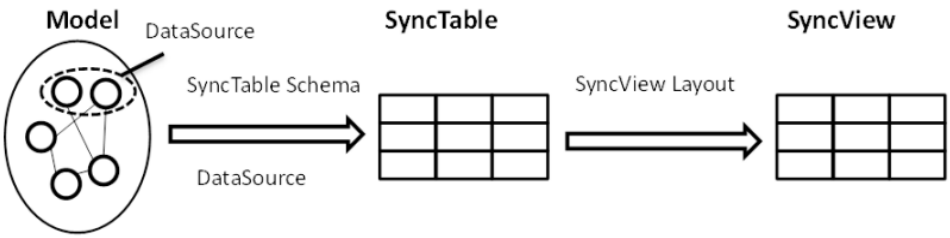


Figure 1.1: Schematic Diagram of How MapleMBSE Works

The definition of a DataSource, a SyncTable Schema, and a SyncView Layout is called an **MSE configuration**. The language used to define an MSE configuration is called MSE configuration language. In this guide we provide the specification of the MSE configuration language. For notation used in the specification, see *Notation (page 19)*. MSE configuration files are text files that can be edited and created with any text editor. However, it is recommended to use MSE Configuration Editor which provides convenient syntax highlighting and checking. For the installation instructions, see *MSE Configuration Editor (page 2)*. Examples in this guide use the MSE Configuration Editor.

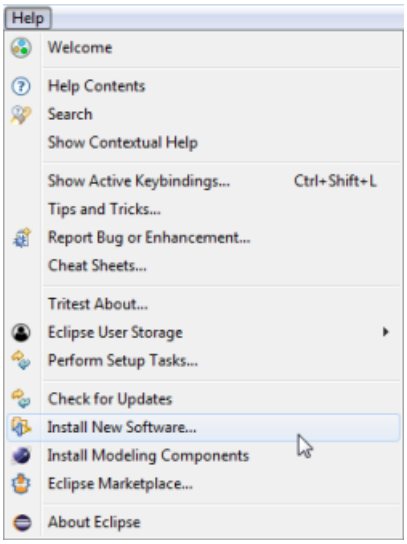
1.3 MSE Configuration Editor

The MapleMBSE Configuration Editor (a.k.a MSE Editor) is provided in the same package as MapleMBSE-Editor_<VERSION>.zip and can be downloaded from:

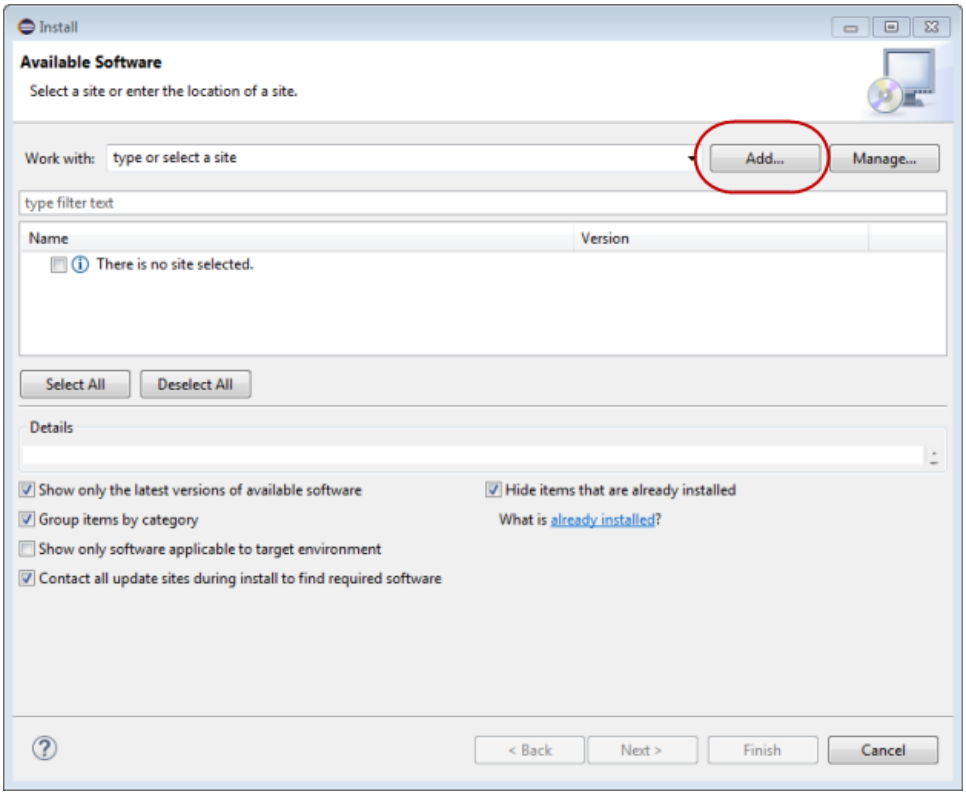
https://www.maplesoft.com/support/downloads/index.aspx#mbse.

The MSE Editor is an Eclipse add-on, and you can install with the following steps:

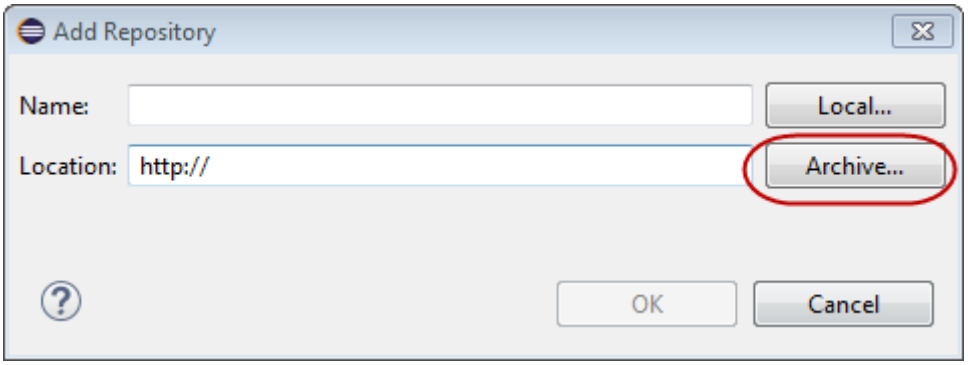
1. Launch Eclipse (Oxygen).
2. Select **Help**, then **Install New Software**.



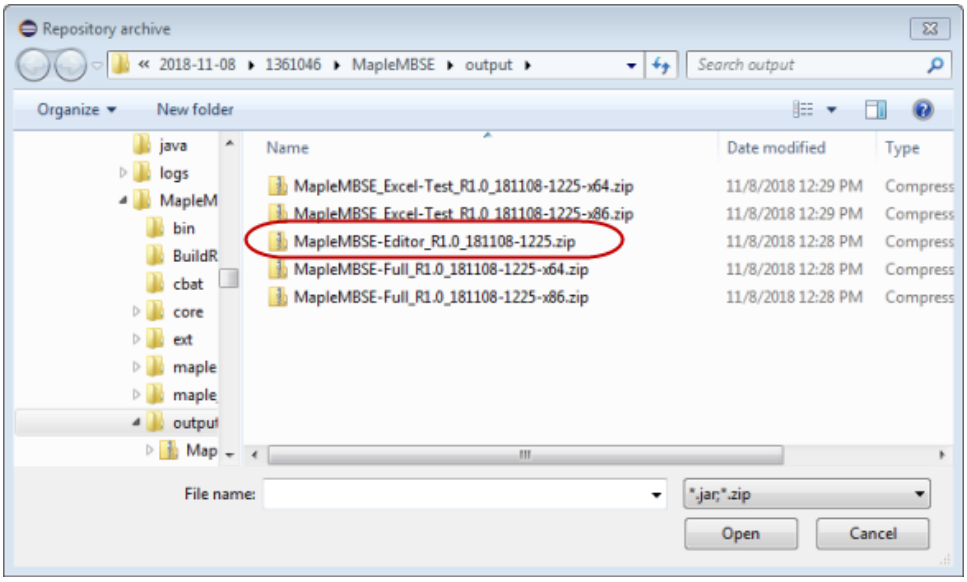
3. Click **Add** to display the Add Repository window.



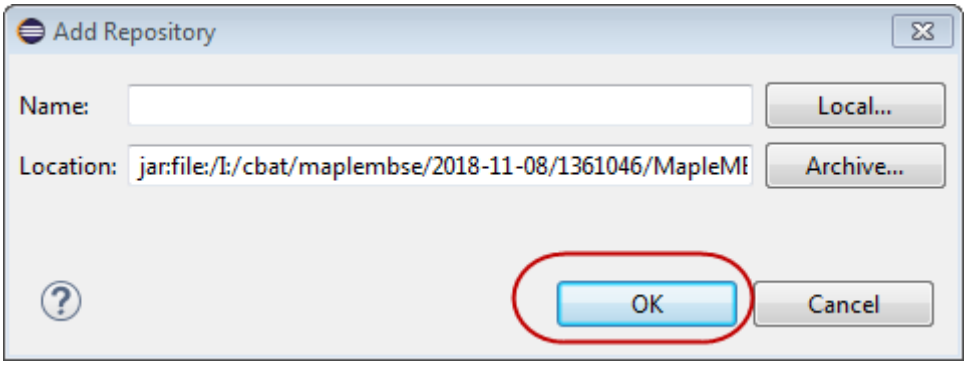
4. In the Add Repository window click **Archive**.



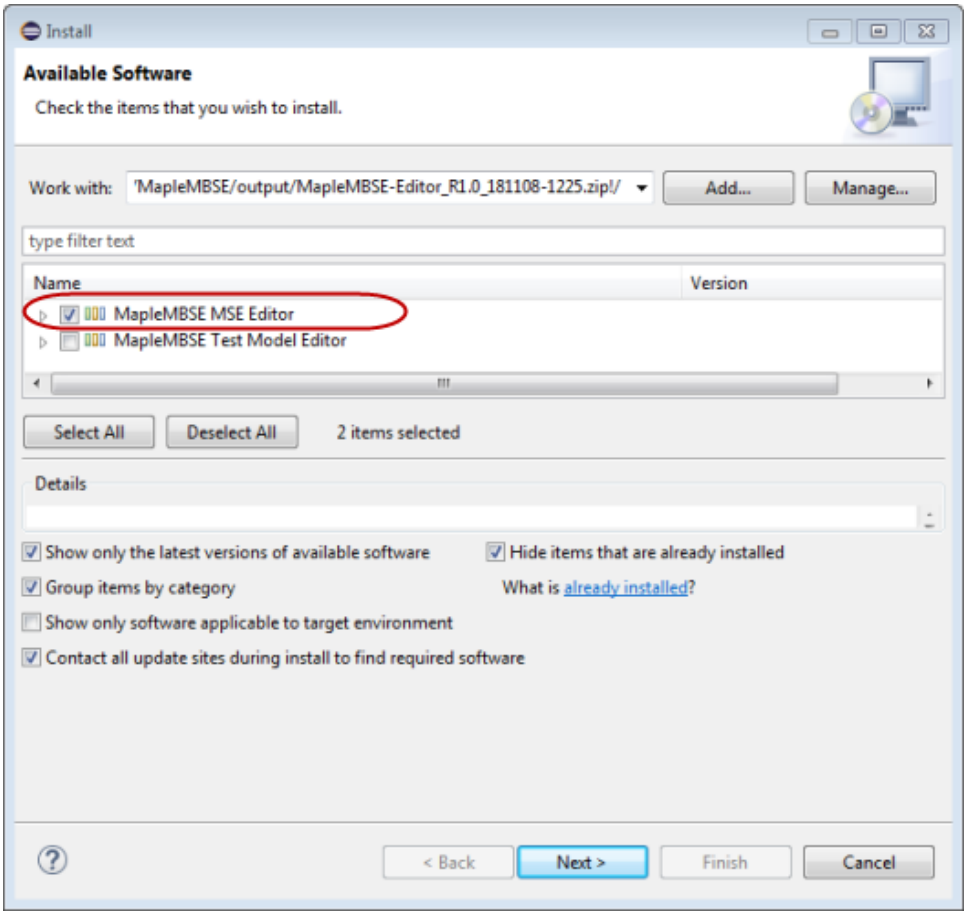
5. Find and select the MapleMBSE-Editor_2021.1.zip file.



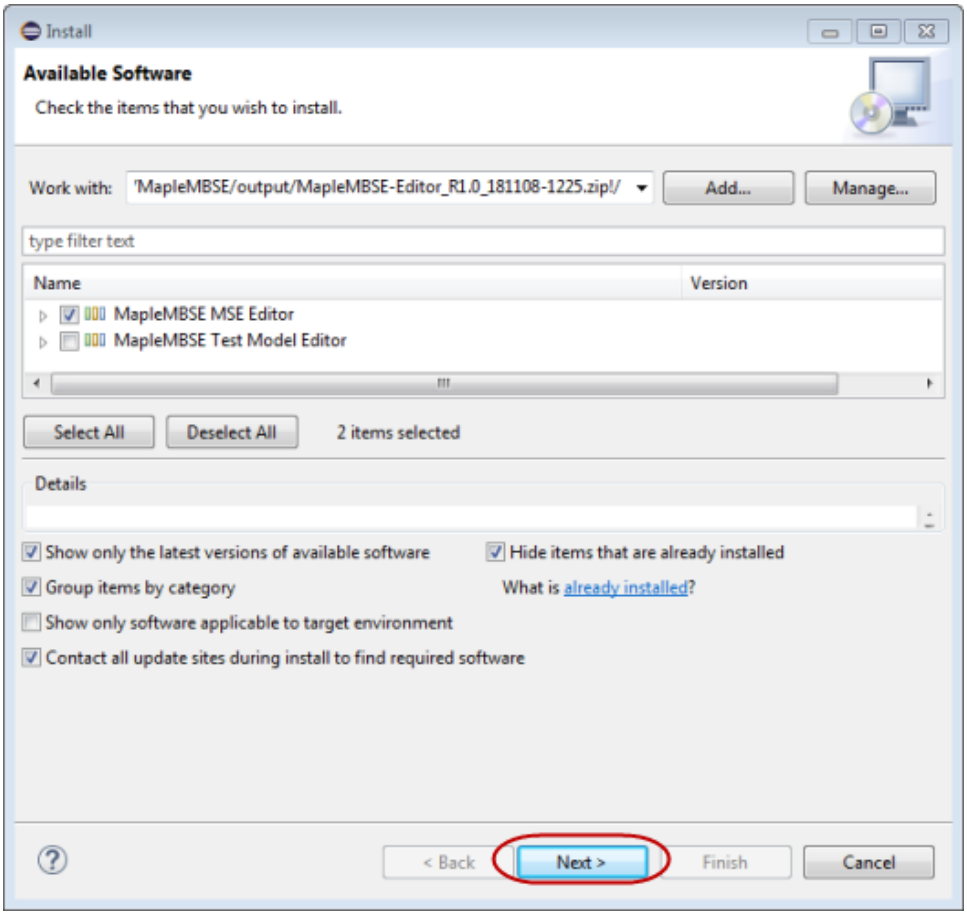
6. Click **Open**.
7. Click **OK**.



8. From the list of files, select MapleMBSE MSE Editor and then follow the instructions shown in the dialog.


Install
Available Software
Check the items that you wish to install.
Work with:
'MapleMBSE/output/MapleMBSE-Editor_R1.0_181108-1225.zip!/
Add...
Manage...
type filter text
Name
Version
MapleMBSE MSE Editor
MapleMBSE Test Model Editor
Select All
Deselect All
2 items selected
Details
Show only the latest versions of available software
Hide items that are already installed
Group items by category
What is already installed?
Show only software applicable to target environment
Contact all update sites during install to find required software
< Back
Next >
Finish
Cancel

9. Click **Next**.



10. Click **Next**.

11. To proceed, accept the terms of the license agreements.

12. In the security warning dialog click **Install Anyway**.

13. Click **Finish**.

14. Restart Eclipse.

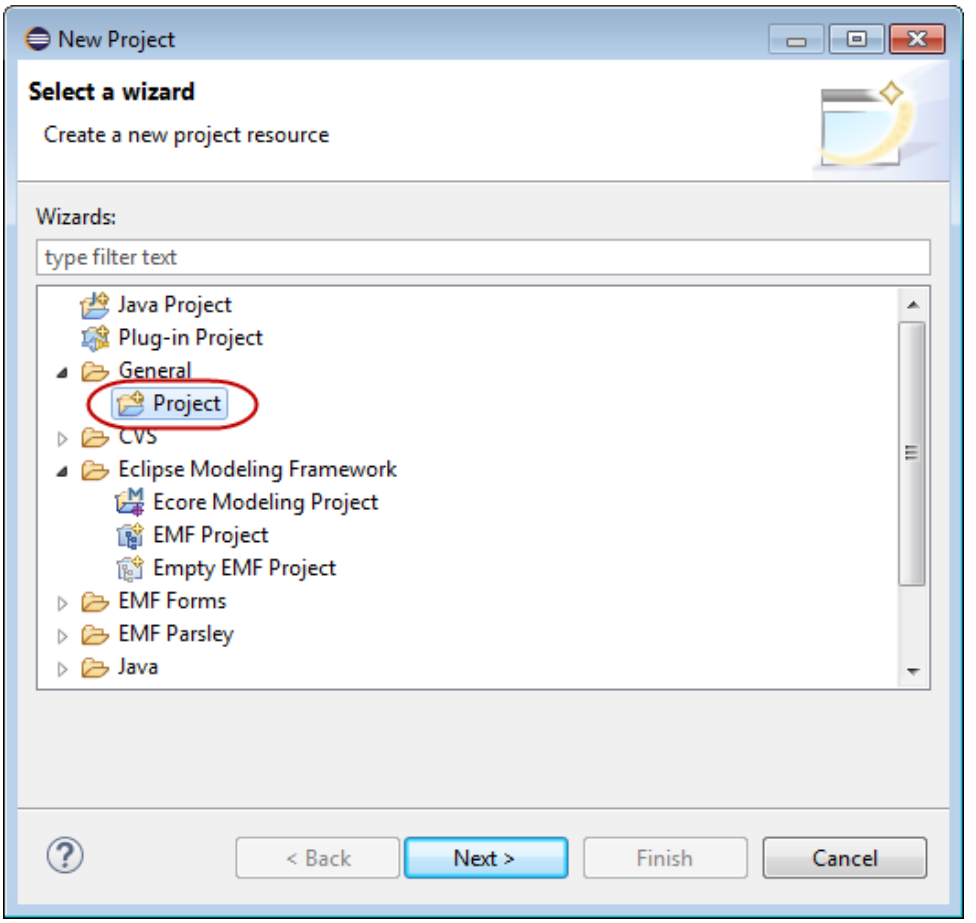
1.4 Creating a Configuration File

To use the editor, you first need to creat a project folder for your configuration file(s) in your Eclipse workspace. Then, add an MSE file to your project file.

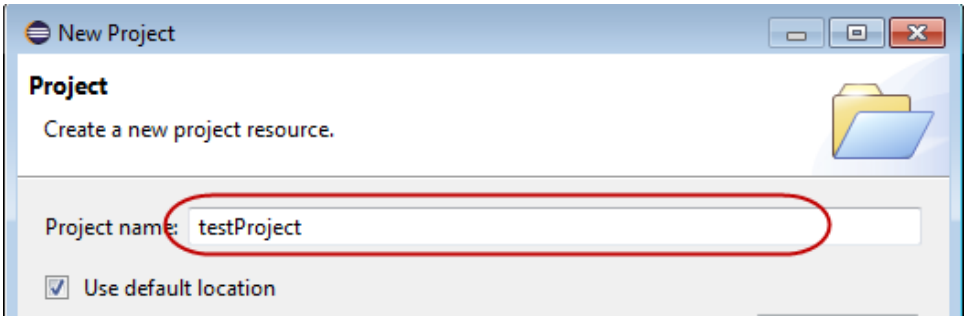
Note: Double-clicking an MSE file in the workspace launches the editor.

To create the project folder and MSE file do the following:

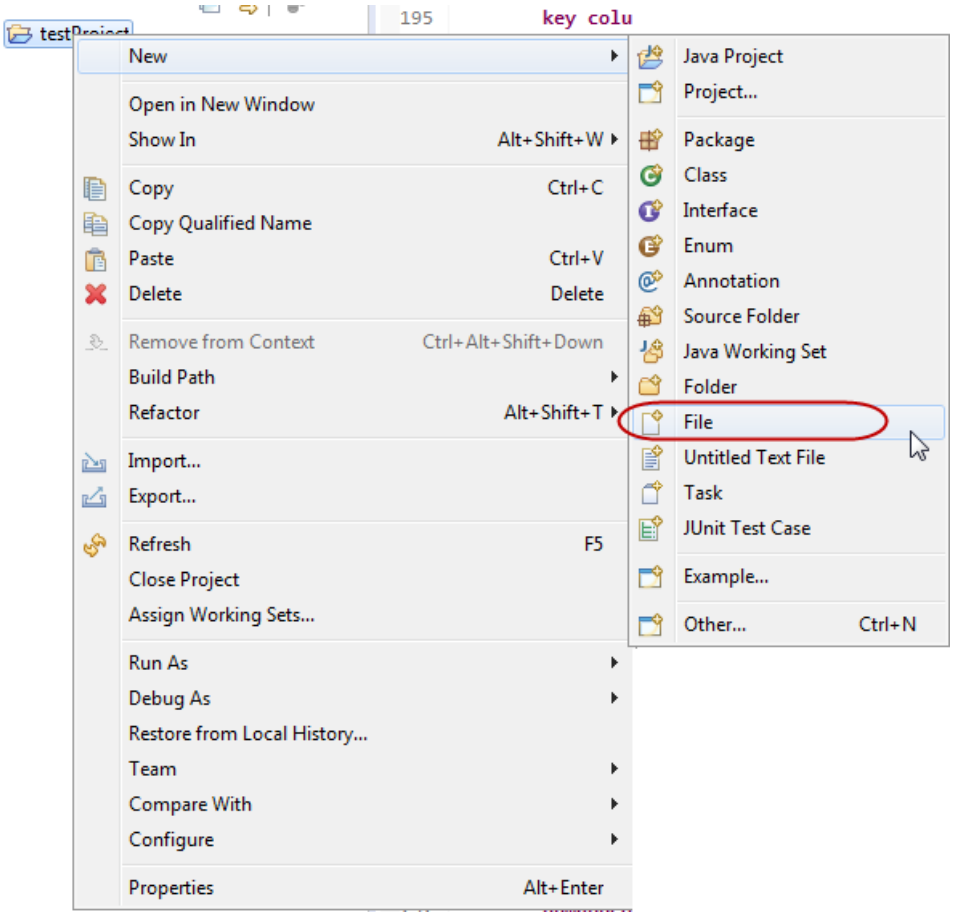
1. In Eclipse, select **File-> New-> Project-> General Project**.



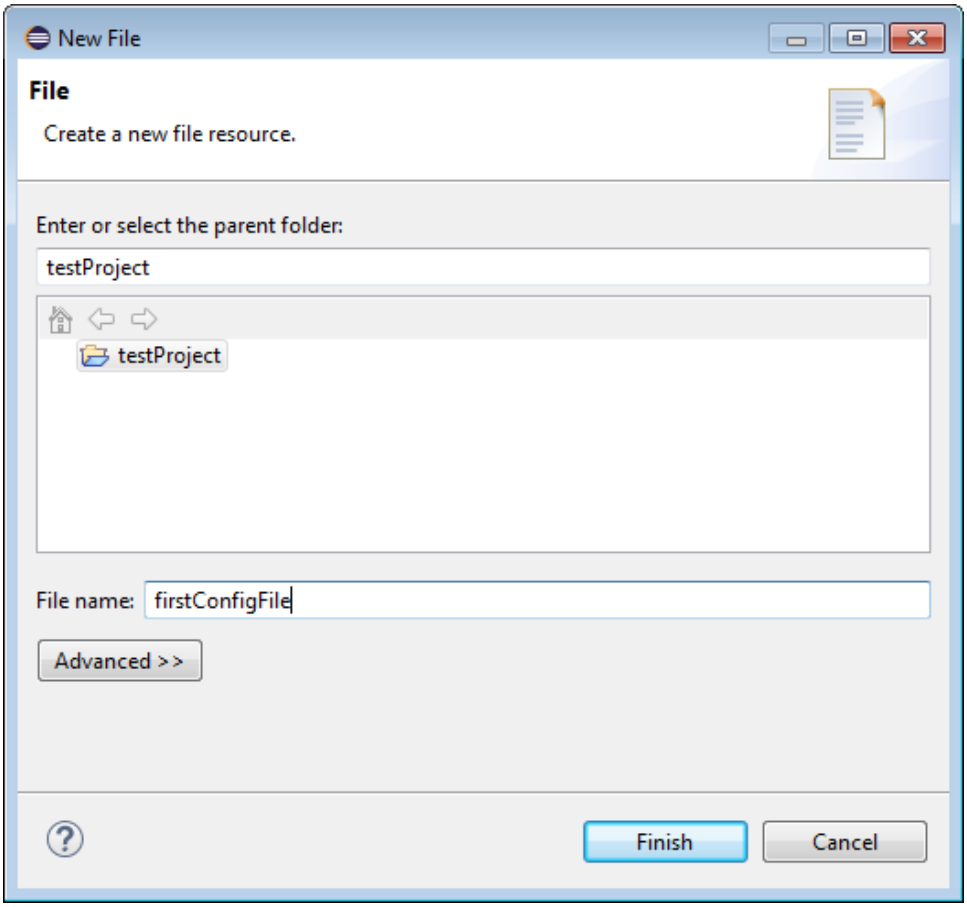
2. Click **Next**.
3. Enter a name for the project.



4. Check default location to save the project to your default Eclipse workspace. Otherwise, enter the path to the workspace you want to save the project to.
5. Click **Finish**.
6. Right-click on the newly created project, then select **New->File**.



7. Enter a name for the new file.



8. Click **Finish**.

1.5 Launching MapleMBSE from the Command Line

MapleMBSE can be launched from the command line by executing the MapleMBSE.exe command in the installation directory. The command has the following syntax.

MapleMBSE.exe [/R <memsize>] [/L <logfile>] [/E <excel.exe>] [[/I | CONFIG(*.MSE)] [MODEL]]

- If you specify **CONFIG** (must be *.MSE) and **MODEL** files, it launches MapleMBSE with these fiLineles. If you do not specify a **MODEL** or **CONFIG** file, you will be asked for these files in a subsequent dialog.
- If you specify **/I**, MapleMBSE.exe will always ask for the MSE file in a dialog and regard the first non-slash argument as a model file.

- **MODEL** can be a local file or URI. For example, consider a model file on the Teamwork Cloud server:
  = "twc://[SERVER:PORT][/PROJECT][?[username=USERNAME][&password=PASSWORD]][&branch=BRANCHNAME][&version=VERSION]"

Note: If you omit some items, MapleMBSE will ask for them in the dialog

- If you do not specify any file arguments, it checks if MapleMBSE.xlsm exists in the AddIns directory inside the directory with MapleMBSE.exe. If found, MapleMBSE launches it.
- **/R <memsize>**: (optional, default=905) Memory address space in MB to be reserved for Java VM in the Excel process.
- **/L <logfile>**: (optional, default="MapleMBSE_Launcher.log") Log file name, relative to %USERPROFILE% or in absolute path.
- **/E <excel.exe>**: (optional) Path of EXCEL.EXE
  If it's not specified, the registry entries will be used:
  = HKEY_CLASSES_ROOT\CLSID\<CLSID of "Excel.Application">\LocalServer32
  = HKEY_CLASSES_ROOT\CLSID\<CLSID of "Excel.Application">\LocalServer
  = HKEY_CLASSES_ROOT\Wow6432Node\CLSID\<CLSID of "Excel.Application">\LocalServer32
  = HKEY_CLASSES_ROOT\Wow6432Node\CLSID\<CLSID of "Excel.Application">\LocalServer

Notes:

- After you have launched MapleMBSE successfully from the command line, if you want to restart MapleMBSE, use the shortcut key combination CTRL+Shift+G.
- You can launch MapleMBSE from the command line using &cache=true. For example:

MapleMBSE.exe "C:\Program Files\MapleMBSE 2021\Example\TWC\19.0\TWC-Sample.MSE" "twc://teamworkcloud:3579/Model?username=****&password=****&cache=true"

1.6 An Introductory Example

In this example, we want to define a configuration that allows us to view and update top-level packages in a UML model. The first step is to import the definition of a UML metamodel. A metamodel, called an Ecore, defines the types of elements a UML model may have and their relationships. The definitions inside the configuration file that allow us to access different elements of a model rely on the structures defined by the imported metamodels. To import an Ecore metamodel, use an **EcoreImport** construct as follows.

```
import-ecore "http://www.eclipse.org/uml2/5.0.0/UML"
```

There are two steps in converting model data into its representation in Excel. First, we define a **SyncTable Schema** that converts the data into an intermediate table called a **SyncTable**. In the second step, the **SyncView Layout** defines how a SyncTable is displayed on a spreadsheet by specifying which SyncTable columns will be displayed, as well as their position and layout. The resulting part of the spreadsheet displaying the SyncTable is called the **SyncView**.

A SyncTable Schema defines how a set of model elements is mapped to a table structure. In this example, we define a SyncTable Schema called PackagesTable.

```
synctable-schema PackagesTable{
    
}
```

Note the MSE Editor performs some testing of the correctness of the defined structures. The syntax error highlighting the closing bracket indicates that definition is incomplete without defining a dimension.

We define the top level dimension to be an element of a Package type.

A dimension is a basic structure of a SyncTable schema. Each dimension corresponds to a model element. The first dimension of a SyncTable Schema is a Top Level Dimension. It represents the type of element to which the schema applies. Each following dimension is defined with respect to the preceding one.

```
synctable-schema PackagesTable{
    dim [Package]
}
```

A dimension consists of columns. Each column represents an attribute of the element that the dimension describes. To identify the element some of the columns must be designated as key columns. They must represent the attributes of the element that would allow you to identify it uniquely. Without the definition of the key column(s) the definition of the dimension is incomplete. It is indicated by a syntax error.

For a package, its name can identify it uniquely. We define a key column that corresponds to the 'name' attribute of a Package class.

```
synctable-schema PackagesTable {
    record dim [Package] {
        key column /name as PackageName
    }
}
```

This SyncTable schema definition allows you to view, add and delete packages by referring to their name. To create a SyncTable, the schema must be applied to a **Data Source**. A Data Source defines a set of model elements. The Data Source representing the top-level data

structure of a model has the name **Root**. This is a reserved name. The Root is declared as follows.

```
data-source Root[Model]
```

The declaration specifies that the type of the top-level data structure is Model. You can see the types and the structure of a UML model by opening the .uml file in a text editor. For example, the following is a snippet from UserGuide.uml in the installer, found in the **<MapleMBSE>\Examples\UserGuide** directory, where **<MapleMBSE>** is your MapleMBSE installation directory.

```
<?xml version="1.0" encoding="UTF-8"?>
<uml:Model xmi:version="20131001"
xmlns:xmi="http://www.omg.org/spec/XMI/20131001"
xmlns:uml="http://www.eclipse.org/uml2/5.0.0/UML"
xmi:id="_D2UUEM_MEee6666BhKb4Cg" name="UserGuide">
  <packagedElement xmi:type="uml:Package" xmi:id="_Oeqy0M_MEee6666BhKb4Cg"
 name="Package1" visibility="public">
    ...
  </packagedElement>
  <packagedElement xmi:type="uml:Association" ...>
    ...
  </packagedElement>
  ...
</uml:Model>
```

The text representation of the model is written in XML. The model and its content are represented by XML elements. The top-level element is defined by the start and end tags:

```
<uml:Model ...>
  ...
</uml:Model>
```

The element is a "uml:Model" that is of a type Model defined by the "uml" namespace. The "uml" namespace is defined among the attributes of the Model element.

```
xmlns:uml="http://www.eclipse.org/uml2/5.0.0/UML"
```

The definition matches the EcoreImport we are using in the configuration file. So the type "Model" used in the definition of the Root Data Source is the same as "uml:Model" in the model file. We want to apply the PackageTable schema to define packages inside a model, and the type of the data source it applies to is Package. We define a data source that represents packages in a model as follows.

```
data-source topPackages=Root/packagedElement[Package]
```

This statement defines a new data source called `topPackages`. The syntax `Root/packagedElement` means we are looking at the Model elements inside the **Root** that are defined by the tag `packagedElement`. The syntax `packagedElement[Package]` means we are choosing only those `packagedElements` that have type `Package`. Looking back at the UML file, we can see that `packagedElement` could have at least two types: `Package` or `Association`. We are choosing only the ones of type `Package`.

To create a SyncTable we apply the SyncTable schema to the Data Source using angle brackets.

```
synctable packagesTable = PackagesTable<topPackages>
```

The next step is to define how the SyncTable is represented in an Excel worksheet. We do this by defining a **Worksheet Template**. We define a Worksheet Template called `Packages`.

```
worksheet-template Packages
```

The template uses one argument `p` of type `PackagesTable`. That is, `p` must be a SyncTable created from the SyncTable schema `PackagesTable`.

```
worksheet-template Packages(p : PackagesTable){
    
}
```

The Worksheet Template must define where the SyncView for the given argument is placed and what orientation it has. In this example we choose the SyncView for the argument p to be a vertical table (named tab1) and start in cell B3 (row 3, column 2). The section defining tab1 is called **SyncView Layout**.

```
worksheet-template Packages(p : PackagesTable){
    vertical table tab1 at (3,2) = p {
        
    }
}
```

We also need to specify which columns of the SyncTable should be included in the SyncView. A SyncTable column becomes a field in a SyncView record. A record is simply the collection of fields. In a vertical SyncView a record is a row in the table and the fields are the cells in the row (see the **Operations Overview** section in **Chapter 2** of the **MapleMBSE User Guide** for more details). Some fields in a record must be marked as key fields to indicate that those fields are used to identify the record uniquely. In the PackageTables schema there is only one column, **PackageName**. It is a key column and should be used as a key field. PackageName is of the String data type. The definition of the SyncView layout is then as follows.

```
worksheet-template Packages(p : PackagesTable){
    vertical table tab1 at (3,2) = p {
        key field PackageName : String
    }
}
```

We also want to indicate that the column should be sorted in ascending order when a model data is loaded or when sort operation is performed after adding new data. We do so by specifying the name of the field in the sort keys.

```
worksheet-template Packages(p : PackagesTable) {
    vertical table tab1 at (3,2) = p {
        key field PackageName : String
        sort-keys PackageName
    }
}
```

Finally, we need to define a workbook that consists of a worksheet based on the defined template applied to an instance of a SyncTable.

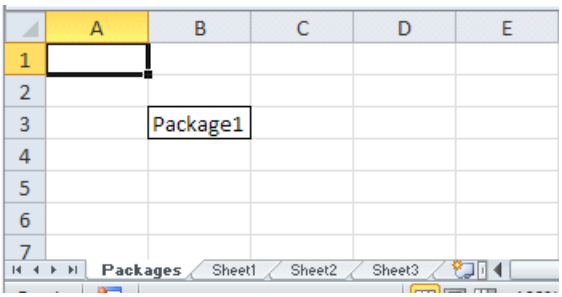
```
workbook {
    worksheet Packages(packagesTable)
}
```

The final content of the configuration file is as follows.

```
import-ecore "http://www.eclipse.org/uml2/5.0.0/UML"


data-source Root[Model]
data-source topPackages=Root/packagedElement[Package]

synctable-schema PackagesTable {
    record dim [Package] {
        key column /name as PackageName
    }
}

synctable packagesTable = PackagesTable<topPackages>
worksheet-template Packages(p : PackagesTable) {
    vertical table tab1 at (3,2) = p {
        key field PackageName : String
        sort-keys PackageName
    }
}

workbook {
    worksheet Packages(packagesTable)
}
```

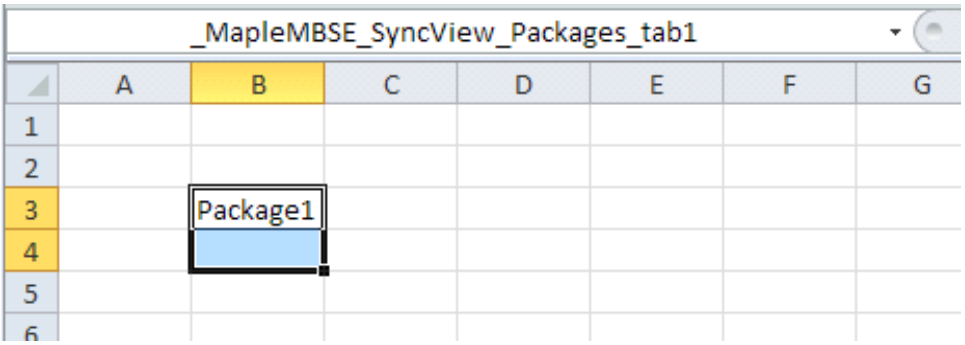
The resulting file can be found in **GettingStarted.MSE**, in the MapleMBSE Configuration Editor Package.

You can use the configuration file with any UML model. For example, opening MapleMBSE with this configuration file and **<MapleMBSE>\Example\UserGuide\UserGuide.uml**, where **<MapleMBSE>** is the location where MapleMBSE is installed, gives the following result.



The SyncView area of the worksheet can be highlighted by choosing the name of the corresponding SyncView in the name box. The SyncView name has the following format.

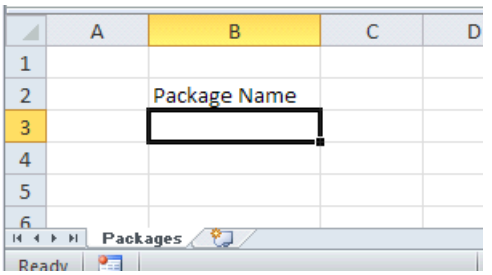
```
_MapleMBSE_SyncView_<Worksheet Name>_<SyncView Layout Name>
```



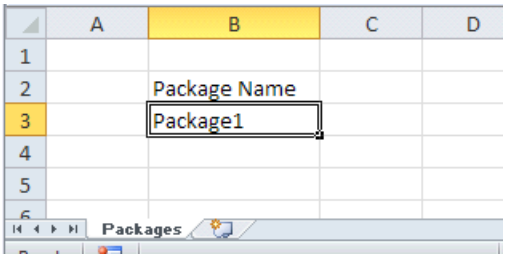
You can add new packages to the model by adding rows in the SyncView area or by entering them in the insertion area (cell B4). See the *MapleMBSE User Guide, Chapter 2, Adding Model Elements* for more details.

For convenience, it is good to add a heading to the column explaining what it is and maybe change the width of the column. Any such formating changes done when editing a model are not saved with the model data. Instead, they should be done in a separate file called **Template File**. A template file is an Excel file that has the same base name as the configuration file and is placed in the same folder. MapleMBSE looks for the sheets in the workbook that match the names of the worksheets defined by the configuration file and loads the specified SyncViews into that sheet. To define a template file for our example, we need to create an Excel file with the name that matches the name of the configuration file and contains a sheet called Packages.

To create the template for this example, we define a new Excel workbook. We name one of the sheets Packages, and delete others. The data with the package name is displayed in column B starting with row 3. We can define the heading for the column in cell B2 and increase the width of the column.



We save the template file with the same base name as the configuration file and in the same folder. Now if we open MapleMBSE with the configuration file and the example model UserGuide.uml we get the following.



Tip: the template file for this example can be found in **GettingStarted.xls**, in the same place with **GettingStarted.MSE**.

Another way to create a template could be to open a model with the configuration file as we did before, then save it as an Excel file using **Add-Ins > MapleMBSE > Export To Excel File**. This way we have the right number of sheets with their names. It is also easier to judge where the headings need to be added and how wide the columns should be. Any model data loaded in the tables should be removed. If it is left in the template it may create confusion when MapleMBSE uses the template. MapleMBSE will load SyncViews according to the specifications in the configuration file, so some data may be overwritten and some may not, depending on the model file with which the template is opened.

2 Configuration Language Fundamentals

2.1 Notation

The formal grammar of MSE Configuration Language is given using a simple Extended Backus-Naur Form (EBNF) notation. Each rule in the grammar defines one symbol, in the form:

```
symbol ::= expression
```

The following notations are used in expressions.

Notation	Usage
`'string'`	literal string matching the string between the quotes
`(expression)`	`expression` is treated as a unit
`A*`	0 or more occurrences of A
`A+`	1 or more occurrences of A
`A?`	0 or 1 occurrence of A
`A | B`	A or B
`<A>`	name of an element of type A

For reference see https://www.w3.org/TR/2008/REC-xml-20081126/#sec-notation

2.2 Overview of an MSE Configuration File

The following is the formal definition of the configuration file.

```
MSEConfiguration ::= EcoreImport*

WorkbookInstance &
( DataSource
| SyncTableSchema
| SyncTable
| WorksheetTemplate
)*
```

In MSEConfiguration, EcoreImports come first, and then other elements can be specified in any order. The definitions of the elements are given in the following chapters. The following is an example of the procedure for writing an MSE Configuration file.

1. Define a Data Source and a SyncTable Schema.
2. Define a SyncTable with the pair of Data Source and SyncTable Schema.
3. Define the view and the layout of the SyncTable on WorksheetTemplate.

4. Define a worksheet in the WorkbookInstance with the pair of the WorksheetTemplate and SyncTable.

2.3 EcoreImport

EcoreImport declares the type of model to be edited with the configuration file. A type of model is defined by specifying an IRI of a metamodel definition. A metamodel, called an Ecore, defines types of elements a model may have and their relationship. Model elements and their attributes are queried using the structural elements defined by EcoreImports. The formal syntax of EcoreImport declaration is as follows.

```
EcoreImport  ::= 'import-ecore' '"'IRI'"' ('as'
                 ID)
```

In the *'import-ecore' '"'IRI'"' ('as' ID) (page 20)*, `IRI` is an identifier of the Ecore metamodel in the form of IRI (International Resource Identifier). Different types of models have their own metamodels. The following is a list of the available Ecore models.

Type	**IRI**
UML	http://www.eclipse.org/uml2/4.0.0/UML
SysML	http://www.eclipse.org/papyrus/sysml/1.4/SysML
Teamwork Cloud 18.5	http://www.nomagic.com/magicdraw/UML/2.5
Teamwork Cloud 19.0	http://www.nomagic.com/magicdraw/UML/2.5.1
Rhapsody	http://w3.ibm.com/Rhapsody/api/
MapleMBSE metamodel	http://maplembse.maplesoft.com/common/1.0

3 Qualifiers

Qualifiers are used as a way to specify the type of model elements that you want to query or create.

The basic qualifier syntax is as follows:

`[Classifier|filter="value"]` where a *Classifier* is a type of element, such as a *Class*.

The `filter` must be an E*Attribute*, which is a property that belongs to the *Classifier* element. There is an option to use EReference as filter but the syntax differs.

The formal definition of a Qualifier is given in the table below.

`Qualifier`	`::=`	`'['`
		
		`(EcoreImport'::')? EClassifier`
		`('|' index=INT)?`
		
		`('|' FeatureFilter (',' FeatureFilter)*)?`
		
		`']'`

`FeatureFilter`	`::=`	`AttributeFilter | ReferenceFilter| PredicateFilter`
`AttributeFilter`	`::=`	`(EcoreImport'::')? EAttribute'='STRING`
`ReferenceFilter`	`::=`	`(EcoreImport'::')? EReference'='? Qualifier`

3.1 Attribute filter

Filters, either attribute or reference based, are applied in 2 scenarios, when gathering elements from the model and when creating new elements from scratch.

The first scenario, when querying the model, attribute filters simply verify that the real value and the value used in the configuration files are the same. Then, only those that match are added to the syncview.
Conversely, the attribute filter when creating a new element, has a different meaning. In this scenario, this kind of filter is initializing some attributes of the newly created element. For example, the following qualifier `[Property|aggregation="composite"]` has 2 functions. The first one, displaying only those properties with AggregationKind equals to composite, and second, initializing the aggregation attribute with the value composite.

3.2 Reference filter

Reference filter has also the querying-creating duality, despite the different natures of attributes and references. The differences between attributes and references are analogous to the differences of basic types and objects like in other programming languages. References are used to point to hard-typed Classifiers, which have their owned attributes and references.

Providing an excepted value for an attribute is easy; strings, integer, boolean, and other basic types have the same values each time the model is queried. But it is impossible to provide a constant value for a reference and use it to filter, pointers change each time, the memory addresses are not constant in the model. This is why both filters have different syntaxes. A reference filter uses an inner qualifier to describe the kind of element that MapleMBSE is expecting to match. For the querying is as simple as the attribute filter, but there are some special cases to consider when initializing.
While querying with the following qualifier `[Property|type=[Class|name="block"]]`, the inner qualifier is helping to filter all Classes named block, and only Properties with a type reference to such classes would be displayed.
The reference filtering initialization has different behaviors depending on the type of reference. EReferences have properties of their own, like containment, multiplicity, and derived. Containment refers to the fact that a reference subsets ownership, for instance, packagedElement is a containment for Package, or slot is a containment for InstanceSpecification. Derived means that the value is a calculation of other attributes, ownedElement is a derived reference. Depending on which one of the reference filters is being used MapleMBSE would create a new element, refer to an existing one, or simply do nothing. Generally, if it is containment the filter initialization would create a new element, if it is derived the filter would not change a thing, and the rest of the time if there is a single element result of the inner qualifier in the whole model then a reference would be initialized to that Element. For example, let us revisit the previous qualifier `[Property|type=[Class|name="block"]]`, but this time during creation. The outer qualifier dictates what kind of Classifier is being instantiated, in this case a Property. The reference type is a non-containment, non-derived, and single valued reference; this means that the inner qualifier would try to find a single Class named block to fulfill the type reference while initializing the filter. If a unique Class named block exist in the model, then the newly created Property would be typed using a reference to that Class. Another example could be `[Class|ownedAttribute=[Port|type=[Class|name="block"]]]`, this illustrates how it is possible to nest filters. When creating a element with such qualifier, a Class would be instantiate and also a Port, this is due to the fact that ownedAttribute is a containment reference. Also, the Port that is created would be typed as mentioned before.

3.3 Predicate filter

A Predictate is a function that returns a boolean (true or false) value.

MapleMBSE uses predicates in the context of filtering.

The three steps involved in creating a predicate filter:

1. Defining the Predicate
2. Using the Predicate
3. Creating the Filter Logic for the predicate.

Defining the Predicate

```
predicate hasnotype := NOT Port.type[Class]
```

Creating the Filter Logic for the predicate

Unary Predicates

MapleMBSE predicates that take a single argument and return a boolean value.

NOT

The NOT predicate syntax is `NOT(Predicate)`.

For example, `NOT Port.type[Class]`

N-ary Predicates

MapleMBSE predicates that can take multiple arguments and return a boolean value.

OR, AND

The OR/AND syntax is: `[OR|AND]( Predicate (; Predicate)+ )`
This means that either OR or AND can take multiple predicates as arguments, with each predicate separated by a semicolon.

OR1

This n-ary operations check that exactly 1 predicate is true in order to return true. The syntax is of the form:
`OR1 ( Predicate (; Predicate)+ )`

Propositions

Unary and n-ary predicates do not query the model data. Those predicates delegate the matching to their children. Propositions, however, have no children and do query the model data. Proposition predicates are the leaves of the predicate tree. They are responsible for matching the data and the custom filter.

Attribute Proposition

Using an EAttribute, this proposition can match a string to a value stored in the model. If

multiple strings are given, they constitute an implicit OR. The syntax is of the form:

```
(EcoreImport::)?(EClassifier/)?EAttribute = value(,value)*
```

Reference Proposition

Using an EReference, this proposition can match Qualifiers to the objects stored in the model. If multiple qualifiers are given, they constitute an implicit OR. The syntax is of the form:

```
(EcoreImport::)?(EClassifier [/ | .])? EReference Qualifier(,Qualifier)*
```

Subset Proposition

Sometimes, MapleMBSE returns comma-separated list of strings, and instead of matching the same concatenated string, it would be better to perform the subset operation. For this subset proposition, the syntax would be of the form:

```
(EcoreImport::)?(EClassifier /)?EAttribute [SUBSET|SUPERSET] value(,value)*
```

Inequality Proposition

SysML supports numeric basic types and MapleMBSE should be able to create predicates using those types. The most powerful way to take advantage of numeric values is with inequality operations of the form:

```
(EcoreImport::)?(EClassifier/)?EAttribute [=|!=|<|<=|>|>=]=? intValue
```

Counting Proposition

There are times when a user wants to filter elements containing a specific amount of sub-elements. To accomplish this, use counting propostions of the form:

```
COUNT (EcoreImport::)?(EClassifier [/ | .])?EAttribute [=|!=|<|<=|>|>=]=? intValue
```

Using the Predicate Filter

When using Predicate filtering, the predicate filter syntax should be:

`/qpe[Classifier| named_predicate]` where /qpe is the query path expression used to navigate from one element to another, in this case `/ownedAttribute.` Classifier, as mentioned earlier in this chapter is a type of element. In the example below, the Classifier is `Port`. The reference filter is `mse::metaclassName="MD Customization for SysML::additional_stereotypes::ConstraintParameter"`, followed by the attribute filter `aggregation="composite"` and finally the predicate filter, `has-notype.`

dim /ownedAttribute[Port|mse::metaclassName="MD Customization for SysML::additional_stereotypes::ConstraintParameter", aggregation="composite",hasnotype]

4 Query Path Expression

4.1 Query Path Expression Definition

Query Path Expression is an expression that queries the model for model elements and attribute values. It is used in defining Data Sources and SyncTable Schemas. The formal syntax definition is as follows.

```
QueryPathExpression  ::=  ( LocalQueryExpression )+ ( '@'
                          ReferenceDecompositionId )?
LocalQueryExpression ::=  ( ( '/' AttributeId ) | ( '.' ReferenceId ) )
                          Qualifier?
AttributeId          ::=  ( EcoreImportId '::' )? <Attribute>
ReferenceId          ::=  ( EcoreImportId '::' )? <Reference>
Qualifier            ::=  '[' ClassifierId ( '|' AttributeFilter (','
                          AttributeFilter)* )? ']'
ClassifierId         ::=  ( EcoreImportId '::' )? <Classifier>
AttributeFilter      ::=  AttributeId '=' '"' <Expression> '"'
```

`ReferenceDecompositionId` refers to ID of a ReferenceDecomposition defined in *ReferenceDecomposition and ReferenceQuery (page 40)*.

`<Classifier>, <Attribute>, <Reference>` refer to the corresponding UML elements, *Classifier*, *Attribute* and *Reference*. The names and their types are defined by a metamodel (via *EcoreImport (page 20)*). In Query Path Expressions we distinguish the following three types.

- **Classifier**
  A type of an element. For example, a UML model may have elements of type Class. Class is a Classifier. An element contains subelements which can be of two types: attributes and references.
- **Attribute**
  A subelement that belongs to the element.
- **Reference**
  A subelement that refers to another element.

To illustrate these types and their relations consider the example code below. The code is a snippet from the UML example model from the MapleMBSE User Guide.

Tip: The model file, UserGuide.uml, can be opened using any text editor. It can be found in the installation folder **<MapleMBSE>/Example/UserGuide**, where **<MapleMBSE>** is the location of your MapleMBSE installation.

```
...
 <packagedElement xmi:type="uml:Class" xmi:id="_vXfqAM_MEee6666BhKb4Cg"
name="Class1">
      <ownedAttribute xmi:id="_AH4akdBoEee6666BhKb4Cg" name="Property1"
visibility="protected" type="_tsYvoNBnEee6666BhKb4Cg" aggregation="shared"
association="_AH3McNBoEee6666BhKb4Cg">
        <lowerValue xmi:type="uml:LiteralInteger"
xmi:id="_AH4aktBoEee6666BhKb4Cg" value="1"/>
        <upperValue xmi:type="uml:LiteralUnlimitedNatural"
xmi:id="_AH5BoNBoEee6666BhKb4Cg" value="1"/>
      </ownedAttribute>
      ...
    </packagedElement>
    <packagedElement xmi:type="uml:Class" xmi:id="_tsYvoNBnEee6666BhKb4Cg"
name="Class2" visibility="private"/>
...
  <packagedElement xmi:type="uml:Association"
xmi:id="_AH3McNBoEee6666BhKb4Cg" name="aggregation_class2_in_class1"...>
   ...
  </packagedElement>
...
```

The text representation of the model is written in XML. The model and its content are represented by XML elements. An element can be defined as an empty element with attributes.

```
<element ... />
```

Or if it contrains other elements it can be defined using the start and end tags.

```
<element> ... </element>
```

The classifiers are highlighted in blue: "uml:Class", "uml:LiteralInteger", "uml:LiteralUnlimitedInteger", "uml:Association". The "uml" namespace is defined in the definition of the Model element, see *Getting Started (page 1)*. Consider the ownedAttribute element Property1 in Class1. The attributes of the element are highlighted in green: name, visibility, aggregation, lowerValue, upperValue. The references of Property1 are highlighted in orange: type and association. You can see that the values of the references are the IDs of the elements they refer to.

The names of Classifiers, Attributes, and References can be written with or without EcoreImportId depending on how EcoreImport was declared. If there is only one EcoreImport in a configuration file and it was declared without an ID:

```
import-ecore "http://www.eclipse.org/uml2/5.0.0/UML"
```

EcoreImportId is not necessary. In this case, a query path expression that queries elements of a package can be written as follows.

```
/packagedElement
```

If an EcoreImport was declared with an ID:

```
import-ecore "http://www.eclipse.org/uml2/5.0.0/UML" as uml
```

EcoreImportId must be used to refer to classifiers, attributes, or references defined by the corresponding model. The same Query Path Expression takes the form.

```
/uml::packagedElement
```

In the following examples we omit EcoreImportId. The above examples of Query Path Expressions query all elements in a package. For the above example of a query path expression, it would include elements of types Class and Association. If we want to specify that only elements of Class type should be queried we need to specify a Qualifier:

```
/packagedElement[Class]
```

A qualifier can include one or more FeatureFilters. For example, to query a class inside a package called Class1, the following Query Path Expression can be used.

```
/packagedElement[Class|name="Class1"]
```

The examples we have considered so far consisted of single LocalQueryExpressions. LocalQueryExpressions can be combined to query nested objects. Each subsequent LocalQueryExpression applies to the result of the previous LocalQueryExpression. For example, to query attributes (the ownedAttribute elements) inside Class1 inside a package, the following Query Path Expression can be used.

```
/packagedElement[Class|name="Class1"]/ownedAttribute
```

So far, we have only used attributes in query expressions. To query a type of an ownedAttribute in a class a reference must be used.

```
/packagedElement[Class|name="Class1"]/ownedAttribute.type
```

The result of the query is the element that the 'type' reference refers to. For Property1, it would return class Class2. Another way to specify a reference is to add the specification of ReferenceDecomposition at the end of the Query Path Expression.

```
/packagedElement[Class|name="Class1"]/ownedAttribute.type @
ReferenceDecompositionId
```

ReferenceDecomposition is defined in Chapter 6, see *ReferenceDecomposition and ReferenceQuery (page 40)* . ReferenceDecomposition is a description of the referenced object. For display purposes there is no difference between a reference query with and without the use of ReferenceDecomposition. However, when updating a field specified by a reference without a ReferenceDecomposition, the updates apply to the referenced object. Whereas, with a ReferenceDecomposition the updates may change which object the reference points to. It is not recommended to use references without ReferenceDecompositions. If necessary, they should only be used in read-only worksheets.

5 Data Source

Data Source defines a set of model elements. A Data Source is combined with a SyncTable Schema to create a SyncTable for the model elements defined by the Data Source. The following is the formal definition of Data Source.

```
DataSource       :: =  PrimaryDataSource | ChainedDataSource
PrimaryDataSource :: =  'data-source' (ID|'Root'|'ROOTS') '*'?
                        Qualifier
ChainedDataSource :: =  'data-source' ID '=' DataSource
                        ObjectQueryExpression
```

`Root` is a reserved Data Source name that refers to the top-level model element. The type of the top-level model element depends on the type of a model. Definitions of the Root Data Source are based on the type of model, as shown in the *Root Data Source Definition (page 31)* table.

Type of Model	Root Data Source Definition
UML, SysML, Teamwork Cloud	`data-source Root[Model]`
Rhapsody	`data-source Root[REProject]`

A ChainedDataSource applies Query Path Expression to the result of the parent Data Source. Consider the example in the Figure below based on UserGuide.uml model (found in **<MapleMBSE>/Example/UserGuide**, where **<MapleMBSE>** is the MapleMBSE installation directory). The Figure *Figure 5.1 (page 31)*) shows the relationship between the elements. For each element its classifier is given in italics. The elements enclosed in boxes with dashed lines are included in the corresponding data sources defined below.

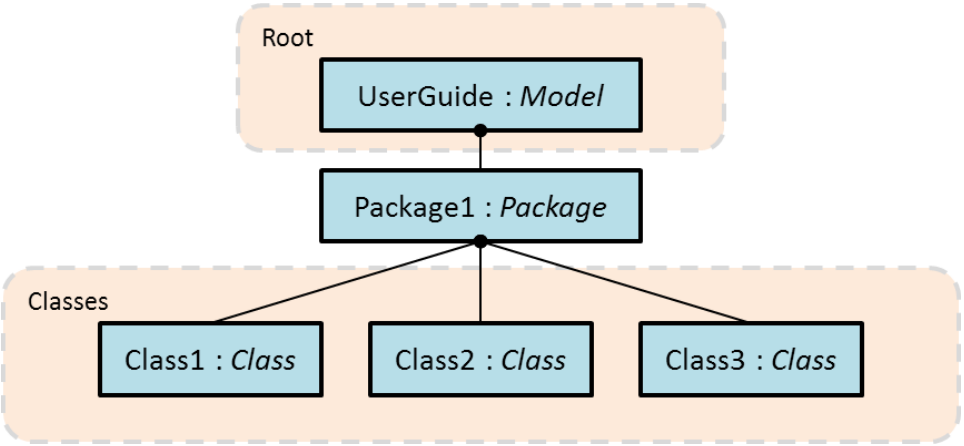


Figure 5.1: Relationship between model elements

- **Primary Data Source**
  The example is a UML model, so the Primary Data Source is defined as follows.

```
data-source Root[Model]
```

In the code snippet above, the data-source retrieves the top-level element of the UML user model (user resource).

- **All Primary Data Source**

```
data-source classes*[Class]
```

In the code snippet above, the data-source retrieves all Classes regardless of their location inside the resource set. In other words, all Classes are retrieved, whether they are the model, or outside the model (for example, user resource, project resource, etc. or the model. In the example above Class1, Class2 and Class3 are retrieved.

This type of Primary Data Source is very useful, however, it should be used only to make read-only *SyncTable (page 47)* and *ReferenceDecomposition and ReferenceQuery (page 40)*reference-decomposition.

Example for primitive DataTypes:

```
...
data-source allDataTypes*[DataType]
...
synctable-schema DataTypesSchema {
    dim [DataType]{
        key column /name as typeName
    }
}

synctable-schema PropertySchema(dts: DataTypesSchema) {
    dim [Property] {
        key column /name as propertyName
        reference-query .type @ typeRef
        reference-decomposition typeRef = dts {
            foreign-key column typeName as propertyType
        }
    }
}
...
synctable dataTypesTable = DataTypesTable<allDataTypes>
synctable propertyTable = PropertySchema<properties>(dataTypesTable)
...
```

- **Chained Data Source**
  The following Data Source, called "classes", defines a set of all classes in Package1. It is defined by applying an Query Path Expression to a previously defined data source. In this case, the top-level data source, `Root`. In the example shown in *Figure 5.1 (page 31)*, `Root` is `UserGuide`.

```
data-source classes = Root/packagedElement[Package|name="Package1"]/packagedElement[Class]
```

6 SyncTable Schema

6.1 SyncTable Schema Definition

A SyncTable schema specifies how model elements are mapped to a logical table. With data sources explained in *Data Source (page 31)*, model elements are first organized as trees, and then mapped to tables. Such tree nodes are defined by *dimensions* in SyncTable schema, which identifies a model element by key columns. The formal syntax of `SyncTableSchema` is defined as:

```
SyncTableSchema      ::= 'synctable-schema' ID ( '(' SyncTableParam ( ','
                         SyncTableParam )* ')' )?
                         '{' TopLevelDimension   AbstractDimension* '}'
SyncTableParam       ::= ID ':' SyncTableSchemaId
TopLevelDimension    ::= ('record')? 'dim' Qualifier '{' DimensionMember*
                         '}'
AbstractDimension    ::= SuccessiveDimension | DimensionGroup
SuccessiveDimension  ::= ('record')? dim QueryPathExpression '{'
                         DimensionMember* '}'
DimensionGroup       ::= ('alternative'|'optional'|'group') '{'
                         DimensionMember* '}'
DimensionMember      ::= PropertyMapping | ReferenceDecomposition
PropertyMapping      ::= AttributeColumn | ReferenceQuery
AttributeColumn      ::= KeyAttributeColumn | NonkeyAttributeColumn
KeyAttributeColumn   ::= 'key' 'column' ObjectQueryExpression 'as' ID
NonkeyAttributeColumn ::= 'column' ObjectQueryExpression 'as' ID
```

where `SyncTableSchemaId` is ID of a `SyncTableSchema`, and `TopLevelDimension` appears first as defined in the formal syntax, and we need to put a qualifier to specify what model element types are selected, then `SuccessiveDimension` follows in which we put a Query Path Expression to query what model elements are selected as dimensions. In this chapter, we explain how to specify SyncTable schemas through examples.

6.2 Examples of SyncTable Schema

First, we show a simple SyncTable Schema as follows:

```
synctable-schema PkgCls {
  dim [REPackage] {
     key column /name as PkgName
    column /description as PkgDesc
  }
  record dim /nestedElements[REClass] {
    Key column /name as ClsName
  }
}
```

Here we define a SyncTable Schema with an ID called `PkgCls` and it consists of two Dimensions. `[REPackage]` in the top level dimension means it picks up `REPackage` model elements, and it must be consistent with that in data sources. The next dimension picks up `REClass` elements in `nestedElements` feature of the top level dimension. By applying this schema to **Pkg1**, **Pkg2** of the data source having **Figure 5.1**, we obtain two trees as shown in **Figure 6.1**, where **Pkg1** and **Pkg2** belong to the top level dimensions; and **Cls1** and **Cls2** belong to the next dimensions.

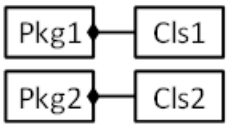


Figure 6.1: SimpleTree

6.3 Mapping the Attribute Values of the Model Elements to the Columns

The trees in the example above are translated into tables by the column definitions. The top level dimension has `PkgName` and `PkgDesc` columns, and they are filled with the QPEs of `/name` and `/description`, respectively. And the next dimension have `ClsName`

column, which is filled with the QPE of `/name`. Then the tree in **Figure 6.1** is translated to:

Pkg1	PkgDesc1	Cls1
Pkg2	PkgDesc2	Cls2

Figure 6.2: SyncTable From Simple Tree

More formally speaking, each path in the trees is translated into record, and then we have two records from the paths of **Pkg1-Cls1** and **Pkg2-Cls2**. Note that synctable schema determines all of the columns in a static way. They are, in this example, `PkgName`, `PkgDesc`, and `ClsName`, and the number is three.

6.4 Mapping the Dimensions to the Records

Let us look at how dimensions are mapped to records in more detail by comparing with the example below. The only difference from the previous example is the `record` keyword in the top level dimension highlighted with bold font.

```
synctable-schema PkgCls {
    dim [REPackage] {
        key column /name as PkgName
        column /description as PkgDesc
    }
    record dim /nestedElements[REClass]{
        key column /name as ClsName
    }
}
```

If any other conditions are the same as the above, the trees generated by this schema are exactly the same as in **Figure 6.1**. However, because the top level dimension has a record keyword, the table has more records as shown in **Figure 6.3**. The added records are the first and third rows, which come from the top level dimension. Note that the last dimension (in this example, that is the one corresponding to **Cls**) always creates records even if it is missing. In this table, the rightmost column in the first and third rows is specially treated as EMPTY. They will be shown as blank cells with light gray backgrounds, and distinguished from the usual blank cells

Pkg1	PkgDesc1	
Pkg1	PkgDesc1	Cls1
Pkg2	PkgDesc2	
Pkg2	PkgDesc2	Cls2

Figure 6.3: SyncTable From Simple Tree (add record keyword to top level Dimension

Note that each record corresponds to one model element. In this example, the first record corresponds to **Pkg1**, and the second one corresponds to **Cls1** while the previous example does not have any records corresponding to **Pkg1** nor **Pkg2**. Therefore, in this example you can add or delete packages by adding or removing a row while in the previous example you cannot. In this sense, record keyword plays a vital role that determines which model elements can be added or deleted by users.

6.5 Alternative and Group Dimensions

Next, we move on to how to organize tree structures by using the following example model. For the sake of simplicity, we denote model elements with lowercase with numbers (e.g. a1) and its types with uppercase (e.g A) in this example.

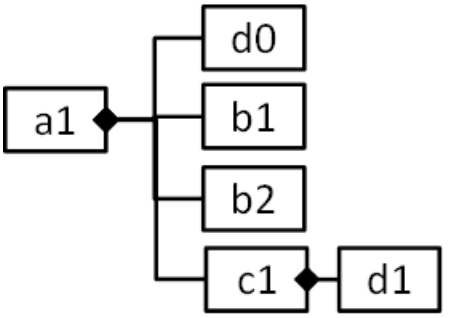


Figure 6.4: Simplified Model Number Two: Using Alternative and Group

Let us consider the following configuration:

```
synctable-schema alternativeExample{
    dim [A] {
        key column /name as Aname
    }
    alternative {
        record dim /nestedElements[B] {
            key column /name as Bname
        }
        record dim /nestedElements[C] {
            key column /name as Cname
        }
        record dim /nestedElements[D] {
            key column /name as Dname
        }
    }
}
```

It generates a tree as show in **Figure 6.5**.

The top level dimension selects type A by `[A]`, and then the root of the tree is a1. In the following dimensions, it selects `/nestedElements[B]`, `/nestedElements[C]`, or `/nestedElements[D]` because these are in `alternative { ... }` clause. That means that if `/nestedElements[B]` is matched, the second dimension is used; if `/nestedElements[C]` is matched, the third dimension is used; and if `/nestedElements[D]` is matched, the forth dimension is used. Therefore, d0, the first model element in the nestedElements feature, is applied to the forth dimension; b1 and b2 are applied to the third dimension; and c1 is applied to the forth dimension. And then, we obtain a tree shown in **Figure 6.5**.

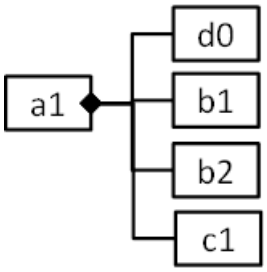


Figure 6.5: Tree From Simplified Model Two

This tree will be turned into a table as shown in **Figure 6.6**. It has four columns consisting of `Aname`, `Bname`, `Cname`, and `Dname`. Since the top level dimension does not have a

`record` keyword, it does not have a record of **a1**. Instead it creates four records for **b1**, **b2**, **c1**, and **d0** corresponding to the tree nodes under "a1" in **Figure 6.5**. Notice that the record for **c1** in the third row fills `Aname` and `Cname` columns, and `Bname` column is specially treated as VOID, which looks blank but filled with thick gray background. Likewise, the second (Bname) and third (Cname) columns in the forth row are also filled with VOID.

a1	b1		
a1	b2		
a1		c1	
a1			d0

Figure 6.6: Table Made From The Tree of Simplified Model Two

Let us move on to the next example using `group` as shown below:

```
synctable-schema groupExample {
    dim [A] {
        key column /name as Aname
    }
    alternative {
        record dim /nestedElements[B] {
            key column /name as Bname
        }
        group {
            dim /nestedElements[C] {
                key column /name as Cname
            }
            record dim /nestedElements[D] {
                key column /name as Dname
            }
        }
    }
}
```

It generates a tree as shown in **Figure 6.7**.

The difference is that now **d1** belongs to **c1** instead of **a1** because the above configuration says B or C followed by D rather than B, C, or D. It means something like (B or (C, D)) in contrast with (B or C or D). That is, group keyword is something like parentheses in dimension definitions and alternative is like the `or` operator.

Then this tree is translated to a table as shown in **Figure 6.8**. Since the dimensions of B and D have a record keyword, it creates three records: **b1**, **b2**, and **d1**, corresponding to the first, second, and third rows. The third and fourth columns that follow after **b1** and **b2**, in the first and second rows, are EMPTY and the second column in the third row is VOID in this table.

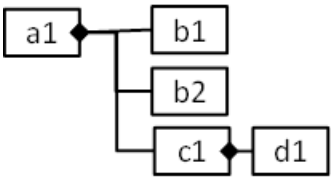


Figure 6.7: Another Tree Made From Simplified Model Two

a1	b1		
a1	b2		
a1		c1	d1

Figure 6.8: Another Table Made From Simplified Model Two

Since group keyword combines dimensions in alternative blocks, using it out of alternative does not give any effects. For example,

```
dim ... {}
dim ... {}
dim ... {}
```

and

```
dim ... {}
group {
    dim ... {}
    dim ... {}
}
```

give the same results.

6.6 ReferenceDecomposition and ReferenceQuery

Mapping reference values with ReferenceDecomposition and ReferenceQuery

`ReferenceDecomposition` is used for presenting references of model elements. The examples so far edit model elements themselves by querying them with QPEs, where we can track references as well. That means we always change values of such model elements instead of references to model elements.

First we specify the formal syntaxes of `ReferenceDecomposition` and `ReferenceQuery` as below:

```
ReferenceQuery      ::= ('key')? 'reference-query' ObjectQueryPath
ReferenceDecomposition ::= 'reference-decomposition' ID '='
                           [ReferrableSyncTable] '{' ForeignColumn* '}'
ForeignColumn       ::= KeyForeignColumn | NonkeyForeignColumn
KeyForeignColumn    ::= 'foreign-key' 'column' [Column] 'as' ID
NonkeyForeignColumn ::= 'foreign' 'column' [Column] 'as' ID
```

In `ReferenceDecomposition`, you should specify all of the key columns in the referred table as `KeyForeignColumn` (that is, you should specify "foreign-key" for such key columns) because we should identify a record by such key columns. If the configuration does not satisfy this condition, it is not guaranteed to identify a unique record to make a reference.

ReferenceDecomposition by Example

We use the following code snippet to explain `ReferenceDecomposition`.

This configuration transforms the target model in **Figure 6.9** into a table as shown in **Figure 6.10**.

This example first introduces the `TypesByName synctable-schema`, which itemizes all of the types as `TypeName`, and `AttsByName` refers to that type by the `type` feature of `REAttribute`. Note that `AttsByName` takes the `tps` argument of `TypesByName`, and in Line 7, the `attributes` synctable takes `types` as an argument and then the `attributes` synctable uses `types` synctable to refer to types by the `ReferenceDecomposition` in Lines 25-28. Let us look into these in the following section.

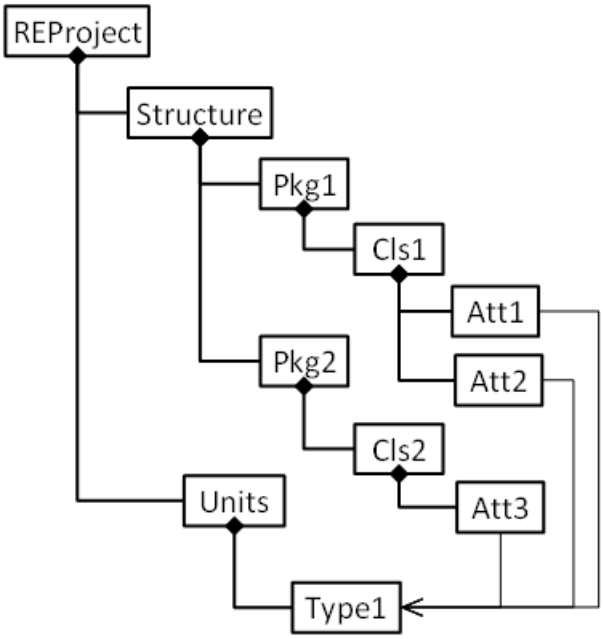


Figure 6.9: Target Model

Figure 6.10: Illustration of ReferenceDecomposition

Adding a Missing Target Element: the create-target Keyword

If the target element exists, then while doing the reference decomposition, MapleMBSE can refer to that element. However, if the target element does not exist, then while doing the reference decomposition you will see an error message indicating that MapleMBSE was unable to resolve the reference. At this point, you would need to create the element in the appropriate schema.

To avoid having to go into the schema to create the element, you can add the `create-target` keyword in the reference decomposition. In this case, while doing the reference decomposition, if MapleMBSE cannot find the element, it will create that element.

The following MSE example illustrates how to use the `create-target` keyword within a reference decomposition.

```
reference-decomposition cls3 = blocks {
create-target
foreign-key column BlockName as Name3
}
```

Note: If you refer to an element in a schema, but while typing make mistake, MapleMBSE will create that element in the model. To correct this, delete the element in the schema or MagicDraw.

References by Dimensions or ReferenceQuery

Next, let us see how we identify references. As show in Line 25 of

(page 40).

, we write `reference-query QPE @ name` in the dimension. Let us look in the part in the example of the previous section:

```
23    dim /nestedElements[REAttribute] {
24         key column /name as AttName
25        reference-query .type @ typedecomp
26        reference-decomposition typedecomp = tps {
27             foreign-key column TypeName as Type
28         }
29    }
```

In this example, we use the `type` feature of `REAttribute` as a reference to be decomposed. Thus, this reference refers to a type identified by the `TypeName` column of `tps` table. This dimension has `AttName` and `Type` columns and `AttName` column is associated with `name` feature of `REAttribute` of this dimension, and `Type` column is used to refer to `type` (see the `reference-query`) by `TypeName` column of `tps` table.

Otherwise, if the reference is associated with a dimension, we put `@ name` after the dimension definition as the example below:

```
dim [REInstance] { … }
dim .otherClass[REClass] @ cls2 {
    reference-decomposition cls2 = clsTbl {
    foreign-key column PkgName as PkgName2
    foreign-key column ClsName as ClsName2
    }
    column /description as ClsDesc2
}
```

where we use `cls2` as the name of the reference. And in the following `reference-decomposition cls2`, we use `PkgName` and `ClsName` columns of `clsTbl` to present that reference. Therefore, this dimension has `otherClass` reference of `REInstance` (in the previous dimension), which refers to `REClass` class identified by `PkgName` (propagated by `PkgName2` column of this dimension) and `ClsName` (propagated by `ClsName2` column, likewise) columns of `clsTbl`. Note that `clsTbl` is a parameter of the `synctable-schema`. Since `PkgName` and `ClsName` are key columns, we specify the `foreign-key` keyword in the reference decomposition. In addition, we can edit `description` of the reference via `ClsDesc2`.

6.7 Key Columns Defined in SyncTable Schema

In a `synctable-schema`, we need to specify key columns to identify the recode by such key columns. So in every dimension, we need at least one key column and all of the model elements associated with this dimension must be uniquely identified by the defined key columns. Key columns in dimensions are one of the followings:

1. Columns defined by **`key`** `column`

2. All of `foreign-key` columns in the ReferenceDecomposition that uses references by **`key`** `reference-query` or dimensions. If you use `reference-query` without the `key` keyword, such `foreign-key` columns are not key columns.

6.8 Using Default Value Generation in a Column

After you assign a name to Attribute Column use '=' and then enter the type of value you want to generate, as in the example below. Here the attribute column has been assigned the name PropertyName, followed by '=', then the text value and Sequence number .

```
synctable-schema BlockSchema {
    record dim [Package] {
        key column /name as packageName
        column  /visibility as pkgVis = "private"
    }

    record dim /packagedElement[Class|mse::metaclassName="SysML::Blocks::Block"] {
        key column /name as blockName = %UUID
    }

    dim /ownedAttribute[Property]{
        key column /name as propertyName = "R" %SEQID
    }
}
```

Limitations

Default value generation column doesn't work in the following cases:

- In root dimension key column
- In alternative, group, optional key columns
- used is reference decomposition any column

Data Insertion Order with the Default Column

If you have default value generation in the sync scheme then order to enter information into the cells is important here let's see the example We have a scheme in which record root dimension is package and visibility, visibility has default value next we have record dimension for Class which has default value generation after that we have record dimension for the property.

```
synctable-schema BlockSchema {
        record dim[Package]{
        key column /name as packageName
                column   /visibility as pkgVis="private"
        }
       record dim /packagedElement[Class\mse::metaclassName="SysML::Blocks::Block"]
       {
            key column /name as blockName=%UUID
            }
            record dim /ownedAttribute[Property]{
                 key column /name as propertyName = "R" %SEQID
            }
  }
```

Example: Create a Package that has Class and Property

This example illustrates the importance of the order of entry of data with default value generation.

1. Enter the Package name. From the table below you can see that the Visibility has been automatically created.
2. Enter the same Package name again. Notice the Class column has been automatically filled.
3. Next, try to create the property by entering the package name. Notice that MapleMBSE creates a new Class. If you repeat this, MapleMBSE will create a another new Class.
4. This time, enter the Class name first.
5. Next, enter the Package name. Notice that a property has been created.

Package	Visibility	Class	Property
Package	private		
Package	private	beaaefaf-e74b-41c7-8502-b72989b72f2b	
Package	private	2e664310-7a99-4df7-9ddb-f9c7559e4048	
		2e664310-7a99-4df7-9ddb-f9c7559e4048	
Package	private	2e664310-7a99-4df7-9ddb-f9c7559e4048	R1

Example: Create a Package and Class with Alternative Dimensions

In this example, a second dimension has default values but not a record dimension. After that, there are alternative two dimensions.

```
synctable-schema B2Schema {
   record dim [Package] {
       key column /name as packageName
   }

   dim /packagedElement[Class] {
       key column /name as blockName =%UUID
   }
   alternative{
       dim /ownedAttribute[Port]{
       key column /name as portName
       column  /visibility as pVis
       }

       dim /ownedAttribute[Property]{
           key column /name as propertyName
           column  /visibility as proVis
       }
   }
}
```

1. Enter the Package name. The Class and other fields are grayed-out so the package has been created.
2. Enter the Package name again. Notice the Class was not created, as shown in the image below.

Package	Class	port	visibility	Property	Visibility
Package					
Package					

3. To create the Class and visibility, you have to enter the Property or port name. After this, the Class will be created as shown below.

Package	Class	port	visibility	Property	Visibility
Package					
Package	7e53a36b-63ba-4c68-8b72-cbe4f0c19d4d	port	public		

7 SyncTable

A `SyncTable` is an intermediate structure created in the first step of converting model data into a table from shown in an Excel spreadsheet. The definition of a `SyncTable` consists in applying a SyncTable schema to a Data Source. The formal syntax of a `SyncTable` definition is as follows.

```
SyncTable ::= 'synctable' ID '=' SyncTableSchemaId '<' DataSourceId
              '>' ( '(' SyncTableId (',' SyncTableId)* ')' )?
```

In the table *'synctable' ID '=' SyncTableSchemaId '<' DataSourceId '>' ('(' SyncTableId (',' SyncTableId)* ')')? (page 47)*.

`SyncTableSchemaId` is an ID of a SyncTableSchema.

- DataSourceId is an ID of a DataSource.
- SyncTableId is an ID of a SyncTable.

8 Laying out SyncViews

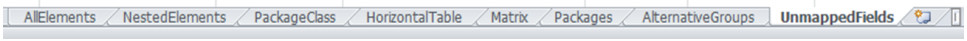
This chapter describes how SyncTables are presented as SyncViews. All of the SyncViews must be laid out in some worksheet in a workbook. The rest of the sections are organized as 1) how to set up worksheets in a workbook; 2) how to lay out SyncTables in a table; and how to lay out SyncTables in a matrix.

8.1 Setting up a Workbook and Worksheets

When MapleMBSE opens a model, it assigns one workbook to the model, and in *Workbook Instance*, we specify all of the worksheets managed by MapleMBSE. In each configuration, one and only one Workbook instance must be specified.

The example below comes from **Example/UserGuide.MSE**, and it defines all of the worksheets.

```
workbook {
    worksheet AllElements(allElementsTable)
    worksheet PackageClassProperty(classesInPackage) {label = "NestedElements"}
    worksheet PackageClass(classesTable)
    worksheet Horizontal(classesInPackage) {label = "HorizontalTable"}
    worksheet Dependencies(dependenciesTable,dependentTable,supplierTable) {label = "Matrix"}
    worksheet DependenciesSheet(dependenciesTable)
    worksheet Packages(packagesTable)
    worksheet NestedPackageClass(nestedPackageClassTable) {label = "AlternativeGroups"}
    worksheet UnmappedFields(nestedPackageClassTable)
}
```



In the example shown "workbook' is used to represent the arrangement of worksheets as shown, AllElements is the name of the worksheet template and allElementsTable is the name of the synctable that is created. By default, a worksheet is created with the name AllElements containing the information from the corresponding worksheet-template. MapleMBSE allows the user to create a name for the worksheet manually by using the 'label' attributes as shown in the above example.
Note: When a worksheet template is created with more than one parameter they should be separated with ',' as shown above for the creating a *Dependencies* worksheet.

If `lazy-load` is specified before `worksheet`, MapleMBSE will load syncviews of that worksheet when that worksheet is activated. By default, thay are loaded at the startup and then the name of `worksheet-template` and its parameters follow. This means the worksheet will be defined by the specified `worksheet-template`. The details of `worksheet-template` are explained in the next section.

If the worksheet declaration has label="XXX", MapleMBSE regards "XXX" as the name of the worksheet. Otherwise, the name, `worksheet-template` is used as the name of the worksheet. For example, `AllElements` will be the name of the worksheet defined by

`worksheet AllElements(allElementsTable)`. If the Excel template has the worksheet with the same name, MapleMBSE will use this worksheet to initialize the syncviews of `worksheet-template`. Otherwise, MapleMBSE will create a new worksheet with the same name.

8.2 Worksheet Template and View Layout

A worksheet template is used to define how a SyncTable should be represented in the Excel worksheet. In a worksheet template, we can specify one or more `ViewLayouts`, each of which can be a table view layout or a matrix view layout, in the following subsections respectively.

The formal syntax of worksheet template is as follows.

```
WorksheetTemplate   ::=   'worksheet-template' ID '(' WorksheetTemplateParam
                           (',' WorksheetTemplateParam)* ')'
                          '{' ViewLayout* '}'
ViewLayout          ::=   TableViewLayout | MatrixViewLayout
```

where ID means the name of the worksheet template and should be referred by worksheet definitions explained in the previous section.

Important: Do not use an *All Primary Data Source (page 32)* directly in the workbook as `WorksheetTemplateParam;`

Table View Layout

Table view layout allows the user to define how the contents of the model should be displayed in the table. It has two possible arrangements: vertical or horizontal. The syntax for table view layout is as show above. To define a table layout: you must specify the arrangement of the table, the cell address to define the location of table in Excel, and the order of the fields. A column can be populated with either mapped or unmapped fields; a mapped field displays the attributes or value assigned to it whereas an unmapped field is used to insert a blank column within the table. Based on how fields are declared in the synctable schema as key column or column inside the view layout they are declared as key field or field respectively. It is necessary to provide the column type as string or integer for every field that is created except for the unmapped field.

The formal syntax of table view layouts is as follows.

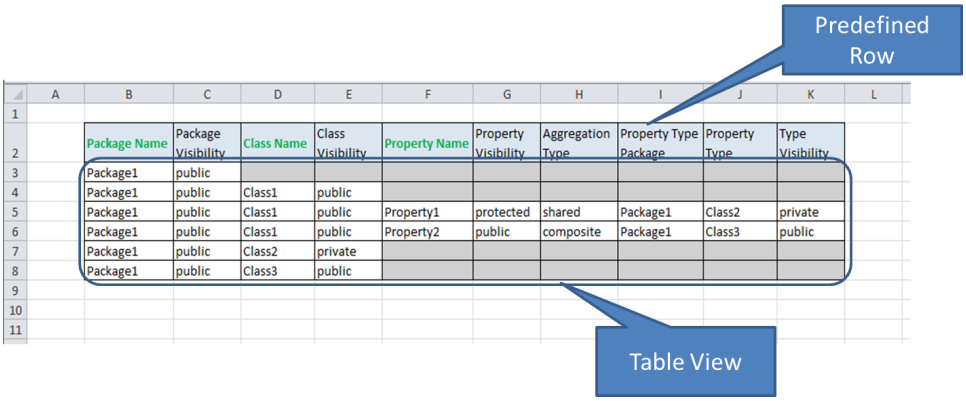
```
TableViewLayout   ::= ('vertical' | 'horizontal') 'table' ID at'
                      CellAddr
                      '=' WorksheetTemplateParam '{'
                      ( 'import-order' INT )? & ( 'enable-import'
```

```
                        BoolType )?
                        ViewColumn*( SortKeys )? '}'
ViewColumn          ::= MappedViewColumn | UnmappedViewColumn
MappedViewColumn    ::= KeyViewColumn | NonkeyViewColumn
KeyViewColumn       ::= 'key' 'ref'? 'field' [KeyColumn] ':'
                        ViewColumnType


NonkeyViewColumn    ::= 'ref'? 'field' [Column] ':' ViewColumnType


UnmappedViewColumn ::= 'unmapped-field'
ViewColumnType      :   ('String' | 'Integer' | 'Double')'[]'? (';'
                        'delimiter' '=')? & (';' 'quote' '=' STRING)?
```

The following code snippet comes from **UserGuide.MSE** which defines a table view layout.

```
worksheet-template AllElements(cls:AllElementsTable) {
    vertical table tab1 at (3,2) = cls {
        key field PackageName : String
        field PackageVisibility : String
        key field TopClassName : String
        field TopVisibility : String
        key field PropertyName : String
        field PropertyVisibility : String
        field AggType : String
        field PropertyTypePackage : String
        field PropertyType : String
        field PropertyTypeVisibility : String
        sort-keys PackageName, TopClassName, PropertyName
    }
}
```

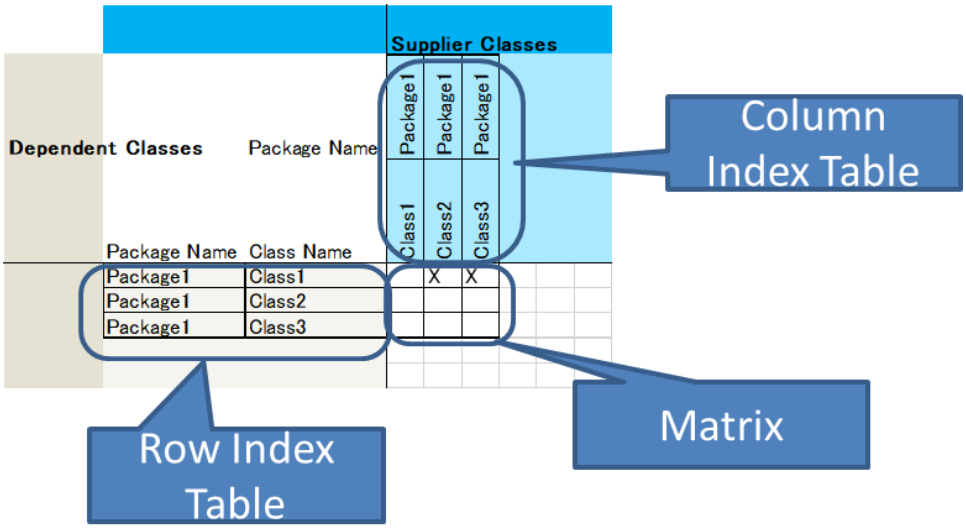
In the example shown above, a worksheet template with ID `AllElements` is created for a `synctable-schema`, `AllElementsTable` that is assigned to a parameter `cls`. Line 2 defines that the table is arranged vertically and `(3, 2)` means it should be displayed from Row 3 and Column B in Excel as show in the figure below. Line 3 to line 12 in the example defines the order in which fields have to be displayed in Excel, shown as Table View in the figure. In line 3 in the sample, `key field` is used for `PackageName` because it was specified as `key column` in the synctable schema for `AllElementsTable`. Column type for every field is provided as shown from line 3 to line 12, in the example shown `String` is the type for all fields. To specify type integer, use `Int` instead of `String`. `sort-keys` are used to indicate the columns that should be sorted in ascending order when model data is loaded or when new data is added to the table.
Predefined Row in the figure below denotes that the Excel sheet can be formatted based on user preference before the model is loaded in the Excel sheet.



Package Name	Package Visibility	Class Name	Class Visibility	Property Name	Property Visibility	Aggregation Type	Property Type Package	Property Type	Type Visibility
Package1	public								
Package1	public	Class1	public						
Package1	public	Class1	public	Property1	protected	shared	Package1	Class2	private
Package1	public	Class1	public	Property2	public	composite	Package1	Class3	public
Package1	public	Class2	private						
Package1	public	Class3	public						

Matrix View Layout

A matrix view layout consists of three parts, row index table view part, column index table view part, and matrix part, as shown in the following example:



The formal syntax of matrix view layout is as follows.

```
MatrixViewLayout      ::=  'matrix' ID 'at' CellAddr
                           '=' [WorksheetTemplateParam] '{'
                           ( 'import-order' INT )? &

                           ( 'enable-import' BoolType )?
                           ViewColumn
```

```
                              MatrixRowIndexViewLayout &
                              MatrixColumnIndexViewLayout

                              '}'


MatrixRowIndexViewLayout ::=  'row-index' '=' [WorksheetTemplateParam]
                              '{' ViewColumn* ( sortKeys=SortKeys )?
                              '}'
MatrixColumnIndexViewLayout ::= 'column-index' '='
                              [WorksheetTemplateParam]
                              '{' ViewColumn* ( sortKeys=SortKeys )?

                              '}'
```

Row and column index tables identify which cell in a matrix should be selected to show a record of the synctable. The following code snippet comes from **UserGuide.MSE** which defined matrix view layout.

```
1  worksheet-template Dependencies(mat:DependenciesTable,tabr:DependentTable,tabc:SupplierTable) {
2      matrix Matrix1 at (5,4) = mat {
3          const-field "X"
4          row-index = tabr {
5              key field DependentPackageName : String
6              key field DependentClassName : String
7
8              sort-keys DependentPackageName, DependentClassName
9          }
10         column-index = tabc {
11             key field SupplierPackageName : String
12             key field SupplierClassName : String
13
14             sort-keys SupplierPackageName, SupplierClassName
15         }
16     }
17 }
```

In this example, Lines 2 to 16 defines matrix layout with the name of `Matrix1` created by `mat`, that is `DependenciesTable` (see parameters of Dependencies). `const-field` means a matrix cell should be filled with the specified value if and only if the corresponding record exists. You can specify some column instead of `const-field`. For example, if you specify `field DepName : String`, you can edit `DepName` in matrix cells. However, you can specify only one field for a matrix view layout.

Lines 4 to 9 define a row index table, and Lines 10 to 15 define a column index table. Using this configuration, `DependentTable` (row index table) needs to have `DependentPackageName` and `DependentClassName` columns, and `SupplierTable` (column index table) needs to have `SupplierPackageName` and `SupplierClassName` columns, and finally, `DependenciesTable` (matrix table) needs to have all of the columns, those are `DependentPackageName`, `DependentClassName`, `SupplifierPackage-`

`Name`, and `SupplierClassName`. In **DependenciesSheet**, we can show `DependenciesTable` in a vertical table as below:

Dependent Classes		Supplier Classes	
Package Name	Class Name	Package Name	Class Name
Package1	Class1	Package1	Class2
Package1	Class1	Package1	Class3

A synctable used to present in a Matrix **must have** all of the same key columns of row and column index tables. In this example, `DependenciesTable` (synctable to be shown in a matrix) must have `DependentPackageName` and `DependentClassName` that are the key columns of row index table (`DependentTable`), and `SupplierPackageName` and `SupplierClassName` that are key columns of column index table (`SupplierTable`). Notice that all of the column names must be unique.