

VDMTools

The Rose-VDM++ Link
ver.1.2



How to contact:

<http://fmvdm.org/>

<http://fmvdm.org/tools/vdmttools>

inq@fmvdm.org

VDM information web site(in Japanese)

VDMTools web site(in Japanese)

Mail

The Rose-VDM++ Link 1.2

— Revised for VDMTools v9.0.6

© COPYRIGHT 2016 by Kyushu University

The software described in this document is furnished under a license agreement.
The software may be used or copied only under the terms of the license agreement.

This document is subject to change without notice

Contents

1	Introduction	1
2	The Rose-VDM++ Link	3
2.1	The Architecture of The Rose-VDM++ Link	4
2.2	The Main Rose-VDM++ Link Window	5
2.3	Translating VDM++ to UML	7
2.4	Translating UML to VDM++	10
2.5	Merging VDM++ and UML Models	14
2.6	Type Checking of Specifications	18
3	Rational Rose	20
3.1	Class Diagrams in Rational Rose	20
3.2	Manipulating Definitions in Rational Rose	23
4	Mapping Rules between VDM++ and UML	27
4.1	The Class Structure	27
4.2	Associations between Classes	29
A	Summarising the Mapping Rules	35
B	Warnings generated by the Rose-VDM++ Link	37

1 Introduction

Object-oriented analysis and design is a development method widely used in the software engineering community. The *Unified Modelling Language (UML)* is a standard graphical language for expressing and communicating object-oriented designs. UML was developed jointly by Grady Booch, Ivar Jacobson, and Jim Rumbaugh at Rational Software Corporation, with contributions from other leading methodologists, software vendors, and many users. Being a merge of the Booch, OMT, and Jacobson notations, the UML provides ways for business process, object, and component modelling. Currently, several commercial CASE tools support UML, one of which is Rose 2000, the successor to Rose 98 from Rational Software Corporation. Throughout the rest of this document we refer to the two collectively as Rose 98/2000.

Though well suited for the overall object-oriented design, UML is not suited for giving formal semantics to the model. VDM++, on the other hand, is designed for formal specification of object-oriented systems with concurrent and real-time behaviour. The language is based on ISO VDM-SL, and has been extended with class and object concepts, facilitating the development of object-oriented formal specifications.

The *Rose-VDM++ Link* combines the two languages. By defining and implementing mapping rules between the two languages, the Rose-VDM++ Link allows the user to translate (parts of) a model represented in VDM++ to UML, and vice versa. The Rose-VDM++ Link supports round-trip engineering, allowing the user to start modelling the overall object-oriented aspects of the system in UML, and proceed by giving formal semantics to parts of the model by translating it to VDM++. Subsequently the VDM++ specification can be mapped, or merged, back into UML, either for documentation purposes or for further modelling of the object-oriented aspects of the model. This mapping back and forth between the two representations can continue until the model eventually is completed.

The Rose-VDM++ Link - an Add-In to Rose 98/2000

The Rose-VDM++ Link is installed as an Add-In to Rose 98/2000, and it can be activated or deactivated through the Add-In Manager of Rose 98/2000. For details on how to install the Rose-VDM++ Link see [1].

Using This Manual

This document is an extension to the *VDMTools User Manual (VDM++)* [4].

Before continuing reading this document it is recommended to read the Toolbox manual. Moreover, knowledge about *UML* [5] and Rose 98/2000 [6] is also preferable.

This manual is structured in the following way:

Section 2 describes the different features of the Rose-VDM++ Link, and Section 3 presents some important features of Rose 98/2000. To fully understand how the Rose-VDM++ Link translates from one representation to the other, it is important to understand the mapping rules applied. These rules are described in Section 4. These rules are the theoretical foundations of the Rose-VDM++ Link. You can use the Rose-VDM++ Link without reading this section, but in many cases knowing the applied transformation rules, will make the use of the tool easier. The transformation rules are summarised in Figure 23 in appendix A.

The presentation of the capabilities of the Rose-VDM++ Link is, whenever necessary, accompanied with user scenarios and various screen dumps. Furthermore some of the features of Rose 98/2000 are described when needed.

With the distribution of the Toolbox a specification of different sorting algorithms is included. It will be used in the following to describe the use of the Rose-VDM++ Link. The example is described in [3]

2 The Rose-VDM++ Link

This section presents the various services offered by the Rose-VDM++ Link.

The Rose-VDM++ Link interfaces the CASE tool, Rose 98/2000, to construct and display classes in UML, and to let you edit the model at UML level and map the modified model to VDM++.

The use of the tool can be divided into three categories:

Mapping from UML to VDM++ (*Forward Engineering*): Use the Rose-VDM++ Link to generate a VDM++ specification from the classes defined in UML.

Mapping from VDM++ to UML (*Reverse Engineering*): Will create a UML model from an existing VDM++ specification.

Synchronising UML and VDM++ models: During system development it is very likely that the VDM++ model and the UML model of the system will be modified simultaneously. The Rose-VDM++ Link allows you to keep track of the changes made in each model, and to synchronise the two models by merging them into one and subsequently propagate the merged model to UML and VDM++.

The basis for the Rose-VDM++ Link is the mapping rules described in Section 4. They state exactly how different constructs are mapped between VDM++ and UML.

Moreover, Section 4 describes, which parts of VDM++ and UML are not included in the mapping rules. For instance, the bodies of operations and functions are not mapped to UML when reverse engineering VDM++ specifications. Only the signatures of functions and operations are translated to UML. This does not, however, imply that the function and operation bodies are lost — they are simply not visible in the UML model. If the UML model is later translated to VDM++ the bodies of functions and operations will be preserved.

In the same way the Rose-VDM++ Link will not alter parts of the UML model that are not included in the mapping rules. For instance, the UML model can be extended with use cases, deployment diagrams, state transition diagrams, etc. Such parts will simply be left unchanged by the Rose-VDM++ Link.

To fully understand the possibilities and limitations of the Rose-VDM++ Link, some knowledge of how the VDM++ Toolbox and Rose 98/2000 are connected,

as well as the flow of information between the two tools is useful. Section 2.1 presents such information. Section 2.2 presents the graphical user interface of the Rose-VDM++ Link. In Sections 2.3-2.5 the functionality of the Rose-VDM++ Link is described in detail and the three different ways of translating between VDM++ and UML are presented.

2.1 The Architecture of The Rose-VDM++ Link

The architecture of the Rose-VDM++ Link is illustrated in Figure 1. The figure illustrates how the VDM++ Toolbox and Rose 98/2000 are connected as well as the flow of data between the tools.

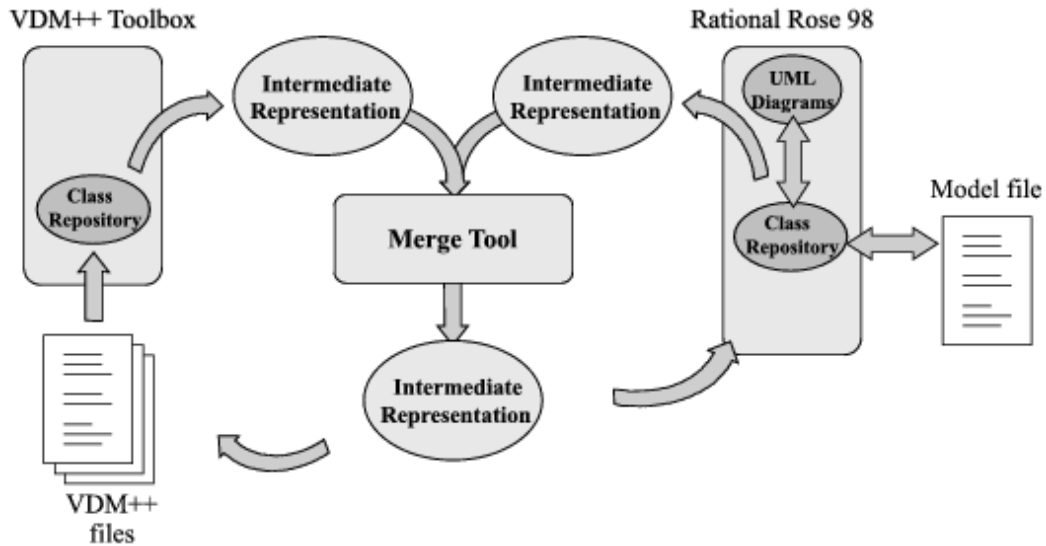


Figure 1: The architecture of the Rose-VDM++ Link.

Starting at the lower left corner of the figure, it shows how a set of VDM++ files (in `rtf` format) are parsed and added to the class repository of the VDM++ Toolbox. The Rose-VDM++ Link translates the VDM++ specification into an intermediate representation capturing the parts of VDM++ relevant in UML.

In the same way, by accessing the class repository of Rose 98/2000, the UML model is translated to the intermediate representation.

The two intermediate representations are “compatible” and can thus be compared or merged. Merging the two models into one common model, and subsequently


propagate this model to the class repository in Rose 98/2000 and the VDM++ specification files, will synchronise the VDM++ and UML model.

The two representations can be merged in several different ways resulting in the three ways of using the Rose-VDM++ Link, described above.

If there is no intermediate representation resulting from Rose 98/2000 or if one chooses to ignore such a representation, the merge will result in a full translation from VDM++ to UML. Parallely, if there is no intermediate representation resulting from the VDM++ Toolbox or if one chooses to ignore such a representation, the result is a full translation from UML to VDM++. Finally, one can choose to merge the two representations.

2.2 The Main Rose-VDM++ Link Window

This section describes the graphical user interface of the Rose-VDM++ Link.

The Rose-VDM++ Link is invoked from within the VDM++ Toolbox by selecting the menu item *Tools/Rose-VDM++ Link* or by pressing the  (Rose) button.

This is exactly the step shown in Figure 2.

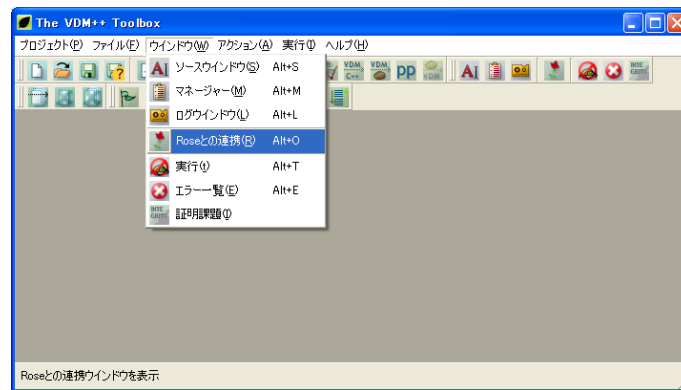


Figure 2: Invoking the Rose-VDM++ Link

Invoking the Rose-VDM++ Link opens a separate window as illustrated in Figure 3. While activated, all the features of the Rose-VDM++ Link can be accessed from this window. The meaning of the different buttons will be described in the following sections.

When the Rose-VDM++ Link is invoked, the necessary connection to Rose

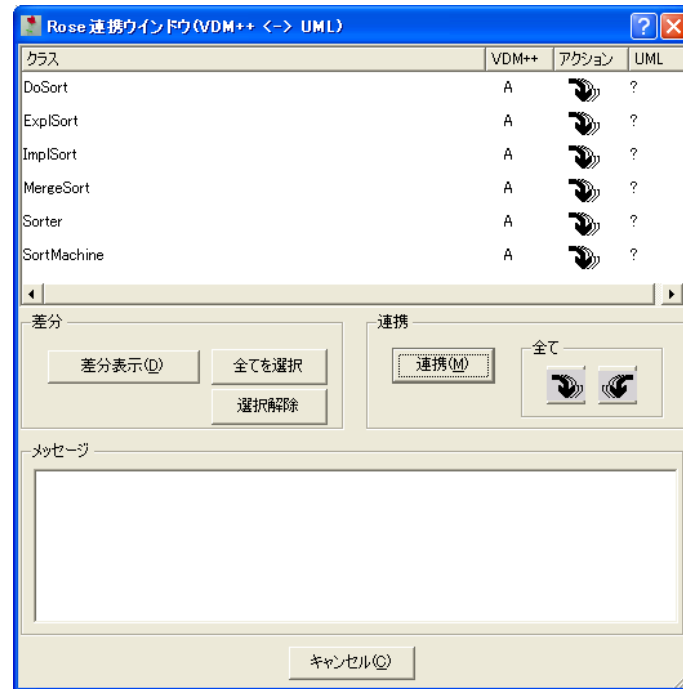


Figure 3: The main window of the Rose-VDM++ Link.

98/2000 is established. If an instance of Rose 98/2000 is already running on the machine, the VDM++ Toolbox connects to this instance. Otherwise Rose 98/2000 is automatically started and a connection is established to this instance. Subsequently the Rose-VDM++ Link will read in a Rose Model File following these simple rules:

- If the current project in the VDM++ Toolbox is given a name (i.e. an existing project was opened or a new project was created and saved), the Rose-VDM++ Link will try to open a model file with the same name and locataion as the project file. For example, for a project file `MyProject.prj` the Rose-VDM++ Link will open the model file `MyProject.mdl` located in the same directory as the project file. If a model file with this name does not exist, an empty model file will be created.
- If the current project in the VDM++ Toolbox was never saved, the Rose-VDM++ Link will simply use the model currently loaded into Rose 98/2000.

When the Rose-VDM++ Link is started the intermediate representations of the VDM++ and UML models are automatically computed, based on the classes

contained in the VDM++ Toolbox and the classes contained in Rose 98/2000. The contents of the two representations are presented to the user as a list of class names, with the state of each class in VDM++ and UML identified by designated status symbols. The exact meaning of these symbols will be described in more detail later in this section.

In the following sections we will describe how to translate one representation to the other as well as how to merge two representations using the Rose-VDM++ Link. To illustrate the capabilities of the tool we will use the Sort Example of [4].

2.3 Translating VDM++ to UML

If one or more classes have been syntax checked before the Rose-VDM++ Link is started, these classes are translated into the intermediate representation using the mapping rules described in Section 4. Assume now, that the VDM++ Toolbox has been configured for the files of the Sort Example and that a project file with name `Sortpp.prj` has been created. Assume furthermore, that the VDM++ files have been successfully syntax checked and that Rose 98/2000 does not contain any class definitions. Invoking the Rose-VDM++ Link would then display a class list as shown in Figure 4.


DoSort	A		?
ExplSort	A		?
ImplSort	A		?
MergeSort	A		?
Sorter	A		?
SortMachine	A		?

Figure 4: The class list for the Sort example

The class designators used here are **A** and **?** with the following meaning:

A Indicates that the class was added since the model was last computed.

? Indicates that the class is not known in this model.

Consequently the class list of Figure 4 should be read as follows: All six classes have just been added to the VDM++ model. On the other hand none of the

classes defined in VDM++ are known in the UML model, simply because we assumed that Rose 98/2000 did not contain any class definitions prior to invoking the Rose-VDM++ Link.

Each class in the list has its own action button used to configure how the merge of the two representations should take place. When the Rose-VDM++ Link window is opened each button will be assigned a default action, based on the two class designators of the associated class. To change the state of an action button you simply click it. Each click will toggle the state of the button between at most four different states:



VDM++ to UML: This action will map the definitions of the class defined in VDM++ to a class of the same name in UML. If this class exists in UML already it is updated. Otherwise a new class will be created.



UML to VDM++: This action will take the definitions of the class defined in UML and map them to a class of the same name in VDM++. Consequently this will create a new class or update the definitions of an already existing VDM++ class with the same name. See Section 2.4.



Merge: Selecting this action for a class will take the definitions of the class defined in VDM++ and merge them with the definitions of the class defined in UML. See Section 2.5.



Exclude: This will exclude the class from the new model being computed. As a consequence the class will be removed from both the UML and VDM++ model.

In the example of Figure 4 all action buttons are set to map from VDM++ to UML by default. Clicking one of the action buttons will reveal that it is only possible to toggle between two states, namely the “VDM++ to UML” and the “Exclude” state. The reason for this is that the other two possible states make no sense since no classes are defined in Rose 98/2000. If the state of the action buttons have been changed, the default settings can be restored any time by clicking the “Default” button.

And now back to the example: To make the Rose-VDM++ Link perform the translation from VDM++ to UML, click the button labelled “Map”. The Rose-VDM++ Link will now add six new classes to the class repository of Rose 98/2000. All new classes generated by the Rose-VDM++ Link are added to a package named “Generated classes”. At any time, however, a class may be

moved from the “Generated classes” package into another package. If this class is later on being updated/modified by the Rose-VDM++ Link it will not be relocated to the “Generated classes” package.

Figure 5 shows a snapshot of the class repository after having translated the VDM++ classes of the Sort Example to UML.

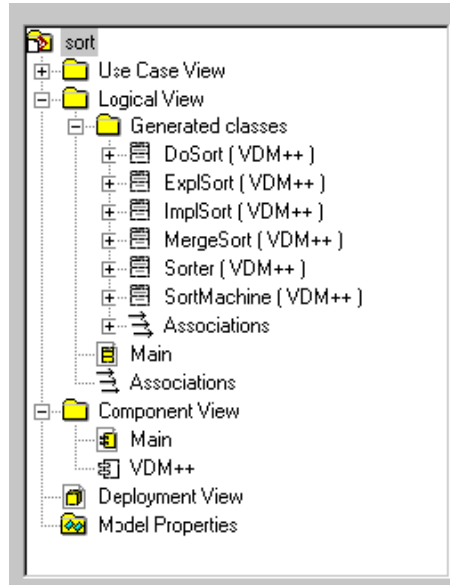


Figure 5: The class repository of Rose 98/2000 for the Sort example

As an example, let us look at the UML class generated for the VDM++ class **Sorter**. Figure 6 shows the VDM++ specification of the Sorter class and the generated UML class. As you can see, the translation is very straightforward for this class. For more information of the used mapping rules see Section 4.

In Rose 98/2000 you can browse the class repository and inspect or modify the definitions of attributes and operations. Moreover you can create class diagrams from the classes in the class repository. In Section 3 we will give a short introduction to Rose 98/2000.

Generated Files

Before updating the UML model of Rose 98/2000 the Rose-VDM++ Link will create a backup of the current UML model. The backup will be named by adding `_old` to the name of the current model of Rose 98/2000. For example the model

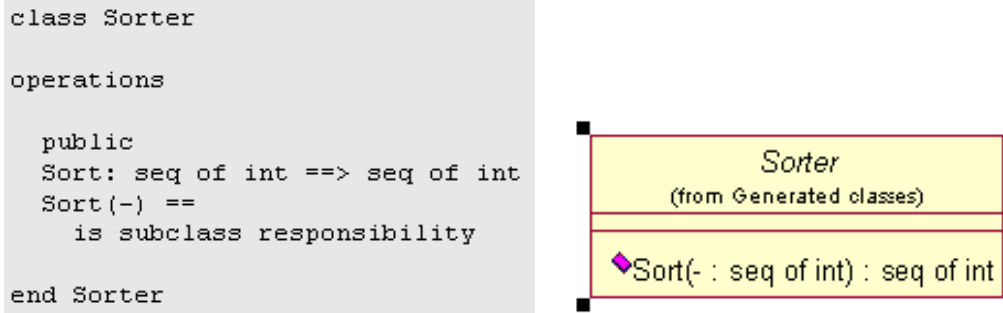


Figure 6: The VDM++ **Sorter** class (left) and the UML **Sorter** class (right)

file `MyProject.mdl` will be backed up in `MyProject_old.mdl`. If the mapping from VDM++ to UML presents unexpected results, the changes made by the Rose-VDM++ Link can easily be undone from this backup.

2.4 Translating UML to VDM++

In this section we proceed by describing how to use the Rose-VDM++ Link to generate VDM++ from a UML model defined in Rose 98/2000. To keep things as simple as possible we will generate VDM++ from the model generated in Section 2.3, and assume that the VDM++ Toolbox contains no model. I.e. we assume that there are no syntax checked classes prior to the invocation of the Rose-VDM++ Link. By selecting the menu item *Project/New* of the VDM++ Toolbox all previously loaded classes from the VDM++ Toolbox will be removed.

Now, by selecting the menu item *Tools/Rose-VDM++ Link*, the class list shown in Figure 7 will be displayed. As expected, we see that six classes are defined in UML whereas none of these classes are known in VDM++.

Clicking the “Map” button now will display the dialog shown in Figure 8. You can select the directory to save the generated VDM++ class files for the six classes defined in UML.

As an example, Figure 9 shows the UML version of the class **Exp1Sort** and the VDM++ specification generated for the same class. Notice that the functions and operations are generated simply as signatures - the user must specify the bodies of them later on.

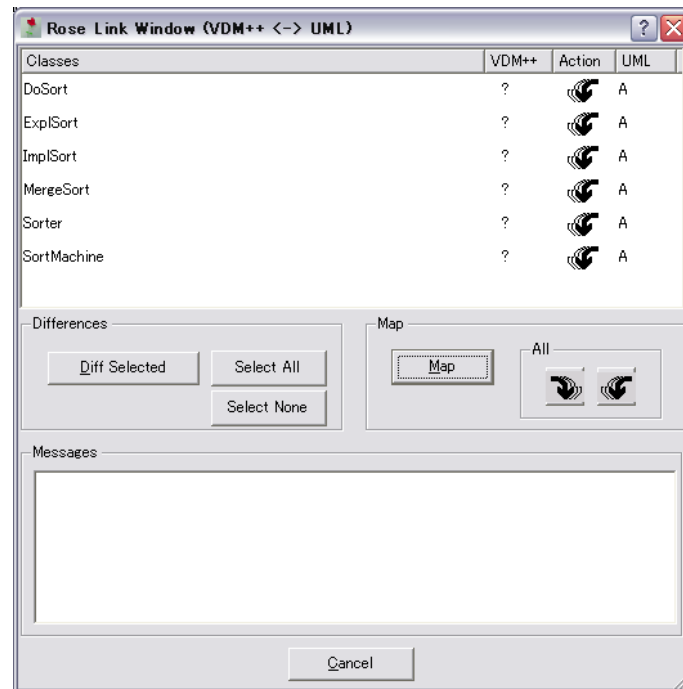


Figure 7: The class list for only UML classes



Figure 8: The dialog to select directory for the generated VDM++ classes

Generated Files

Each new class introduced in UML will be generated in a separate `rtf` file with the same name as the class. For example the class `MyClass` will be generated to a

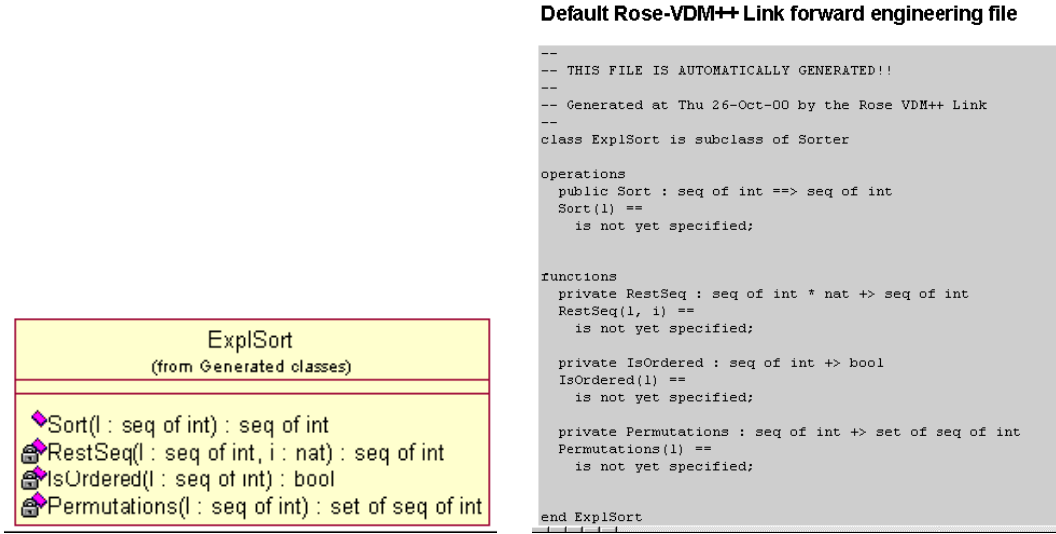


Figure 9: The UML ExplSort class (left) and the VDM++ ExplSort class (right)

file with the name `MyClass.rtf` located in the same directory as the project file. If a file with this name exists already, the Rose-VDM++ Link issues a warning and the new file will *not* be created. In this case the existing file should be moved or renamed, or the class introduced in UML should be given another name to avoid the name conflict.

The log window of the VDM++ Toolbox will display the names of all files generated, as well as the directory in which they were generated.

When generating new `rtf` files for classes introduced in UML the Rose-VDM++ Link uses the special file named `NewClass.rtf` as “skeleton” for the new file. This file *must* be located in the `uml` subdirectory of the VDM++ Toolbox installation. The contents of this skeleton file may be changed, but it is important that the following requirements are met:

- The document should *always* contain a VDM block, i.e. at least one (possibly empty) paragraph in style VDM.
- The filename must be `NewClass.rtf` and the file must be located in the `uml` subdirectory of the VDM++ Toolbox installation.

Using the Generated Classes

The generated VDM++ classes are automatically added to the current project and parsed. This is possible, because the Rose-VDM++ Link generates classes which almost always will be syntax correct. However, it is possible that syntax errors from the Rose model will be carried over to the VDM++ model (normally inappropriate use of keywords).

You will now probably proceed by extending the VDM++ specification, for example by writing the bodies of operations and functions. One could also imagine, that you want to modify the object-oriented aspects of the model (inheritance and association relations), or you find it necessary to add new instance variables, functions, etc.

This should be done by simply editing the generated `rtf` files. Later, the new VDM++ model can easily be reverse engineered to UML in order to make the two models consistent. Syntax check the classes (and make a type check if you like) and proceed as described in Section 2.3, provided that the UML model was not changed during the modification of the VDM++ model.

Warnings Generated during the Translation from UML to VDM++

When a UML class is translated to VDM++, the UML attributes and operations are converted to VDM++ constructs. The names and types used in the UML model must consequently comply with the syntactical rules of VDM++. If this is not the case, the UML definition is simply ignored and the user is notified by a warning. This is done in order to maximise the syntactical correctness of the VDM++ specification generated.

It is important to give notice to the warnings generated when the Rose-VDM++ Link reads definitions from UML. The reason is that whenever a warning is presented, something in the UML model could not be translated to VDM++, and is consequently ignored in the common representation being constructed. As a consequence of the architecture of the Rose-VDM++ Link (see Section 2.1) the representation computed by the Merge Tool will replace the current model in the class repository of Rose 98/2000. This means that constructs ignored while reading the UML model are removed if the “Map” button is clicked!

See appendix B for more information.

2.5 Merging VDM++ and UML Models

In Sections 2.3 and 2.4 we assumed that the model was expressed in either VDM++ or UML, i.e. one of the two representations of Figure 1 was empty. This was to keep matters as simple as possible, and to illustrate how it is possible to take a model from one representation and translate it to the other. This can be useful either when documenting an existing VDM++ specification by generating UML diagrams, or when generating VDM++ from an existing UML model. However, when modelling larger and more complex systems, it is likely that the UML and VDM++ models will evolve simultaneously, and the assumption of Sections 2.3 and 2.4 do no longer hold. To meet this requirement the Rose-VDM++ Link facilitates the merging of two different models into one, as will be described in this section.

In the following we will assume that initially the VDM++ Toolbox has been configured with the Sort Example of Section 2.3, and that this specification has been translated to UML as described in Section 2.3. Subsequently, both the VDM++ specification and the UML model have been slightly modified.

Figure 10 shows the class list of the Rose-VDM++ Link after some changes have been made to both the VDM++ and the UML model. From this figure we can see

DoSort	-		-
ExplSort	-		M
ImplSort	D		-
MergeSort	M		D
QuickSort	?		A
SortMachine	-		-
Sorter	-		-

Figure 10: Class list when both UML and VDM++ have been changed

that in UML a new class, **QuickSort** has been added and the class **ExplSort** has been modified. Moreover, the class **MergeSort** has been removed from the model. In VDM++ the class **MergeSort** has been modified and the class **ImplSort** has been deleted.

Figure 10 introduces three new class designators **M**, **-** and **D**, which together with the previously mentioned two designators constitute the five different class designators used in the Rose-VDM++ Link.

The three class designators have the following meaning:

M Indicates that the class was modified.

- Indicates that the class has not changed.

D Indicates that the class was deleted since the models were last computed.

To investigate the differences between two representations of the same class, simply click the checkbox next to the class and subsequently the “Diff” button. The result will be displayed in the log window. To compute the differences of the VDM++ and UML representations of all classes, press the “All” button to select all classes.

Figure 11 shows the result of computing the differences of the VDM++ and UML representations of all classes. We see, for example, that the two versions of the `ExplSort` class only differ in the name of one of the arguments to the function `RestSeq`; an “i” was changed to a “j”.

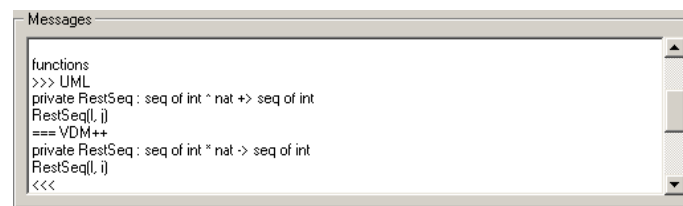


Figure 11: The differences between the VDM++ and UML representations for all classes

Assume now, you want to merge these two models into one single representation, which subsequently is propagated to the class repository in Rose 98/2000 and the `rtf` files containing the VDM++ specification.

In order to determine how the merge of the two representations should take place, the desired action has to be chosen by changing the states of the action buttons. See Figure 12 for a list of possible actions for the different classes.

Merging Classes with Conflicting Definitions

When the UML and VDM++ models are allowed to evolve in parallel the possibility of conflicts arise. If, for instance, both the VDM++ and UML model defines an operation with the same name but with different signatures the Merge Tool will not know which one of the two operations to choose for the resulting model. Clicking the “Map” button will start the merging of the two models,

























Class	Possible actions	Consequences when pressing the “Map” button
DoSort	 (default)   	<p>No updates will be made, because the class has not been changed in either model.</p> <p>as the default action</p> <p>as the default action</p> <p>The class will be excluded from both models.</p>
Sorter	 (default)   	<p>No updates will be made, because the class has not been changed in either model.</p> <p>as the default action</p> <p>as the default action</p> <p>The class will be excluded from both models.</p>
ExplSort	 (default)   	<p>The modification made in the UML model will also be made in the VDM++ model.</p> <p>The modification previously made in the UML model will be deleted. The class returns to its previous state.</p> <p>A conflict will result, as explained below.</p> <p>The class will be excluded from both models.</p>
ImplSort	 (default)  	<p>The class will also be deleted from the UML model.</p> <p>A new ImplSort class will be generated in the VDM++ model.</p> <p>as the default action</p>
MergeSort	 (default)  	<p>A new MergeSort class will be generated in the UML model including changes made in the VDM++ model.</p> <p>The class will also be deleted from the VDM++ model.</p> <p>The class will also be deleted from the VDM++ model.</p>
QuickSort	 (default) 	<p>The class will be added to the VDM++ model.</p> <p>The class will be excluded from both models.</p>
SortMachine	 (default)   	<p>No updates will be made, because the class has not been changed in either model.</p> <p>as the default action</p> <p>as the default action</p> <p>The class will be excluded from both models.</p>

Figure 12: Possible actions when merging the VDM++ and the UML models

but if conflicts are encountered, the merge will be aborted, and the user will be informed about the classes with conflicts.

For example merging the VDM++ and UML representations of the class `Exp1Sort` will cause conflicts, because the models differ in the name of one of the arguments to the function `RestSeq`. This conflict is also evident in Figure 11.

The user will typically use the “Diff” button to locate the conflict(s) and decide which one of the two definitions to keep, simply by modifying the UML or VDM++ model accordingly. When all conflicts are resolved, the VDM++ specification should be syntax checked and the Rose-VDM++ Link re-invoked to do the merging of the two models.

Instead of modifying one of the models to avoid conflicts, the user can also choose to change the mode of merging in order to solve the conflict. Simply click the action button to toggle between the different states and choose an action that suppresses one of the two representations. Conflicts will only arise when representations are merged!

Updating the VDM++ Specification

If the VDM++ and UML model are not in conflict with each other, clicking the “Map” button will automatically update the class repository of Rose 98/2000 to contain the result of merging the two original models. In the same way the `rtf` files of the VDM++ project will automatically be updated and parsed to reflect the changes made to the UML model. If new classes were introduced, they will be generated as new `rtf` files as described in Section 2.4.

Before updating an `rtf` file the Rose-VDM++ Link creates a copy of the file. The copy is named by adding `_old.rtf` to the original file name. I.e. the file `Exp1Sort.rtf` will be copied to `Exp1Sort.rtf_old.rtf` before it is updated.

The following rules are applied when the VDM++ specification files are updated:

New elements are added at the topmost position possible in the class. For example, a new instance variable will be inserted at the top of the first `instance variables` block of the class. If an `instance variables` block was not already declared in the VDM++ class, it will be generated at the topmost position of the class.

Obsolete elements will be removed from the file by converting them to VDM++ comments. In this way unexpected removals can easily be restored.

Modified elements are handled by simply removing (by comments) the old definition and adding the new definition.

All modifications of the specification files will be identified by comments like: “Added by the Rose-VDM++ Link” or “Removed by the Rose-VDM++ Link”.

Figure 13 shows how the class `ExplSort` was been updated by the Rose-VDM++ Link because the signature of the function `RestSeq` had been modified in UML.

In some situations the Rose-VDM++ Link will not be able to update the necessary files. Certain word processors lock the file(s) that is currently being edited such that other applications may not access them. As a consequence the Rose-VDM++ Link is unable to update such locked files. If files that need to be updated are locked by another application, the Rose-VDM++ Link will list the names of these files, and the user can then choose to either unlock these files (by making sure that other applications are not using them) or cancel the merge process.

All files modified by the Rose-VDM++ Link are subsequently automatically parsed so that the model contained in the VDM++ Toolbox is identical to the model contained in Rose 98/2000.

```
class ExplSort is subclass of Sorter
...
functions
-- Removed by the Rose-VDM++ Link:
--   private RestSeq : seq of int * nat +> seq of int
--   RestSeq(l, i) ==
-- Added by the Rose-VDM++ Link:
  private RestSeq : seq of int * nat +> seq of int
  RestSeq(l, j) ==
    [l(j) | j in set (inds l \ {i})];
...
end ExplSort
```

Figure 13: The updated `ExplSort.rtf` file

2.6 Type Checking of Specifications

In order to use the Rose-VDM++ Link the VDM++ specification need only be syntax checked, but it is recommended that it is type checked as well. Information about relationships between classes defined by instance variables is not available

until the specification is type checked. Therefore if a VDM++ specification is only syntax checked it may not be complete in the sense that the Rose-VDM++ Link cannot generate association, which are defined by such relations.

3 Rational Rose

So far we have discussed some of the features of the Rose-VDM++ Link. Since this tool relies on a very tight coupling with Rose 98/2000 it is within the scope of this manual to give a short introduction to Rose 98/2000. Initially we will describe how to create class diagrams from the classes generated by the Rose-VDM++ Link or classes created by the user. Section 3.2 will describe how to modify, delete or add any definitions in Rose 98/2000.

The Rose-VDM++ Link is a so-called Language Add-In of Rose 98/2000, which is installed as described in [1]. The Add-In may be activated or deactivated through the Add-In Manager of Rose 98/2000. Activating the Rose-VDM++ Link (it is activated by default) makes Rose 98/2000 aware of VDM++, the fundamental data types of VDM++ and the special stereotypes used by the mapping from UML to VDM++.

3.1 Class Diagrams in Rational Rose

Creating Class Diagrams

Translating VDM++ class definitions to UML only creates or updates the class definitions in the class repository of Rose 98/2000 - it does not automatically create class diagrams. However Rose 98/2000 makes it easy to create class diagrams.

In order to create class diagrams in Rose 98/2000, you should do as follows (note that the following actions are made from inside Rose 98/2000):

1. Either create an empty class diagram, or open an already existing one.
2. Add classes to the class diagram by either
 - Drag-and-drop: click a class in the class repository and drag it into the class diagram to which you wish to add it.
 - Select *Query/Add Classes...*, and choose the package containing the classes you wish to add (this would typically be the "Generated classes" package). Move the classes to be added into the "Selected classes" list box, and click "OK". This will automatically add all selected classes to the currently active diagram, and automatically layout the diagram as well.

In Figure 14 we show a class diagram created by following this approach.

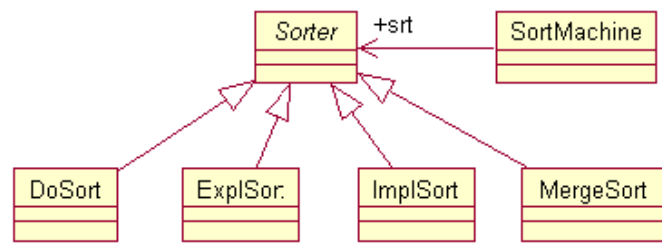


Figure 14: The class diagram in Rose 98/2000 for the Sort example

Modifying Diagrams

Rose 98/2000 also lets you change the layout of generated diagrams. Simply move the classes and relations to other classes as you like, by using the mouse. Modifying a class diagram in Rose 98/2000 has the following consequences:

- Changing the layout, i.e., by moving classes and relations only modifies the diagrams, will have no consequence to the class repository. Furthermore, if the Rose-VDM++ Link at a later time updates the repository, this will not alter the layout of the diagrams. Only if a class is removed from the repository, or if its relations to other classes are changed, this will change the appearance of the diagram. However, the layout will still be intact.
- Adding relations (either inheritance or association) between classes will change the class definition in the class repository, i.e. add the new relation to the model. The diagram in which the new relation was added will naturally show the changes immediately. Other diagrams will, however, not show the new relation(s), but must be updated in order to reflect the changes. The next section describes how to update your class diagrams.
- Changing the instance variables, operations, etc. of a class will immediately be visible in all other diagrams containing the class.

Filtering Relationships

Rose 98/2000 offers you the possibility to leave certain types of relations out of the currently selected diagram. Using this functionality you can easily construct a

diagram showing only the inheritance relations between the classes in the diagram. In Rose 98/2000, select *Query/Filter relationships...* in order to specify what kind of relations to display in the diagram. Figure 15 shows the dialogue box for filtering relations.

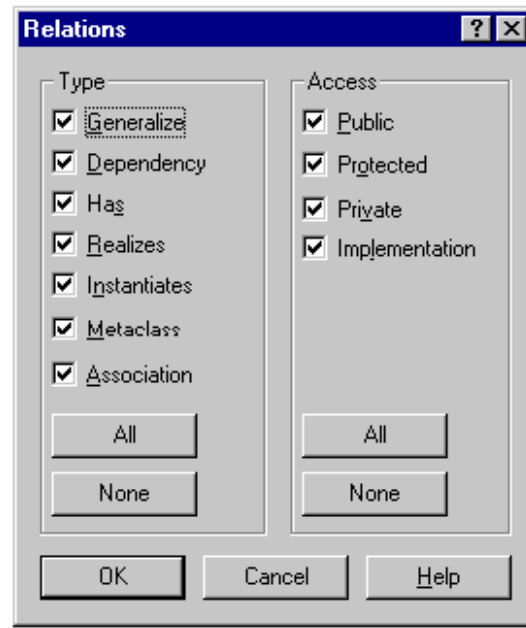


Figure 15: Specifying relations

You can also use this dialogue box to update the relations of a diagram. Simply select all possible kinds of relations (use the “All” button) and click “OK”. This will re-display all relations defined between the classes in the currently selected class diagram. This is the only way to make new associations added by the Rose-VDM++ Link visible in already existing diagrams.

Local Views

Often one single class diagram, showing all classes and their mutual relations, will become too complicated, and consequently difficult to comprehend. For this reason Rose 98/2000 gives you the possibility to generate local views.

To do so, generate a new class diagram, which initially consists of one or more classes, for which you want to generate a local view. Select afterwards the classes in this new diagram and choose the *Query/Expand Selected Elements...* menu item. A dialogue window will pop up allowing you to determine the number

of levels and the kind of relations you want to see in the generated local view. Clicking the “OK” button will now create a class diagram showing the generated local view.

Using this procedure you can for example easily generate a new class diagram representing a class and all its immediate super- and sub classes.

3.2 Manipulating Definitions in Rational Rose

At any time you are allowed to modify, delete or add any definitions in Rose 98/2000. This section presents some examples of how to change your Rose model.


Modifying Definitions in the Repository

The definition of attributes (i.e., instance variables and values in VDM++) and operations (operations and functions in VDM++) can easily be modified in Rose 98/2000. Simply double-click the class you wish to modify, and you are presented with a dialogue box offering you to inspect and change the definition of every part of the class. The “Class Specification” window is used to browse the specification of a given class. Figure 16 shows the class specification window for the `SortMachine` class.

Creating Certain Types of Entities

Here we will describe how to create some new entities in Rose 98/2000. It is recommended to read Section 4 in order to get some information about the entities of UML, which can be mapped to VDM++ constructs. The VDM++ add-in of Rose 98/2000 helps you when creating new entities. In the following, we will describe the creation of classes, attributes, operations and associations.

- **Classes**

A new class can be created by selecting the  button of Rose 98/2000. To make use of the fundamental VDM++ types and the predefined stereotypes defined by the Rose-VDM++ Link a new class must be assigned to a VDM++ component. To do so, follow these steps:

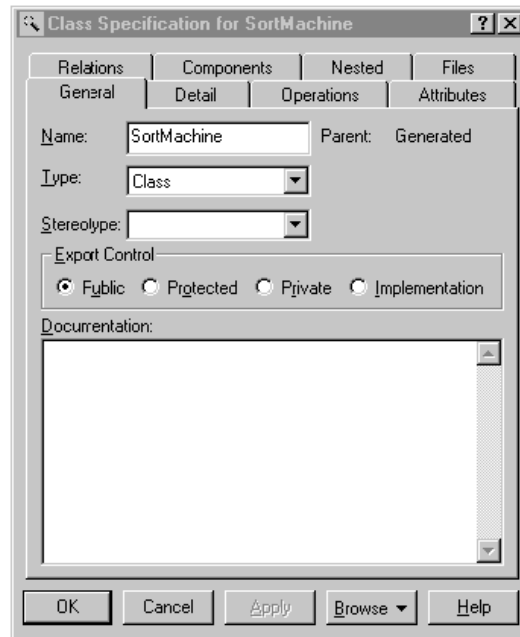


Figure 16: UML Class Specification for the class `SortMachine`

- Create a new component by right-clicking the “Component View” package in the browser of Rose 98/2000 and select *New/Component*. Double-click the new component, and in the specification window for this component (on the “General” tab) select VDM++ as the language for this component.
- Select the “Realizes” tab in the specification window for the new component. Select the classes that should be assigned to this component. Right-click the selection and choose “Assign”. Now these classes are assigned to the VDM++ component.

Alternatively classes can be assigned to a particular component by simply “dragging” them into the component with the mouse.

• Attributes

As described in Section 4 attributes in UML are used to represent instance variables and values of VDM++. A new attribute can easily be added to a class by right-clicking the class and selecting the menu item *New Attribute*.

We use *stereotypes*, as described in Section 4, to distinguish the three different VDM++ constructs. You can assign a stereotype (indicated by “<<”

“>>”) to each attribute. If you do not specify a stereotype the attribute will be considered as an instance variable by default.

The VDM++ add-in helps you in defining stereotypes for attributes. Figure 17 shows the “Attribute Specification” window of a newly added attribute. Because the class is assigned to the VDM++ component, one can choose between the three predefined stereotypes, <<instance variable>> or <<value>>.

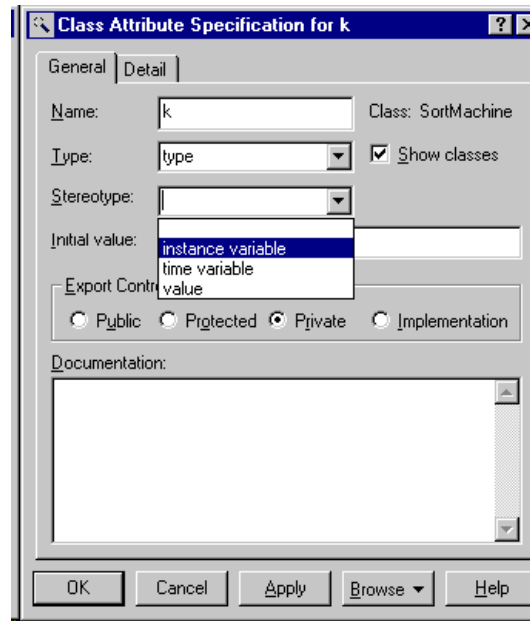


Figure 17: Selecting a predefined Stereotype for an Attribute.

- **Operations**

A new operation can easily be added to a class by right-clicking the class and selecting the menu item *New Operation*.

As with attributes, we use stereotypes to distinguish between functions and operations of VDM++. These stereotypes are created as with attributes. By default everything is considered as operations if the stereotype is omitted.

- **Associations**

To create an association between two classes, simply click the “Association Tool” and define the association by clicking one class and dragging the mouse to the other class. You can specify further details for the association by double clicking it, which will open the “Association Specification”

window. Here you can add role names, cardinalities and constraints to the association.

Role names: As described in the mapping rules the role name of an association is used as a name for the instance variable the association will be translated to. For this reason, giving an association at least one role name is vital for it to appear as an instance variable in the generated VDM++ class. If no role names are specified for an association it will simply be ignored.

Cardinality: If no cardinality is specified for a role a cardinality of one is assumed as default.

Constraints: In the “Association Specification” window it is possible to add constraints to the role of an association. This can be used to add a constraint like `{ordered}` to a role, which would generate a *sequence*, rather than a *set* of object references when translated to VDM++.

Qualifiers: A qualifier can easily be added to an association. Simply right-click the association close to the class to hold the qualifier and then select *New Key/Qualifier*. Specify as name of the qualifier the VDM++ type to be used as domain in the map used to model qualified associations in VDM++ (see Section 4). Notice that right-clicking the association near the role to be modified can modify many details of the two roles of an association.

4 Mapping Rules between VDM++ and UML

In this section we present the relation between VDM++ and UML. That is, we present the rules applied by the Rose-VDM++ Link when translating one representation into the other.

Clearly, to fully understand the following mapping rules, a detailed knowledge of VDM++ and UML is required. Here we will not introduce the two notations, but merely describe various constructs whenever necessary. We refer to [2] and [5] for a complete presentation. The rules presented in the following section are summarised in Figure 23 in appendix A.

The rules will be presented by presenting several VDM++ specifications along with their corresponding UML translation. In this regard it is important to mention that the defined mapping is *injective*, i.e., by defining the mapping from VDM++ to UML, we have defined the inverse mapping at the same time.

Furthermore, it is important to note that the following rules do *not* cover every construct possible to write in VDM++, nor do they cover every aspect of the UML. The mapping rules have simply been defined in order to make the mapping injective, meaning that constructs that can only be described in one of the two representations are left out of the mapping.

The Rose-VDM++ Link will simply ignore constructs not covered by the following mapping rules.

4.1 The Class Structure

This section covers how the internal class structure is being mapped between the two representations. Each subsection addresses various VDM++ constructs.

Instance Variables and Values

In VDM++ the internal state of an object is described by its instance variables and its value definitions. The main difference between values and instance variables is that values are read-only. In UML the internal state is described in the attribute section in which we will simply list the instance variables and values along with their corresponding types and value. In order to distinguish instance variables and values we will use two *stereotypes*, namely `<<instance variable>>`

and `<<value>>`.

The syntax for declaring attributes in UML is:

```
name : type = initial-value
```

which is almost identical to the syntax of instance variable and value definitions in VDM++. Consequently values and instance variables can be mapped directly to UML. Figure 18 illustrates the mapping of instance variables and values.

Operations and Functions

Operations and functions map into the operations section of the UML class. They are distinguished by the stereotypes `<<operation>>` and `<<function>>`. The syntax for defining operation headers in UML is:

```
name(parameter : type, ...) : return-type
```

which is almost identical to the syntax for implicitly defined functions/operations in VDM++.

Explicitly and implicitly defined function headers, as well as operation headers must be transformed into this syntax. See Figure 18 for an example of mapping operations and functions.

Pre and post conditions defined in operations are preserved by the UML mapper. Thus they need to be written as syntactically correct VDM expressions. Since UML does not provide any way of defining a result identifier for an operation, the special identifier **RESULT** should be used in post conditions to represent the result of the operation.

Mapping explicit and implicit functions into the same syntax in UML makes it difficult to distinguish between implicitly and explicitly defined functions when mapping from UML to VDM++. For this reason, the following rules are applied when mapping functions from UML to VDM++:

- If a function is already defined in VDM++, it is mapped into the same kind (implicit or explicit) as defined in VDM++.
- If a function is not known in VDM++ (i.e., the function was defined at the UML level) it is mapped as explicit.

4.2 Associations between Classes

In VDM++ an object (an instance of a class) may have relations to objects of other classes or objects of its own class. Such clientship relations are possible through the object reference type. In UML such relations are called associations and represented by an arrow from the *client* class towards the class being referenced. In this way the arrow indicates the *navigability* of the Rose-VDM++ Link, i.e., which object can in fact be referenced by the other. Figure 19 shows how such simple associations are mapped between VDM++ and UML. Notice, that the instance variables representing the object references are not shown as attributes in the UML class.

In UML the *far end* of an association is called a role. Usually roles are given names which, at least in real world models, tend to *describe* the role of the association. Roles are particular helpful in the case of binary associations. Here we will construct role names simply from the name of the instance variable representing the association. This is also shown in Figure 19.

Adding Multiplicity to Associations

So far we have only considered simple “one-to-one” associations, i.e., relations connecting one instance of a class with exactly one other instance. By using constructs like `seq of A` and `set of A`, it is possible to relate one instance to several other instances. In UML multiplicity is represented by adding numbers/symbols to the ends of associations, thereby indicating the multiplicity.

We will now present how different VDM++ constructs related to multiplicity of associations are mapped to UML.

set of objref: This construct is translated into a “one-to-many” association, which in UML is modelled by placing a “1” at the one-end, and the range “0..*” at the many- end of the association.

seq of objref: This is also a one-to-many association, but using the sequence type adds an ordering to the references, which in UML is indicated by adding the ordered constraint to the many-end of the association.

seq1 of objref: The non empty sequence of object references is represented by the range “1..*” at the many-end of the association.

[objref]: The optional object reference will be identified by adding the range “0..1” to the role of the association.

```
class Queue
```

```
types
```

```
  public Item = token;
```

```
instance variables
```

```
  q : seq of Item := [];
```

```
  inv len q <= max;
```

```
values
```

```
  max : nat = 256;
```

```
operations
```

```
  public Enqueue(i:Item) ==
    q := q ^ [i];
```

```
  public Dequeue : () ==> Item
```

```
  Dequeue() ==
```

```
    is not yet specified;
```

```
functions
```

```
  public Merge (q1 : seq of Item,
                q2 : seq of Item) q : seq of Item
```

```
  pre IsSorted(q1) and IsSorted(q2)
```

```
  post IsSorted(q);
```

```
  IsSorted : seq of Item +> bool
```

```
  IsSorted(q) ==
```

```
    forall i,j in set inds q & i < j => LessThan(q(i), q(j));
```

```
  LessThan : Item * Item +> bool
```

```
  LessThan (i,j) ==
```

```
    is not yet specified
```

```
end Queue
```

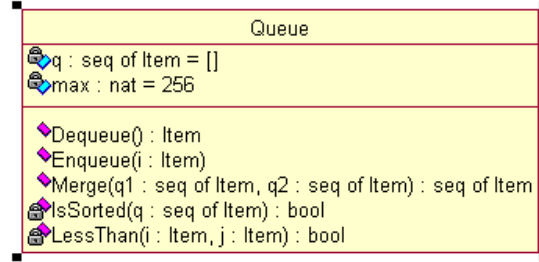


Figure 18: Mapping a class between VDM++ and UML

These constructs are summarised in Figure 20.

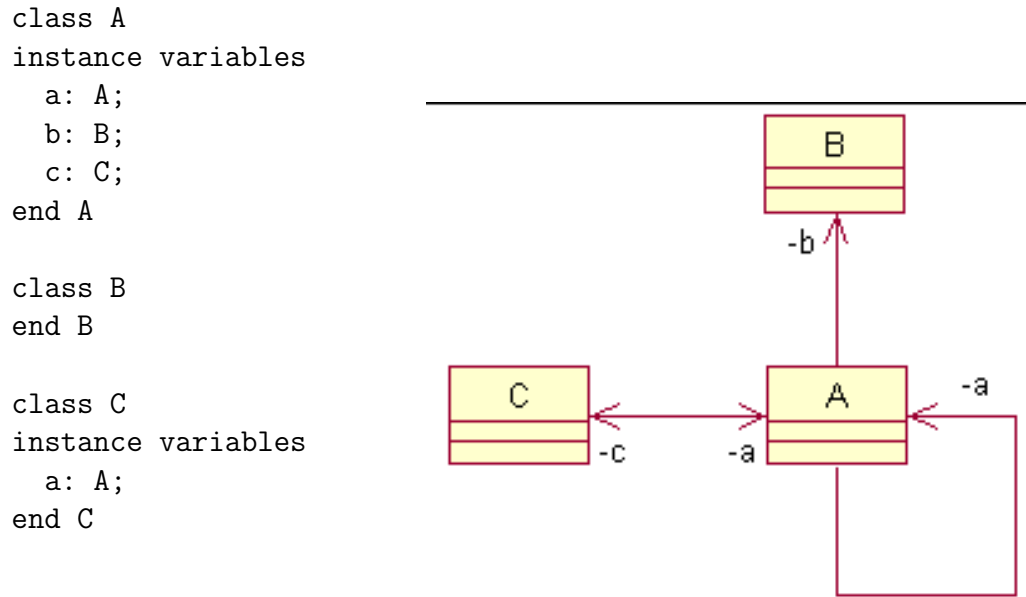


Figure 19: Mapping object references/associations between VDM++ and UML

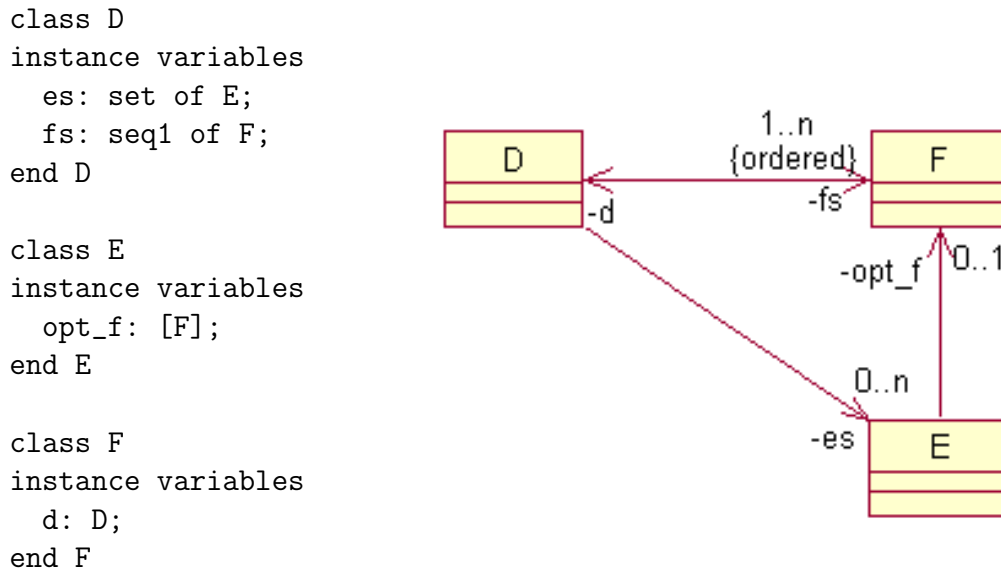


Figure 20: Mapping multiple object references between VDM++ and UML

Object References using map and inmap

In VDM++ it is possible to relate objects using the map and inmap type. VDM++ constructs like: `map type to objref`, where `type` can be any VDM++

type and `objref` is a simple or multiple object reference will result in a *qualified association*.

The multiplicity of qualified associations is added as with the other types of associations. In UML the qualifier itself can be assigned a name, which in this case will be the type of the domain of the map. As with other associations the role name of the association is the name of the instance variable representing the map. See Figure 21 for an example.

```
class G
instance variables
  qual_h: map nat to H;
end G

class H
instance variables
  qual_j: map real to J;
end H

class J
instance variables
  qual_h: map nat to set of H;
end J
```

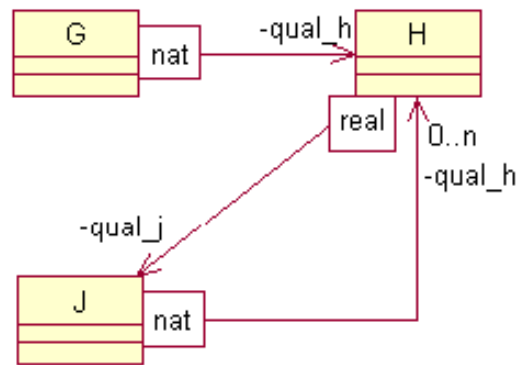


Figure 21: Qualified associations from object references defined using maps

Inheritance Relationship

The translation of inheritance from VDM++ to UML and vice versa is straightforward. Figure 22 shows an example. The class `SubA` and the class `SubB` inherit from class `Super`. In UML, inheritance relationships are shown as generalisations.

Delegation

In VDM++ the specification of operations can be delegated to subclasses by using the `is subclass responsibility` clause. In UML such operations are called *abstract operations*, and classes containing abstract operations are called *abstract classes*, as opposed to concrete classes. A class will be considered abstract, if it delegates at least one of its operations (i.e., at least one operation is specified

```
class Super
operations
  methodA() ==
    is subclass responsibility;
  methodB() ==
    ...;
end Super

class SubA is subclass of Super
end SubA

class SubB is subclass of Super
end SubB
```

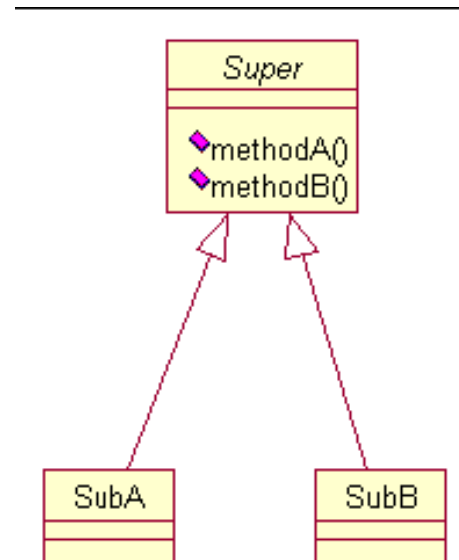


Figure 22: Mapping inheritance between VDM++ and UML

using `is subclass responsibility`), otherwise it is considered concrete. In UML abstract classes and operations are identified by writing their names in *Italics* font.

References

- [1] CSK. *VDM++ Installation Guide*. CSK.
- [2] CSK. *The VDM++ Language*. CSK.
- [3] CSK. *VDM++ Sorting Algorithms*. CSK.
- [4] CSK. *VDM++ Toolbox User Manual*. CSK.
- [5] GRADY BOOCH, IVAR JACOBSON AND JIM RUMBAUGH. The Unified Modelling Language, version 1.1. Tech. rep., Rational Software Corporation, September 1997. Available at: <http://www.rational.com/>.
- [6] RATIONAL. *Rose 98 User Manual*, 1998.

A Summarising the Mapping Rules

The table in Figure 23 summarises the mapping rules applied by the Rose-VDM++ Link:

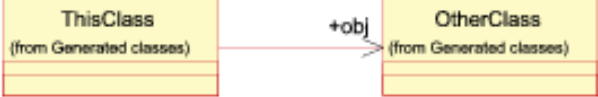
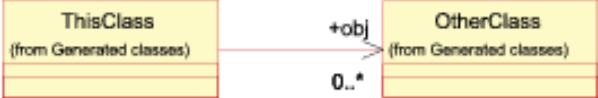
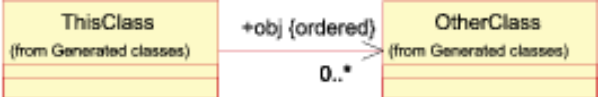
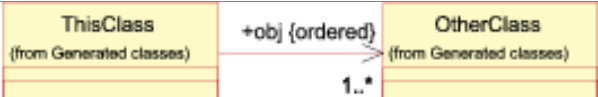
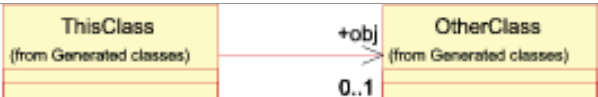
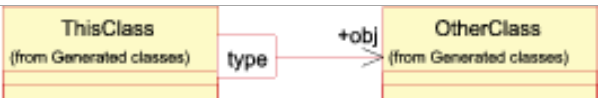
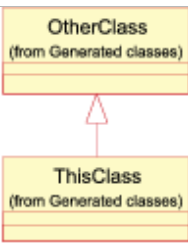
VDM++	UML
instance variable	attribute with stereotype <<instance variable>>
value	attribute with stereotype <<value>>
operation	operation with stereotype <<operation>>
function	operation with stereotype <<function>>. Functions, that are only defined in UML are mapped to VDM++ as explicit functions. Otherwise functions are mapped implicit or explicit as defined in VDM++.
obj: OtherClass	
obj: set of OtherClass	
obj: seq of OtherClass	
obj: seq1 of OtherClass	
obj: [OtherClass]	
obj: map type to OtherClass	
class ThisClass is subclass of OtherClass	

Figure 23: Mapping rules applied by the Rose-VDM++ Link

B Warnings generated by the Rose-VDM++ Link

Review Figure 1. Before merging the two models, both the VDM++ and the UML models are translated to an internal representation.

This translation involves some checks, which can lead to different warnings.

Warnings Generated during the Translation from UML to VDM++

When a UML class is translated to VDM++, the UML attributes and operations are converted to VDM++ constructs. The name and types used in the UML model must consequently comply with the syntactical rules of VDM++. If this is not the case, the UML definition is simply ignored and the user is notified by a warning. This is done in order to maximise the syntactical correctness of the VDM++ specification generated.

As an example look at the UML class shown in Figure 24.

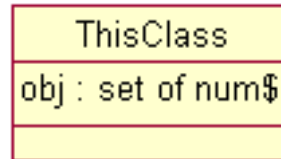


Figure 24: UML class that does not comply with the syntax rules of VDM++ (the \$ sign cannot be used here)

The generated warning when loading the UML model is shown in Figure 25. Mapping the UML model to VDM++ without removing the UML definition causing the warning will result in the following VDM++ class definition:

```

class ThisClass
end ThisClass
  
```

As you can see, the generated class definition is simply empty.

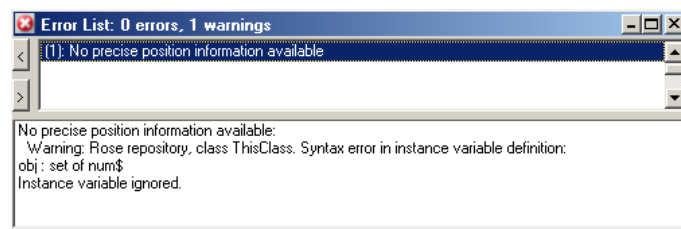


Figure 25: The warning generated if the definition of an instance variable does not comply to the syntactical rules of VDM++