Statement of the problem

We have a Rose ANSI C++ model with 4 controlled units, each being a namespace (the source code of the classes in one namespace should be ultimately used to build a shared library). Some of namespace packages have sub-packages. All the classes are assigned to a single ANSI C++ component. We want to migrate Rose model into XDE for Java (C++) and to continue development of the model and the code. This means, we want to jointly use 2 Eclipse plug-ins: XDE for Java (C++) for models, and CDT for C++ development.

Application versions required:
- XDE developer plus for Java 2003.06.12 (this is the only one that supports non M$ C++)
- PICT IBM installed to be able to use Eclipse 2.1.x (with the previous registry tweak)
- CDT 1.2 (running with Eclipse 2.1.x)
- Cygwin with make, gcc/g++, gdb

All is running on Win2k, SP4.

We have grouped into packages those classes that are supposed to be declared and implemented in a single pair of (`.h`, `.cpp`) files. We use a Rose script (see the appendix) that will assign ANSI C++ properties `HeaderSourceFile` and `BodySourceFile` with the names of the containing packages. Therefore, for a namespace package without sub-packages, there is a single pair of files (`namespacePackageName.h`, `namespacePackageName.cpp`), while for the namespaces with sub-packages, there is one pair of files per sub-package (`subPackageName.h`, `subPackageName.cpp`).

The ANSI C++ component (the only one we have) has as root directory `$CURDIR\code`. Creating a number of components that would follow the desired namespace modules doesn't bring anything with the ANSI C++ add-on (only with the standard C++). Also, setting the option to generate namespace packages as directories will produce include directives with the hard-coded names (i.e., `#include directory\file.h`), which is certainly not what we want.

In XDE for Java (C++), we want to have one project per namespace, i.e., per controlled unit of Rose model. In XDE for Java (C++), it is possible to assign the `BodySourceFile` property per class, or per group of classes. However, it is impossible (or we found no way) to assign the `IncludeSourceFile` property to classes, or to packages. XDE will also ignore the corresponding Rose ANSI C++ model properties (at least, if there is no associated source code), so we have to:
1. Prepare Rose model and generate code into appropriate files (all within the component root source directory);
2. Then migrate both the model and the code to XDE; and,
3. Finally, combine XDE C++ code models and their code into CDT projects.

Before describing in detail the steps 1)-3) above, in the following section we discuss some of the limitations of XDE and its handling of Rose ANSI C++ models, which makes it indispensable to carefully do the preparation before the actual migration. This is by no means an exhaustive list, but reflects just our findings during this migration.

XDE for Java (C++): what you must consider first

- XDE generates no code for bi-directional associations, i.e., those whose both roles are navigable. Therefore, consider replacing each of them with two directional (navigable in one sense) associations.
- XDE accepts at the most one blank line within the comments in code (for classes, etc.) or within the documentation of UML elements (classes, attributes, operations, parameters, etc.). To be on the safe side, avoid any blank line in the code comments, and in the elements documentation.
- XDE does not synchronise the documentation for associations. I.e., it does create comments at the first code generation, and it does create UML documentation at the first reverse engineering from code, but it

does not keep up to date further changes. So, once you have the code and the model, all the updates to the documentation/comments for associations have to be kept up to date manually.
- XDE translates Rose class attributes which are UML classes, typedefs or enums (and not primitive types) into associations with no cardinalities, as they are implicitly [1..1]. There is no automatic way to transform them back to attributes with XDE for Java (C++); if needed, you have to do it manually (as a side note: XDE for .NET allows the conversion of association to attribute and vice versa).
- XDE attaches the documentation of the Rose association role, to the XDE association itself, not to its navigable role (as a side note: XDE for .NET "elegantly" ignores the documentation of role, so you have to copy the role documentation to the association documentation before migration!).
- If importing Rose ANSI C++ model without source code (which would allow us to synchronise the code with the Rose .mdl migrated into XDE .mdx), XDE ignores `BodySourceFile` and `IncludeSourceFile` (class) properties of Rose ANSI C++. XDE will simply generate a pair of (`.h`, `.cpp`) per class (only `.h` file for typedefs and enums), which may be overkill, and that's why we had to proceed as explained in the next section.
- We found no way to import Rose analysis model (without ANSI C++ properties) and to make an XDE C++ code model out of it.
- XDE C++ code model must have a single root directory (like in Rose), but that directory MUST be under the XDE C++ code project folder (unlike Rose). In other words, it is impossible to point to a directory with sources on the file system; they must be within the same directory as the code model .mdx, or below it.
- XDE C++ code model can have references to other C++ code models (i.e., projects that contain them), but it CANNOT reference any other kind of XDE model, neither content model nor another language code model.
- (Acknowledged bug) After adding a reference project to an XDE C++ code model, you must first close the XDE C++ code model and reopen it, in order for it to "see" the added reference.

Independently of this conversion project, here some more missing or buggy features:
- Synchronisation in XDE C++ does not make certain updates correctly. For instance, when we change the name of a class `Old` into `New`, the code of another class, say `User` that has an attribute or an association end of type `Old` (e.g., `User.attribute : Old`), is not synchronised, i.e., in `User.h` I'll still have `Old`, and not `New`. The same applies to the name of the attribute or association, if changed.
- For types of attributes, parameters etc., there is no possibility to click on the combo-box (as in Rose) and choose the type out of those already defined in the model. The `TypeExpression` property is simply a string, so you have to type in whatever you need (and whatever you want – no validation here)!
- XDE does not import from Rose the constraints on association ENDs.
- XDE does not allow one to attach a constraint to an operation (although the help mentions that this is possible).
- Etc.

Let's go back to our model that we first have to prepare in Rose ANSI C++ for migration to XDE C++.

Rose ANSI C++ model and code preparation

We are starting from a Rose ANSI C++ model for which the code has never been generated. We want to generate code with a custom source file assignment (all classes in a package go into one pair of (`.h`, `.cpp`) files), to be able to logically separate the code per namespace (i.e., control unit in our case). Each of these control units will have to be transformed into one XDE C++ code model project.

(1.a) Insure that the auto-syncronisation is turned off!

We first configure our only C++ component. By changing the default property set as follows (note that all but the first are optional):
- Root (code) directory: should be `$CURDIR\code`, but as the folder is deep in the hierarchy, it makes problems, so we just temporarily use a directory near the top (e.g., `d:\data\RoseCpp\code`)
- Use Tabs: True
- Tab Width: 4
- ModelIdCommentRule: Code generation and reverse engineering

- Copyright: “Your copyright stuff”

(1.b) Then we assign the file names, by running the script `setFileNamesCppH.ebs` (see the appendix). Open the script and run it once per package, while having all the classes in the package selected.

(1.c) Now we can generate code, and do debugging, compilation etc. (for instance, by creating a CDT managed project in the `d:\data\RoseCpp\code` directory). At each file change, reverse engineer the code into the model to have it updated. Note that after reverse engineering in Rose, even if you exit Rose, the `rose.exe` process will still be running, so you’ll have to kill it, otherwise the ANSI C++ add-in options are not available.

(1.d) Before making the final version for XDE, change the root source directory of the C++ component to `$CURDIR\code`. Copy the already compiled and synchronised source code from (the temporary) `d:\data\RoseCpp\code` directory into `$CURDIR\code`. Do a last synchronisation (reverse engineering), save, exit Rose and kill `rose.exe`.

(1.e) We want to keep the original Rose control units (namespace packages) accessible from within another (analysis) Rose model. As it is too complicated to import a Rose model with controlled units into XDE C++, the fastest way to go is to create a copy of the model, un-control all the controlled units, and save the obtained model, which now contains it all within a single .mdl file. This is the Rose ANSI C++ model that we resynchronise once more with the code (just to be on the safe side), and that we migrate to XDE C++.

Creating XDE C++ code models from Rose ANSI C++ model and code

As already mentioned, we are going to create one XDE C++ code model project per namespace (four in our case), plus one that is just an overview of the whole. Therefore, we create five directories, each with a code subdirectory. These will be project directories for our initial XDE C++ code models. Note that the project directory is flat, instead of having, for instance, the overview folder hold the 4 sub-folders (namespace packages). The reason is that with XDE C++ (and Eclipse in general), you cannot have projects whose home directories overlap.

We will first migrate the Rose ANSI C++ model and code to XDE C++ code model, then create 4 empty XDE C++ code models, and finally drag and drop the corresponding namespace packages (and their code) from the full XDE C++ code model into the empty XDE C++ code models.

(2.a) Copy the Rose ANSI C++ model into the overview directory (`_xdeCpp`), and all of its code in the corresponding subdirectory (`_xdeCpp\code`).

(2.b) Launch XDE and create a *New->Simple* project (`xdeCpp`) in non-default directory (`_xdeCpp`). The project navigator shows the Rose ANSI C++ model (.mdl). Double-clicking will transform it into XDE *content* model, and show it in the model navigator. Right click on the model and select “Create C++ code model”. The petal translation wizard will appear. There are only 2 things to set through the wizard. The first is on the second wizard page, to select the Rose ANSI C++ component (`CppComp`). The second is on the third wizard page, where to create the C++ *code* model – select “Use existing” and browse to the current project, and give it a name (e.g., `xdeCpp code model.mdx`). Model explorer now shows two models, the second one being the C++ code model we just created. At this point, you can close the original Rose model without saving it (this is a known bug, that the original Rose .mdl file gets “dirty” although it has not been modified).

(2.c) Now we have an XDE C++ code model. The petal translation wizard translates all the Rose views (Use Case, Logical, Component and Deployment), and we need only the namespace packages and their Main diagram(s) that are under the Logical view. So, move the namespace packages and the diagrams from Logical view under the model root (drag & drop), copy eventually the documentation for Logical view, if any,

and then remove the rest. In our case, the model now contains one overview diagram and four namespace packages. Save the model, close the model, change to project navigator and close the project.

Now, be ready for some repetitive work. The following steps need to be done for all (in our case: four) namespace packages – we have to create empty XDE C++ code model projects for them. Start with the project for the basic namespace (which depends on no other one), continue with those namespaces that depend only on the basic one, and so on. This is important, because at each new project, you need to add references to the other, existing projects, if necessary.

(2.d) Create a New->*XDE modelling->C++ code* project. Give it a name (`xdeTypes`) and create it in the non-default directory (`_xdeTypes`). In the next wizard page, if the model depends on another model, select the existing projects to refer to (for the first one you create, for the basic namespace, there will be no references to other projects). When done, a default C++ code model appears in the model explorer. In the Properties of the model, change its name to something meaningful (`xdeTypes code model`).

(2.e) Now we have to set two preferences for roundtrip engineering (RTE). In the menu Window->Preferences->Rational XDE->C++->Round-trip engineering, set the "Root directory" to the prepared code sub-directory of the project (browse to `_xdeTypes\code`), and for ModelIDs, select "Code generation and reverse engineering" option. Save the model, close the model, change to project navigator and close the project (note: closing the model is mandatory because the referenced projects will not appear if the XDE C++ model is not closed and re-opened; closing the project is to easily distinguish the project we are working on, and which is the only open one).

Repeat the steps (2.d) and (2.e) for all the remaining projects that will host namespaces. When done, there are all together 5 projects, each of them hosting one XDE C++ code model: the first one is the full model migrated from Rose, the other four are empty C++ code models, one per namespace package.

In the next steps, we will move the namespace packages (and their code) from the full model to the separate models, and keep the original (now empty) model for the overview purposes, only.

(2.f) Open all five projects (you can do the grouped open). Then, open the models one by one (by double-clicking on respective .mdx file), in the order you created them. In the model navigator, there will be at the top the full model, and then the four empty models.

(2.g) In the model navigator, first move the respective namespace packages from the full model to the other four models (optionally, at this point, for each project, you can open the model's Main diagram and drag the namespace package onto it).

(2.h) In the project navigator (or in the file explorer out of Eclipse and XDE), show the code sub-directories of all the projects (`_xdeCpp\code`, .., `_xdeTypes\code`), and move the sources from the full model code sub-directory, to the corresponding code sub-directories of the other four projects. Switch to model navigator, and for each of the four models (each now containing a namespace package), right click on the model and select "Add source files" to verify that the correct files figure in the list (i.e., those that are under the respective project code sub-directory). Save each of the four (namespace) models, but not (yet) the overview model.

(2.i) In the model navigator, for the (now empty) overview model, right click and select "Add source files" option, then remove all the files. Again right-click on the model and select "Properties", then "Project References" and check the four projects containing namespace code models. Then save the overview model, close the model and reopen it. Double-click on its Overview (Main) diagram and verify that all the packages on the diagram now have reference adornment (they refer to the namespace packages from the other C++ code models). If you had no overview diagram, you can drag the namespace packages (of other models) onto the main diagram of the overview model.

(2.j) Finally, synchronise each of the four C++ code models with the code. Before synchronising, you may want to clear the Output window, to see exactly the XDE log, of what is being done.

(2.k) Now you may want to set some more preferences for new model elements (like defaults for new classes, their getter and setter methods, etc.).

Yes, it's true, we have thus migrated Rose ANSI C++ to XDE C++ code models. The last thing to do is to create CDT projects, to be able to do build/compile/debug cycles with our code.

CDT C++ managed make projects for XDE C++ code models

Thanks to the help from IBM/Rational support, I have learnt that XDE C++ projects have nothing special with respect to models, since it is the XDE code *models* that carry all the necessary information, not the *projects* hosting them. Therefore, there is absolutely no problem to import, e.g., an XDE C++ code model (.mdx) into a CDT managed make project. This was good news for me, who tried the other way round (to create a CDT project under XDE C++ project), which, obviously, didn't work because of the overlap of project directories.

So, now that we have several XDE C++ code models and their source code, we create CDT managed projects, import the source code from the code sub-directories of XDE projects, and finally import the XDE C++ code models themselves. This way, we'll have it all together, within the managed make CDT projects.

(3.a) Create in your file system the five directories that will host CDT projects (`cdtTypes`, …), each with a code sub-directory, in the similar way we did for XDE projects (`cdtTypes\code`, …).

(3.b) Go back to XDE (Eclipse) and switch to the C/C++ perspective. *Important note*: To disallow automatic "build on save" for CDT, it is not enough to set the corresponding CDT preference – you must uncheck this option at the global (workbench) level, through the menu Windows->Preferences->Workbench. The managed make CDT build algorithm is somewhat buggy, so if you leave this global option enabled (as you usually do for java builds) while using CDT, you'll spend a lot of time staring in the repetitive builds (make clean all) of ALL the CDT projects, at every single file import or still worse, file save (I hope this has been fixed in CDT 2.0…).

What follows is again quite a repetitive task, as it should be done several times (in our case: five).

(3.c) Create one after another the five C++ managed make projects, starting from the simplest one (that will host the base namespace), and going towards those that will depend on already created CDT projects. Give the project a meaningful name (`cdtTypes`, …, `cdtMain`). The non-default project directory should be one of those that we just created in the file system (`cdtTypes`, …, `cdtMain`).

(3.d) Inter-project dependencies should exactly mimic those used when we were creating XDE C++ code projects.

(3.e) The first four CDT projects (`cdtTypes`, …) will host the code and the XDE C++ models of our namespaces, and should thus be Cygwin shared library (dll) projects. The fifth one (`cdtMain`) will host only a `main.cpp` source file and the overview XDE model, and will serve as the test driver for the above dll projects. Therefore, it should be a Cygwin executable (exe) project.

(3.f) Since CDT 1.2 does not deduce search paths for include files and libraries based on inter-project dependencies, for each project (except the first one in our case) you have to go to Project->Properties->C/C++ Build and appropriately set three things: Directories (for `.h` files), path to Libraries (dlls) and the names of Libraries themselves. *Important note*: For Windows paths, you have to use normal slash '/' instead of backslash '\' (e.g., not `d:\data\cdtTypes` but `d:/data/cdtTypes`). For each referred project, create a new Directory entry as a relative path to the place where `.h` files reside (`../../cdtTypes/code`), a new Library path entry (`../../cdtTypes/Debug`), and a new Library name entry (`cdtTypes` will link `libcdtTypes.dll`). This step is the most tedious of all.

(3.g) For the OS to find our dlls, we have to add the paths to the environment (right-click on MyComputer, in tab Advanced go to Environment). For instance, add two entries:

MY_APP_PATH	<common path to all CDT projects directories>
MY_APP_DLLS	%MY_APP_PATH%types\Debug;%MY_APP_PATH%<next project>\Debug; etc.

At this point, we have created five empty CDT managed make projects, with properly set build, link and environment paths. The last step is to import the source code and the XDE C++ code models into each of them.

(3.g) In the project navigator, for a dll project, select its code subdirectory (cdtTypes/code) and with right-click do Import->FileSystem and browse to the corresponding XDE project code sub-directory (_xdeTypes/code). After OK, the CDT will automatically build the project (libcdtTypes.dll). For the exe project, there is nothing to import in our case, but you can just create a main.cpp file in its code sub-directory (cdtMain/code) and write some test code – CDT automatically builds cdtMain.exe.

(3.h) In the project navigator, for each project (starting from the base namespace, cdtTypes, finishing with the main, cdtMain), select the project and do menu File->Import->XDE Model (note: never do copy/paste/move from the file explorer, since there is not only the model file, .mdx, but a few hidden files as well; if you do the Import, XDE handles that correctly). Browse to the respective XDE project directory (_xdeTypes, …, _xdeCpp) and select the model. If the XDE C++ code model has references to other models, you'll be asked to resolve broken references. This means, you'll replace the references to the XDE models residing in XDE projects (from where we import) to the references to models we just imported into CDT projects. Since the CDT projects have a code sub-directory, in the same way as our migrated XDE C++ code projects had, everything should be in place.

(3.i) Optionally, you may want to delete the XDE directories and projects in it, since we have them now within managed make CDT projects. Or, you can keep them for a souvenir :-)

Et voilà, that's it. Good luck!

Appendix: The Rose script for assigning source file names

```
'setFileNamesCppH.ebs
'The following script sets the HeaderSourceFile and BodySourceFile
'properties of each ANSI C++ class selected, to that of the package the
'class belongs to.
'
'T.Kostic on 2004-08-14
'Used and modified Main from package2directory.ebs
'
Option Explicit

Sub Main
      Dim selectedClasses As ClassCollection
      Set selectedClasses = roseApp.CurrentModel.GetSelectedClasses()

      Dim iClass As Integer
      For iClass = 1 To selectedClasses.Count
        Dim aClass As Class
        Set aClass = selectedClasses.GetAt(iClass)
        If aClass.GetAssignedLanguage() = "ANSI C++" Then
                Dim packageName As String
                packageName = aClass.ParentCategory.Name
                Dim IsOverridden As Boolean
                IsOverridden = aClass.OverrideProperty("Cplusplus","HeaderSourceFile", _
                                                       packageName & ".h")
                IsOverridden = aClass.OverrideProperty("Cplusplus","BodySourceFile",_
                                                  packageName & ".cpp")
        End If
      Next iClass
End Sub
```