Rational Rose RealTime Migration to Rational Software Architect RealTime Edition

Migration Best Practices

Steven R. Shaw

September 14, 2009

Introduction

Rational Software Architecture RealTime Edition (RSARTE) takes an evolutionary leap from the existing Rational Rose RealTime (RoseRT) technology base for two main reasons:

1. RSARTE is built on the Eclipse Integrated Development Environment (IDE) platform (www.eclipse.org) and
2. RSARTE is based on the latest UML2 specification (http://www.omg.org/spec/UML/).

In addition, RSARTE integrates with the existing stack of Rational UML Modeling tools that are built into Eclipse; these tools provide an all-in-one development environment for enterprise modeling and code generation. Consequently, this change in underlying technology means that the migration from the Rational Rose RealTime tooling is complex and necessitates some paradigm shifts to accommodate the integrations into Eclipse and Rational Software Architect.

Technology mapping from Rose RealTime to RSARTE

Technology	**Implementation RoseRT**	**Implementation RSARTE**
Base platform	Windows MFC application (porting technology allows it run on Linux / Solaris)	Eclipse IDE + tools (GMF / GEF)
Semantic specification	UML 1.x + Custom RealTime Extensions	UML 2.1
Meta-model engine	Custom implementation	EMF http://www.eclipse.org/modeling/emf/
Workspace	Single model per workspace	Multiple projects + multiple models per workspace

The Rational Rose RealTime import wizard in RSARTE attempts to make migration as seamless as possible, but it still requires direction at the Enterprise - architecture level to ensure that all software components are migrated in a scalable way. The goal of this document is to give some guidance to the "Enterprise Architect" whose role is to oversee the migration process and coordinate teams to ensure that they migrate in a compatible way to each other with-in an enterprise organization.

This document assumes that the reader is an expert in Rose RealTime model structure and general functionality. In addition, knowledge of Source Control management systems such as Rational ClearCase is beneficial to understand how the resulting fragmented model is stored in the repository.

Using the Rose RealTime Import wizard

If the various sections and parts of Rational Rose RealTime models were always contained in a single model file, then the migration process would be straightforward requiring little architectural consideration other then the how big the resulting monolithic model would be after migration to RSARTE (since the entire model would need to loaded into memory all the time). However, this is not a practical arrangement in RoseRT because many developers reside in different teams and typically contribute to a single model or top level executable. Consequently, it is essential that the model be sub-divided into smaller controlled units to allow for sharing and proper control with-in a source control management system such as Rational ClearCase. How a model is divided and organized is complicated further by the fact that a Rose RT model can share controlled unit packages with other models. This fact raises a number of questions: should these shared packages exist in a multi-project workspace environment such as Eclipse or should they be shared in their own project? Should they exist in the context of their original model owner? There are many ways to arrange the model architecture; therefore, the Rational Rose RealTime model import wizard allows for many different permutations to accommodate various model architecture approaches.

Controlled Unit Converter

The Rational Rose RealTime model import wizard has several pages that provide options on how to import the model into RSARTE. The most significant is the "Controlled Unit Conversion" page which allows you to select how the different controlled units in the RoseRT model are migrated into RSARTE. A "Controlled Unit" is a separate file that stores a particular model element at its root. Controlled units allow you to edit elements such as packages and classes independently of other elements in the source control system. The corresponding terminology in RSARTE is called a "fragment," which is essentially the same thing as a controlled unit.

ControlledUnit Conversion Table

The following table describes the different ways controlled units can covert to fragments in RSARTE:

Convert to:	**Description**
Absorbed Element	Unit is loaded into the same resource as the owning model or root package; no fragment is created.
Owned Fragment	Unit is converted into a fragment in the same relative location to the model as in RoseRT, unless a Project is specified (Packages only). If you specify a project, the Project is created (if it doesn't exist) and the fragment will be created as a root package in the project.
Shadow Package	This option creates a new fragment that is owned by a RSARTE model that is a shadow of a corresponding RoseRT controlled package unit. The package is read-only and changes can only be made in the corresponding RoseRT package unit that can be synchronized into the shadow package. Any controlled units that are owned by this package are absorbed into the package. It can also be extracted into a separate package similar to owned package fragments.
Short-cut	Available only for packages that have been imported into another model in the workspace. Only packages for models that are currently open in the workspace can be detected as previously imported. On import, the unit is converted into a short-cut and references will point to the originally imported package.

On import, controlled units can be in a number of different states that represent how they will be imported. For each state, there is a default action that takes place; this action depends on the "Owned by model" attribute on the "Unit Information" tab of the element specification dialog in RoseRT.

The following table lists the default conversion states; you can adjust these states as needed:

Controlled Unit Package State in Rose RT	Default Action
Owned but not migrated previously	Load as owned
Owned but detected as migrated previously in another model in workspace	Load as element import to existing package
Owned but detected a shadow package exists	Load as owned and log a warning that shadow package exists
Shared but not migrated previously	Load as owned shadow package
Shared and migrated previously	Load as element import to existing package
Shared and exists as shadow package	Load as owned shadow package

Another choice, which is not displayed as a default, is the option to simply absorb the unit into the model resource (absorb into model). Alternatively, you can import the model as a standalone model, although this is not a default choice.

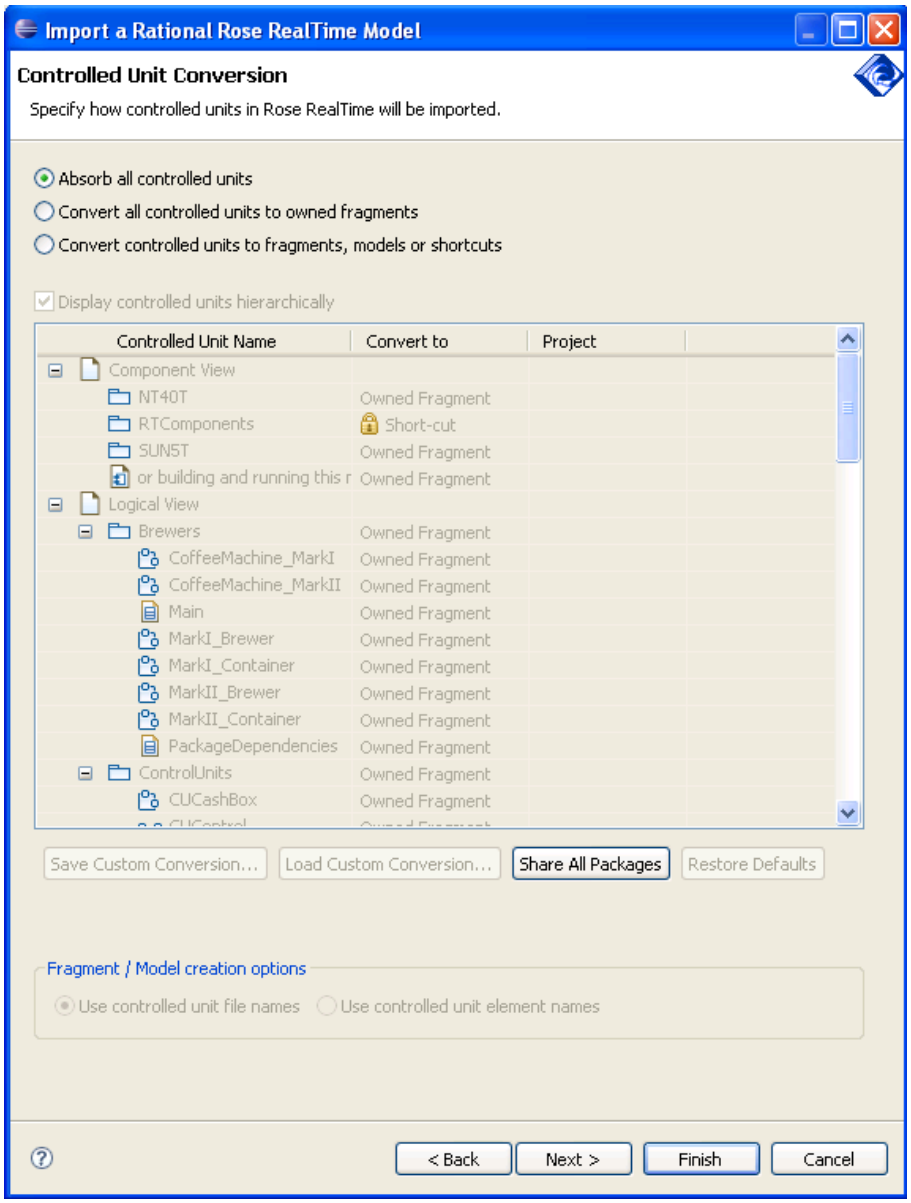
The controlled unit page of the Rational Rose RealTime model import is beneficial for model hierarchies that have inconsistent usage of the "Owned by model" property on controlled units. For instance, when you set the controlled unit package to "shared" in Rose RT you cannot do several fundamental actions that allow you to manage the unit. Further, the source control system (such as Rational ClearCase) may own the unit; therefore, only certain users may modify the controlled units. When source control systems own units, it is not important how the ownership is specified in the tooling because the ownership is defined by the source control system.

Controlled Unit Conversion Page:

Considering the modes of conversion above, the dialog needs to accommodate these as well as provide flexibility to provide blanket conversion for all elements. Common methods of importing may be to ignore all fragments for testing purposes in which case all units will be absorbed into the model. Likewise, the practitioner may simply want all fragments to be owned which-out concern for their shared properties. This is accomplished in the UI using a radio-button in the dialog for the three modes of import.

Absorb all controlled units mode

If you want to merge all controlled units with the owning model, so that the controlled units no longer reside in separate files, select the first radio button that appears on this page. All other controls in this page are disabled.

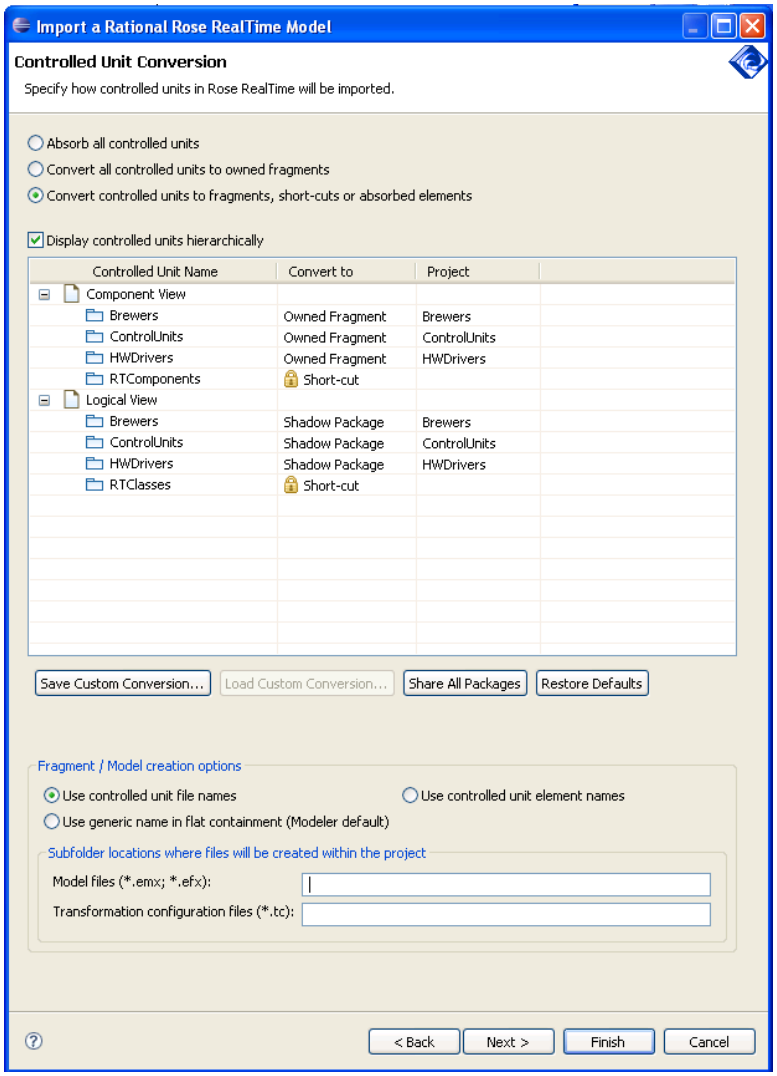


Convert all controlled units to owned fragments mode

If you want to import all controlled units as fragments that the imported model owns, select the **Convert all controlled units to owned fragments** radio button. In this case, the custom unit UI is disabled; however, the "fragment / model creation options" are enabled and allow you to determine the default naming scheme for the new fragments.

Convert controlled units to fragments, models or short-cuts

In this final mode of operation (Convert controlled units to fragments, models or short-cuts), you can control how each unit is converted on an individual basis. By default, the units will be displayed in a hierarchical manner in a tree that can be collapsed if needed. The "Convert to" column allows you to select the conversion method for the individual unit. If you are unsure of the true ownership of the packages in the model, the safest approach is to click the "Share All Packages" button under the list control. This option converts all owned fragment packages into Shadow Packages or maintain them as "Short-cut" links if the original package has already been imported into the workspace.

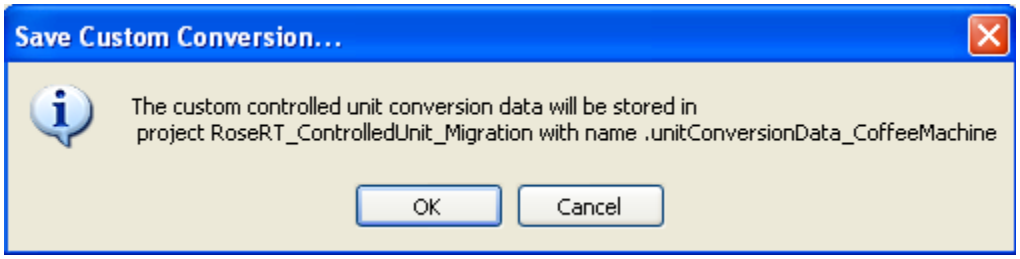


Persisting custom fragment conversion settings

For complex models with many controlled units, it may be difficult to decide how each controlled unit migrates into RSARTE. You may need considerable time to decide what units need to become models and/or whether something should be a shadow package. The defaults attempt to reduce this overhead; however you must inevitably perform analysis so you can figure out the best migration approach

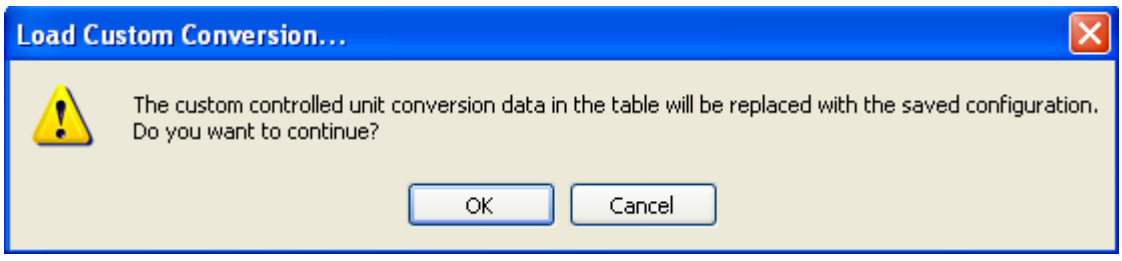
To save the settings for how to import controlled units, you may want to click **Save Custom Conversion**. During the experimentation phase, you can use the **Save Custom Conversion** functionality to avoid modifying the conversion method settings for each controlled unit multiple times. This functionality is particularly helpful when you experiment with migrating controlled units in very large models.

When you click the "Save Custom Conversion…" button, the current settings in the custom conversion table are saved to a file in the workspace. A message box will appear indicating the location of the saved configuration file.



If the import wizard detects a custom conversion configuration file in the workspace, you can click **Load Custom Conversion** to restore the saved settings for importing controlled units.

The "Load Custom Conversion" button is disabled unless a custom conversion configuration file exists in the workspace at the location specified above. When enabled, you can click this button to load the current settings into the dialog. A warning dialog will appear to ensure that you want to perform the operation.

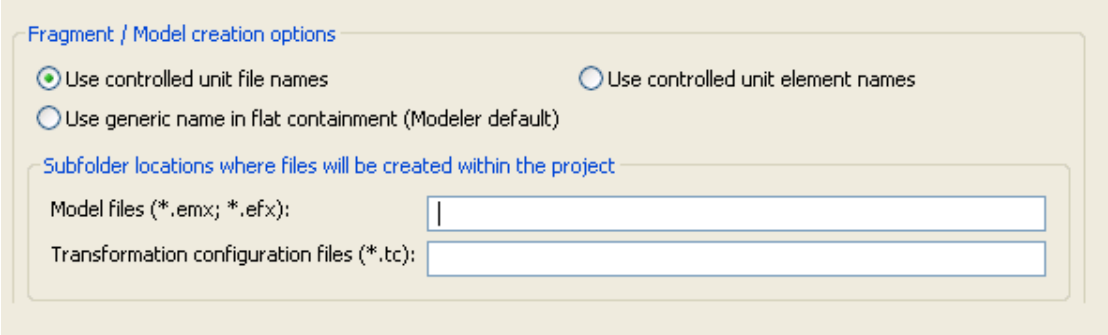


Allow conversion to independent models or root packages

It can be useful, since fragments are not sharable entities in Rational Software Architect RealTime Edition, to allow for sharing of a fragment if in the Enterprise's set of models there is no clear owner of a particular fragment. For instance in RoseRT, the owner of a fragment is determined conceptually by a Boolean property on the fragment which determines if it is owned by the model or has been shared. If this property is not clearly defined or is set as "owned" in multiple models, it may make sense to have this unit be pulled out into its own model that it is a distinct sharable entity. Otherwise, it may be shared in the context of another model imported from RoseRT that has other packages that haven't been designed to be shared in other models.

Creation options for new models and fragments

Finally there are some global settings for how individual fragments and models are created during the import. These options are for how fragments are named and their location in the file system. You may wish to use the import as an opportunity to consolidate the fragment naming based on the logical name of the unit in the model. This could be useful if the controlled unit names in the file system have become out of sync with the corresponding logical name over time.



The first two radio buttons control this behavior. The default setting ("Use controlled unit file names") will continue to use the controlled unit names in Rose RealTime and translate them to the RSARTE fragment file extension (.efx).

Rose RealTime organized controlled units in a similar fashion to how the logical model is organized. If a particular element was contained deep in a package hierarchy, then the corresponding controlled unit would typically be stored in a similar containment hierarchy on the file system. This organization is convenient if the user needs to locate particular controlled units and be able to "mind-map" them back to the corresponding element in the model. This paradigm is brittle when considering cases where the model element is refactored by either renaming it or moving it to a new location. The controlled unit must be either similarly renamed or moved to correspond with the logical element change. These can be expensive CM operations and can result in loss of file history in some systems (i.e. CVS). Modeler has accommodated these concerns by adopting an element agnostic naming approach for fragments. This means that fragments are named generically and have a flat containment relative to their parent fragment. Users can override this behavior if desired, but the default lets the fragments be resilient to refactoring use-cases.

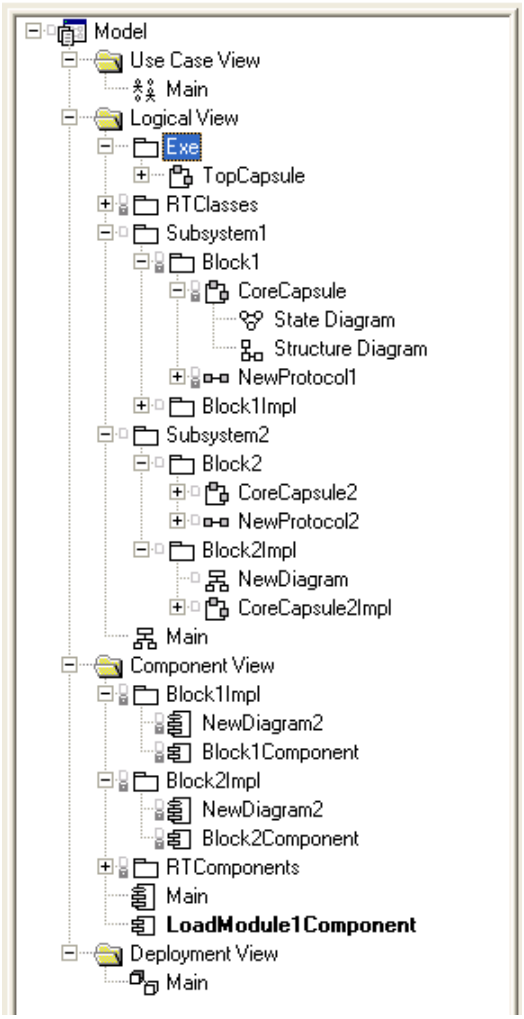
The first two radio box options ("Use controlled unit file names" / "Use controlled unit element names") will respect the Rose RealTime controlled unit organization. The fragments will be created in a similar containment hierarchy and named according to either the controlled unit filename or the corresponding logical element. This may be familiar to users but once they start using the tool and creating new fragments they are faced with inconsistencies unless they explicitly locate the new fragment and override the default Modeler behavior as described above. To support the Modeler behavior there is a third option "Use generic name in flat containment (Modeler default)" which supports a mode where the user can choose to import the controlled units using the same flat containment and generic naming that Modeler uses for its default behavior. In addition, they can specify a subfolder where the fragments are located as opposed to having them created flat in the owning project.

Example Import

A load module has packages that are shared by teams and are also owned packages that other teams utilize. Consequently, they want to keep most packages set as "shadow" packages initially.

For example, the RoseRT model below represents a load module executable that is ready to be migrated.

Example RoseRT Model



Assumptions

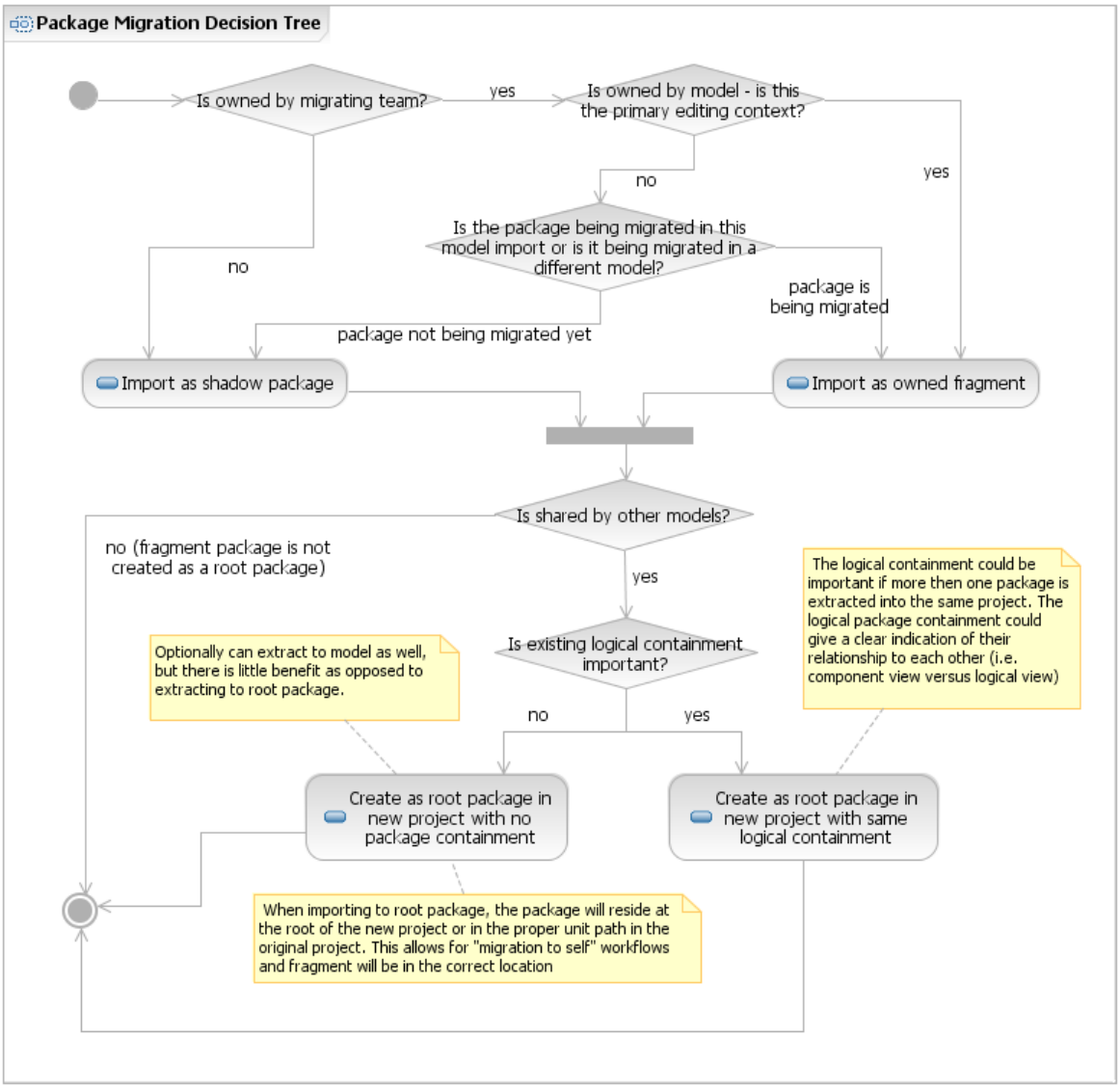
- Subsystem1 is owned by the team that is migrating loadModule1
- Subsystem1::Block1 (interface package) is shared by another team that isn't migrating yet
- Subsystem1::Block1Impl package is owned.
- Subsystem2 is shared from another model

Given these assumptions, also assume that the loadModule1 team has a configuration management (CM) structure around how they control and manage the versions. Therefore, they want to maintain the CM structure of how Subsystem1 (which is owned) after migration into RSARTE. However, parts of Subsystem1are shared by outside teams, so how it is separated with respect to projects needs to be considered as well. Subsystem2 is managed and owned by outside teams, so we don't need to care about its CM structure.

Decisions

When you decide how to convert package fragments, you must make some important decisions on how they are shared, who they are owned by, and the whether the team that owns it is ready to migrate. Each package controlled unit will be migrated either as a shadow package or an owned fragment. In turn, the package may be extracted to another project as a root package or model depending on whether it is necessary to share it or edit it independently.

The following activity diagram represents this decision tree:



Take a look at the example from above and make some decisions around how you import it. In this example, you want to maintain the containment structure of the controlled units because there are many scripts that support versioning and releasing of the model at the load module context. To do this and support sharing of these units, the controlled units must be separated into their projects.

First considering "Subsystem1", since you own it, that means it should be an "owned fragment". Following the decision tree, you know it is owned but you also know this package is a container only and isn't shared by itself. Therefore, it can

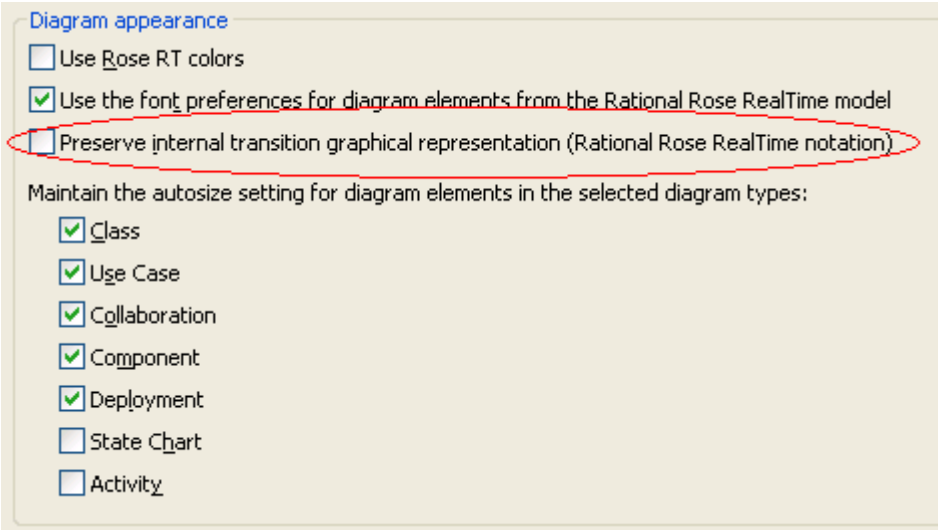
be kept as owned in the same project as the original owning model. The sub-package contents in "Subsystem1" are less straightforward though. "Block1" is owned but shared by other team projects, so it will need to be extracted as a root shadow package in a new project. "Block1Impl" is also owned, but the owning model isn't the primary editing context. In addition, you know it isn't shared by other models in or outside of the migrating team. However, we want to do the official migration of this package during this import as opposed to its "owning" model (i.e. Block model). So, it also will be extracted to a root into the same project as the "Block1" fragment package except as an owned fragment. The corresponding packages in the "Component View" would be extracted in the same way as their "Logical View" counterparts. For these packages, it would make sense for them to retain their logical package containment to differentiate them from the "Logical View" packages.

The "Subsystem2" package would be imported as a shadow package. The decision to make it a root package would depend on whether other load modules existed that also depended on this subsystem. If this was the case, it would make sense to extract it into a separate project.

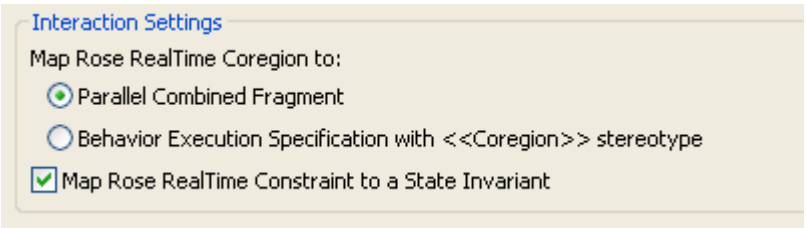
Diagram Appearance migration

Since the underlying model representation has changed from UML 1.x to UML 2.0, there are also associated notation differences that have changed. However, the goal is to try and preserve the diagram notation as much as possible, considering the notational differences. For instance, in UML2, internal transitions are no longer represented as self-transitions on the state frame. Instead, they appear in a separate compartment as a list that can be optionally filtered out. While it is important to support this notation, at the same time, we have to consider the time and effort users spent to organize and annotate their diagrams with internal transitions in Rose RT. The diagram organization may be familiar or have a specific spatial format that helps you locate a particular internal transition in an efficient way. Considering how you organize your diagrams, RSARTE supports an option that displays internal transitions as they appear in Rose RT.

Diagram appearance section in the RoseRT Import Wizard



There are also differences in the semantics for sequence diagrams that will appear notationally different. For instance, co-regions in Rose RealTime don't have a direct semantic mapping. The closest mapping is a Parallel Combined Fragment in UML2 and the notational representation is significantly different then the co-region bars in Rose RealTime. If you desire a similar notation mapping, then the wizard does provide an option to maintain this. The option "Behavior Execution Specification with <<Coregion>> stereotype" will maintain a similar notation to Rose RealTime however, the semantics are assumed to be derived from the applied <<Coregion>> stereotype.



The "Map Rose RealTime Constraint to a State Invariant" is a convenient feature if your sequence diagrams have lots of floating constraints. This options will detect constraints that have proximity to a lifeline and/or are attached to the lifeline and will convert them to State Invariant elements which are on the lifeline. This is convenient especially when converting coregions to PAR fragments since the layout will change and these floating constraints won't move with the layout changes. Since a State Invariant is a constraint, it is a useful way to associate a constraint with a position on the lifeline. The State Invariant being on the lifeline will move relative to the other items on the lifeline and as such is resiliant to layout changes.

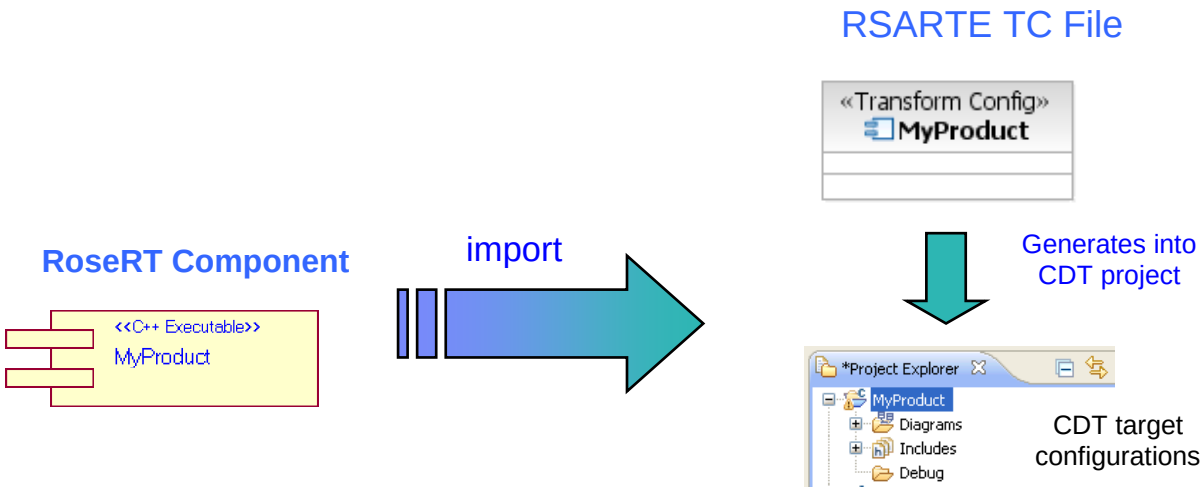
In addition from the UML2 differences, there are also tooling differences such as the color scheme, fonts and how elements are sized. The recommended approach is to adopt the new RSARTE color scheme because these colors will be consistent with the newly created models and other existing models. However, there is a **Use Rose RT colors** option that maintains the Rose RT color scheme if there are constraints or consistency issues with other published models. The diagram-based auto-size settings are important because shapes have different margins and compartment sizes between the each tool. These differences may make the shape cut off compartment contents if forced to be the same size. When the auto-size settings are maintained, then the auto-size functionality of RSARTE will accommodate the local compartment margins and make the shape look appropriate in the new tooling context. By default, the auto-size option is turned off for State and Activity diagrams because diagram layout and spacing is more of a concern. In addition, shape size is more critical to the overall diagram esthetic. The recommended approach is to keep the defaults and observe the results before making changes to the diagram element auto-size settings.

Component migration

Components in Rational Rose RealTime describe how to generate and build the model into an executable or library artifact. RSARTA is built on top of a C++ IDE (CDT) as well as a model transformation framework that is part of Rational Software Architect (RSA). Consequently it makes sense to integrate into those frameworks in order to harness their existing functionality, extensibility and known workflows. In migration into RSARTE, the component migrates into two separate entities:

- A transformation configuration file which is responsible for model to C++ code generation
- The CDT project it generates into which contains the compilation settings

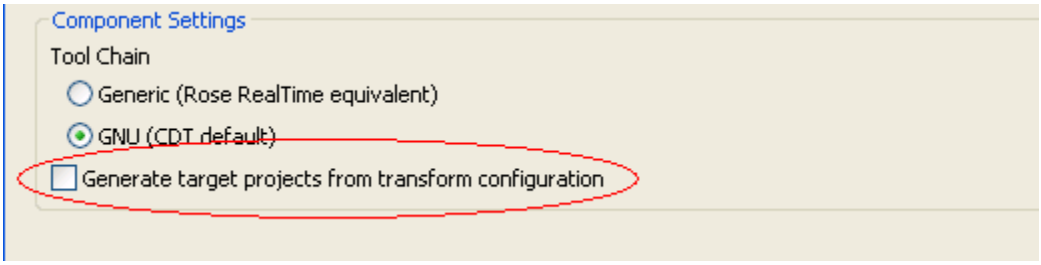
During import, this conversion is handled automatically and the components that used to exist in the component view in RoseRT are converted into short cuts to the new transformation configuration files (TC).



CDT Project Generation

Since the compilation and execution settings are stored in the CDT project, this could represent more overhead in large Enterprises. When you manage these artifacts in your source control repository, both the TC file and the CDT project need to be checked in to preserve the settings. There is an option to provide the ability to store the CDT configuration properties (execution and compilation) in the TC and dynamically create the CDT project at transformation time if the project doesn't exist. This allows for only the TC file to be checked in to the source repository and have the CDT projects local to your workspace. This behavior is optional and can be set during import. This option must be unchecked to allow the CDT projects to be master of their configuration settings (and therefore must be added to source control).

Component Settings section in the RoseRT Import Wizard

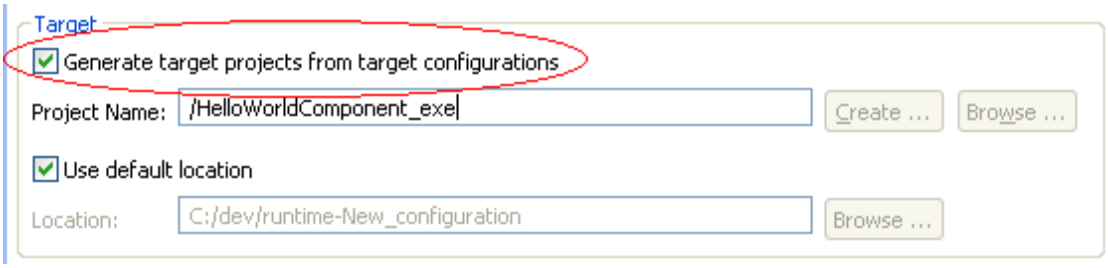


If you select the "Generate target projects from transform configuration" checkbox, then the imported TC files are set as master for the target configuration properties. When those properties are changed, the corresponding CDT configuration is updated

After import, this setting can be re-adjusted for individual TC files in the transformation configuration editor. On the "Sources and Target" page the same checkbox exist and will be populated with the value set in the import. If the checkbox is selected, then the "Target Configurations" tab is enabled so that the target configuration can be changed and it will be regenerated into the CDT

project on the next transformation. If cleared, then the "Target Configurations" tab will be disabled and the CDT project becomes the "master" for the compilation configuration. Consequently in this case, the CDT project should be checked into source control.

Target properties in the Transformation Configuration Editor



Tool Chain

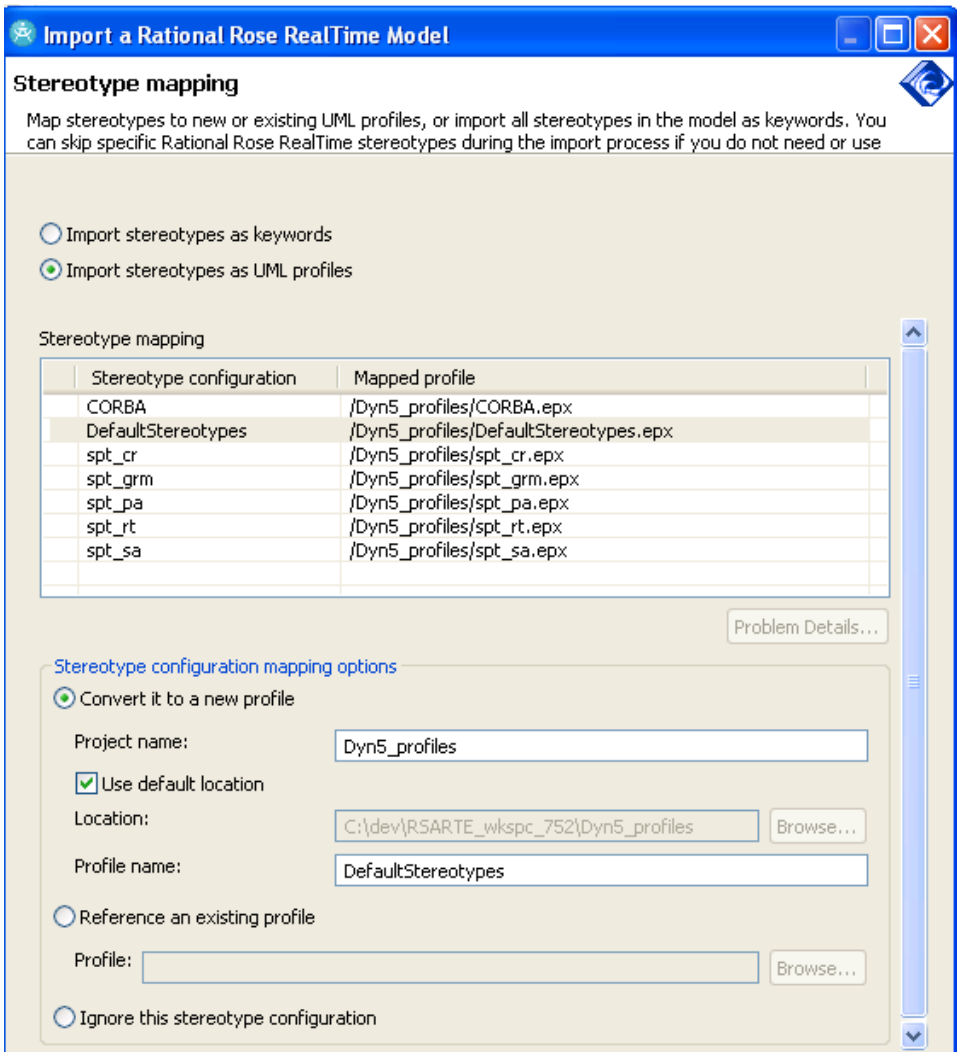
The CDT has a concept of a "Tool Chain" which represents the set of tools which together generate the makefile, build, link, launch and debug the executable. This allows the CDT to be open to almost any set of tools provided by different vendors and have them integrate into the Eclipse environment. However, defining a new "Tool Chain" in the CDT is a fairly non-trivial prospect that requires in-depth knowledge of the CDT extensibility. Consequently, the CDT supports some default "Tool Chains" ("Cygwin" and "MinGW") that represent known standards and are widely supported. Additional "Tool Chains" are contributed by 3rd party vendors. The RSARTE tooling supports the "GNU" or ""Cgywin" tool chain on import and will do it's best to convert the component data into compatible configuration data for the CDT. Sometimes, the target that is being supported has different makefile syntax or other incompatibles that make in not possible to use the "Cygwin" Tool Chain. In this case, the wizard offers a "RoseRT compatible" mode that utilizes a new "Tool Chain" contributed by RSARTE that will generate the makefiles in the same way as RoseRT. If you are unsure about which one to choose, the "Generic" toolchain (RoseRT Compatible) is the default and can be relied upon to give the best results. See section #Component Settings section in the RoseRT Import Wizard to see where this option can be set in the wizard.

Stereotype Mapping

Stereotypes are a way of categorizing elements for a particular domain or specialization. In UML2, these stereotypes can be imported in the context of an actual profile which contains a set of stereotypes or merely as a keyword which is a textual way of identifying an element. UML2 has a more robust implementation for stereotypes by allowing them to be grouped together into a profile and letting stereotypes define a set of properties. If the more advanced capabilities of UML2 profiles and stereotype properties are of no interest, then simply importing "Stereotypes as keywords" should be adequate.

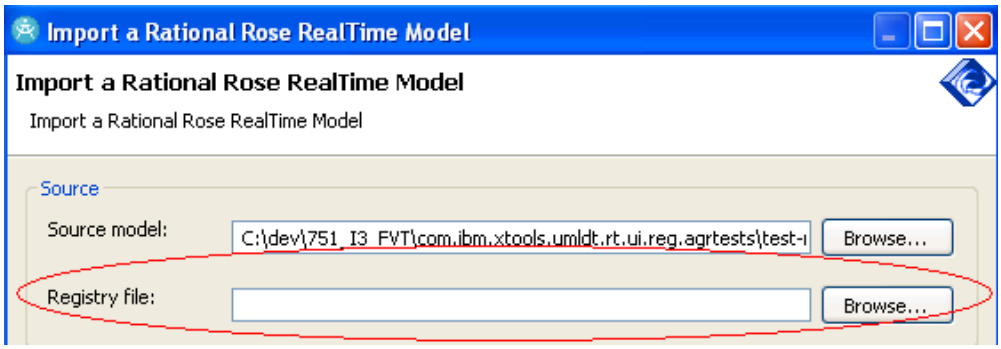
If the benefits of the UML2 profiles look useful, then each stereotype configuration needs to be mapped to a specific profile. Each entry in the table represents a stereotype configuration file, not just a single stereotype. These stereotype configurations are defined explicitly in an *ini* file and can be accessed through RoseRT to be applied to model elements. There are also model defined stereotypes, which don't have a stereotype configuration and are simply keywords on model elements. Each detected stereotype configuration can be selected and then the "Stereotype configuration mapping options" will change to reflect the selection. The stereotype configuration can be converted into a new profile, an existing profile or ignored in which case it will be imported as a keyword.

Stereotype mapping Import wizard page



In RoseRT, the stereotype configurations are stored in the registry. On non-Windows platforms this registry needs to be exported and specified on the first page of the import wizard. Otherwise, the stereotype configurations won't be detected and any elements with stereotypes on them import as elements with keywords.

Specifying a Registry File



Here are the instructions for exporting a registry file (Linux / Solaris platforms):

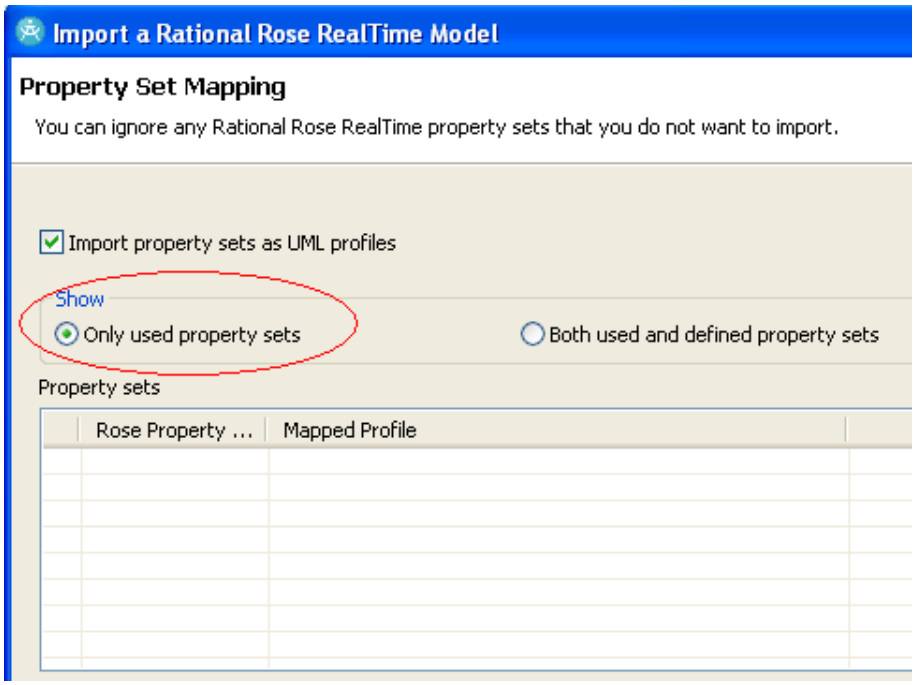
1) Start RoseRT with the following option: "RoseRT -regedit" : This actually starts the regedit application(provided by mainwin) instead of RoseRT app. RoseRT/MainWin will load all the different registry files into the application. This is how the Rose/RoseRT users modify the registry contents and export them.
2) Export the complete registry as a single file: To do this execute **File->Export Registry File**". This creates a single flat ".reg" file.
3) Use the files created in step-2 as input to the Importer (#Specifying a Registry File).

Property Set Mapping

A property sets is a mechanism in RoseRT that allows you to extend the model using domain-specific properties. Unless you've defined custom property sets in RoseRT, this wizard page can be ignored. All the default property sets used for C++ code generation (C++ Generation, Compilation, Library, Executable, External Library, etc) are automatically migrated regardless of the choices on this page. A useful way to check this is to click the "Import property sets as UML profiles" and ensure "Only used property sets" is selected. If no property sets show up in the list, then there is no need to migrate any existing property sets since they aren't referenced in the model at all.

Tip: Some Rational Rose RealTime models may contain property sets that are not useful after import. Examine the property sets and identify any that you can ignore during the import process. Skipping unwanted property sets reduces the size of the imported model and reduces clutter in the Navigator view.

Property Set Mapping Import Wizard Page



If property sets are detected during the import process, then the migration process is similar to the stereotype mapping page. As with stereotypes, each property set must be mapped to a particular profile. If a property set is mapped to a new profile, and you select the "Import as property set profile" is chosen, then the "Language" and "Group" of the property need to be specified. Otherwise, it will be created as a regular UML stereotype profile.

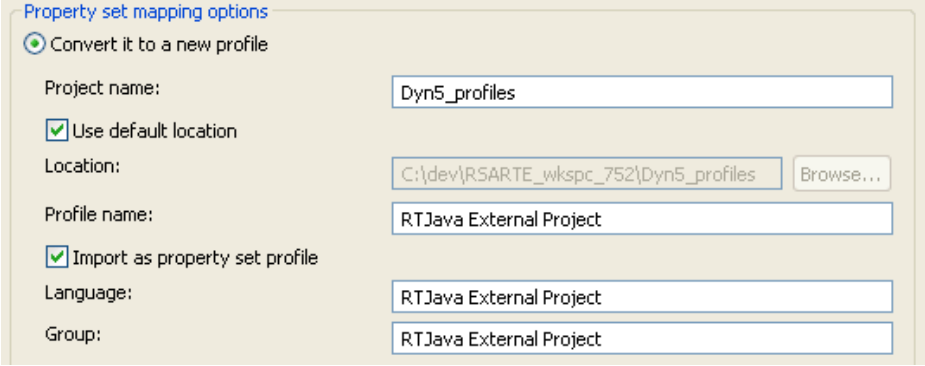
The difference between importing as a property set profile and importing as a regular profile is kind of subtle - maybe the best way to describe it here would be to say something like "This option creates a profile that can be used in a way that is more similar to Rational Rose RealTime property sets". Here are a couple of guidelines for when to choose to import as a property set profile:

- 1) If you want to be able to view and edit values for these properties using a UI similar to the Rose RT UI for property sets, then you must choose to import as a property set profile. Otherwise, these properties can only be edited in the same way that the properties of regular stereotypes are.
- 2) If you want to be able to set default values for these properties at the model level (the way that you can for property sets in Rose RT), then you must choose to import as a property set profile.
- If in doubt, just accept the default setting (which is to import as a property set profile).
- It may be worth emphasizing that you will *not* lose any data by choosing one way over the other.

The "Group" and "Language" of a profile are used to display property sets on the RT Properties tab of the Properties view:

- The language of a property set profile indicates the domain that the property sets apply to; in RSD, when you view all of a model's property sets in the Properties view, they are grouped together by language.
- The group of a property set profile is used to indicate subcategories within property sets for the same domain; for example, you could have two property set profiles that both have the language "Java", but one could represent a "Basic" group of properties, and the other could contain "Advanced" properties. The properties from each group are then viewed separately in the UI.

Property set mapping options



Post import migration

In a small software project where a single model is independent of other models, you can consider the migration complete once the model is compiled and runs in the new environment. In large Enterprise organizations, this case is extremely rare. Most models are very large and need to be maintained by developers in smaller sub-component models. These models often depend on models developed by other teams and/or have sub-components that are consumed by other teams. These scenarios make the migration process more complex because parts of the model may need to be separated into their own projects to allow for sharing. When multiple teams are involved, the migration process may need to be staggered to allow one team to migrate before another team. If this is the case, then there is a need to support linkages between components in RSARTE to RoseRT. This is accommodated through shadow packages which can be synchronized from a corresponding RoseRT package. These packages are volatile by nature and will eventually migrate as normal packages or they will be removed and have all their references changed to a migrated package. Therefore, there is a need to support management of the migration process after import of a single model for synchronization, refactor for reusability and finally remove shadow packages as other models come online in RSARTE.

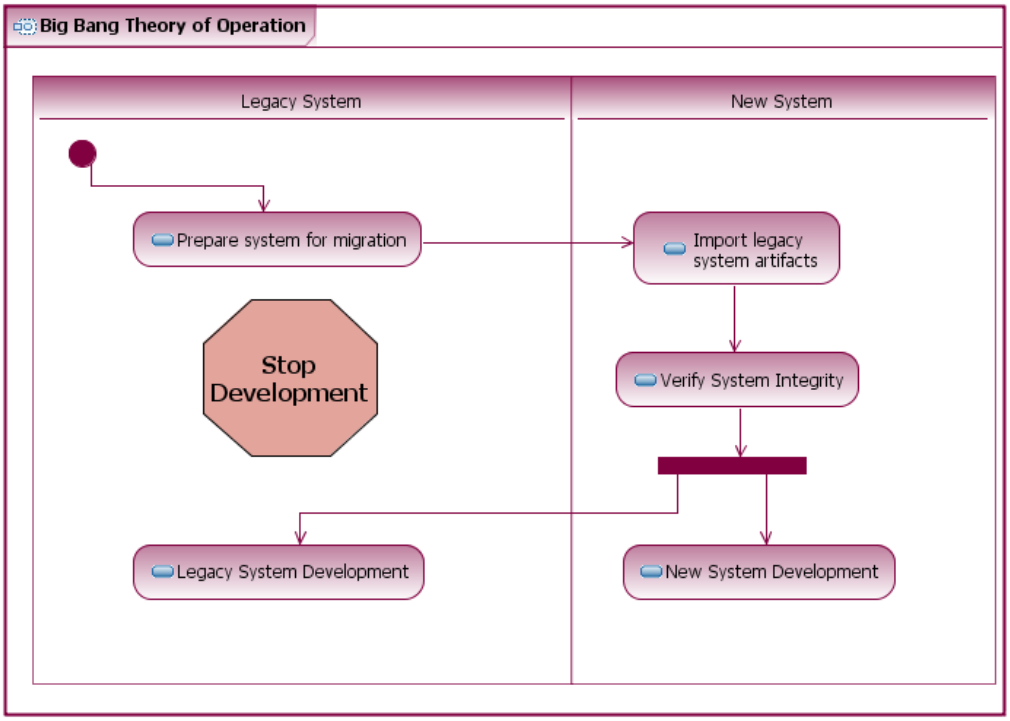
Incremental Migration

Theory

Software tooling and development environments are evolutionary in that they typically change over time. These changes can be minor which don't require any considerable effect, or they can be major which causes the file format schema to change and consequently precipitates a major migration effect to move the legacy tooling model artifacts to the new tooling. As observed in the technology map - #Technology mapping from Rose RealTime to RSARTE, all aspects of the underlying tooling and file format have changed in this migration. The mapping from the UML 1.4 to 2.1 specifications are different enough that there is no backwards compatibility; therefore, you must perform a concentrated import in order for the UML 1.4 artifacts to be converted to UML 2.01.

One way to tackle this incompatibility is called a "Big-Bang" theory of operation. This means simply that the legacy tooling (RoseRT) is shut-down and migrated all at once to the new tooling (RSARTE) using tools that convert any relevant artifacts to the formats understood by the new tooling. Realistically though, the legacy tooling is usually kept in production for maintenance purposes and is still required while verification of the new tooling continues.

Figure I: Big Bang Migration

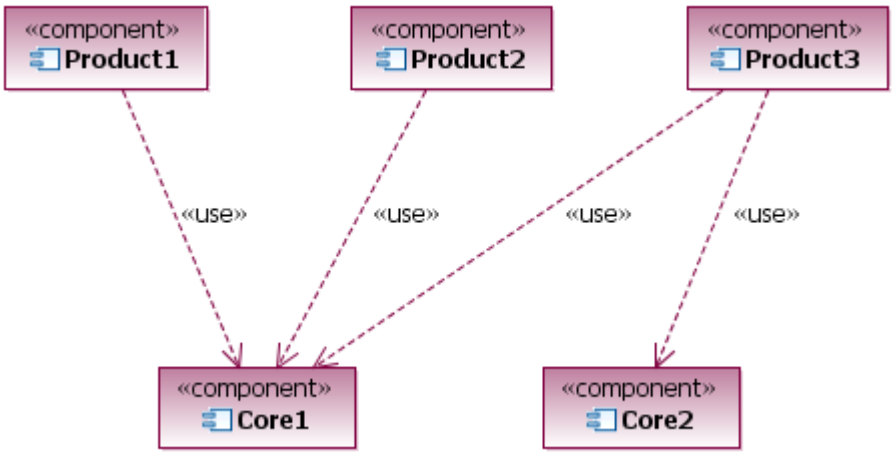


If we examine the activity diagram above, it describes two partitions of workflow: one for RoseRT and one for RSARTE. First, the legacy tooling data is prepared for the migration which may entail some clean-up or refactoring then it is imported into the new tooling. From there the content is usually verified and tested to ensure model integrity before it can be brought back into production. During this time, the legacy tooling is shut-down and not available. After the migration, the legacy may be brought back up for read-only access or to support data streams

not being migrated to the new tooling. This tooling stoppage can be costly to an organization since it implies they aren't developing their models during this time.

System software architecture is usually divided into components that represent different aspects or functionality within the tooling. The components will depend on each other in a layered fashion where the core components are at the bottom of the dependency chain and the leaf or product components are at the top. The core components by their nature are reusable across different product level components and are critical to the execution of the different models. Different product components may have different release cycles that require them to have schedules that aren't in sync. Since they may depend on the same core components, one product stack may be ready to migrate to the new tooling, but other product stacks may not be because of schedule or release concerns. This means that the core components are by nature synchronized with the slowest moving product stack since they have to support all dependent components above them. Consequently the core components wouldn't be ready to migrate at the same time as the more progressive product components at the top of the dependency chain.

Figure II: Product component stacks

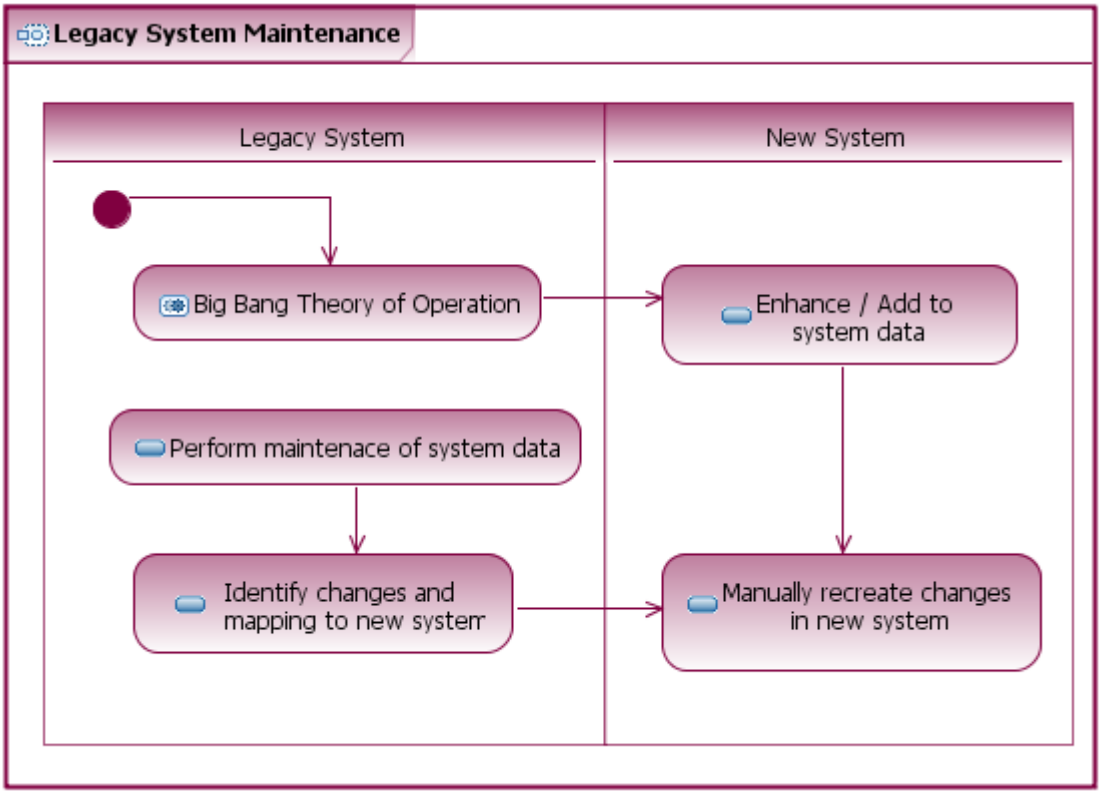


In the above example, all the product (leaf) components depend on "Core1". If "Product3" is ready to migrate, but "Product1" is not, since they are both dependent on "Core1" then "Product3" must wait for "Product1" to be an appropriate stage in order for migration to proceed.

It is perhaps naïve to think that once the migration to the new tooling is complete, the legacy tooling will no longer be needed. In an ideal situation, the data can be 100% migrated and there is no longer a need to support the data or a subset of data on the legacy tooling. This will probably not happen often. If the legacy tooling is supporting a particular release of software, then it would need to support that release for its lifecycle. It would be too risky to release the software from the legacy tooling and immediately migrate it to the new tooling and subsequently support bug fixes. Issues from the field would not map directly into the new tooling and migrating data in a fix pack that is supposed to address particular issues would be foolhardy at best. So this implies that the legacy tooling and new tooling need co-exist for a period of time (and it could be considerable amount of time depending on release schedules). Fixes or changes in data / software on the legacy side would need to propagate into the new

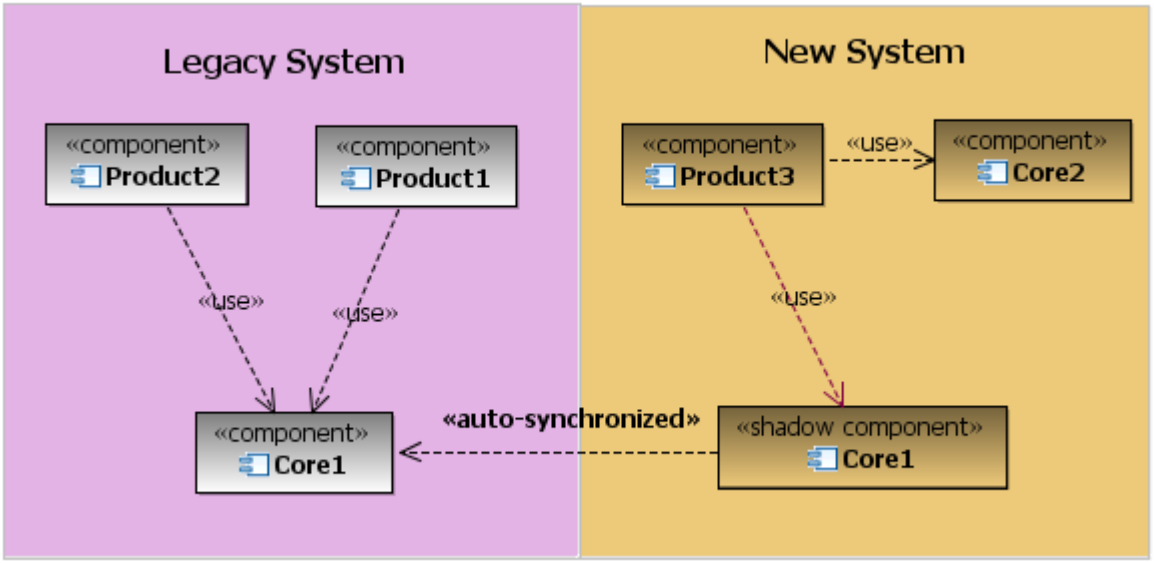
tooling. If the differences between the data structure are considerable between the two systems, then a file system merge is not sufficient. Generally, these changes would need to be integrated manually through code or data inspection.

Figure III: Legacy Tooling Maintenance



To address these concerns, we need to provide a way to do incremental migration of the different sub-components within a model. This means that we need to introduce the concept of "mastership" where a package can exist in the "RSARTE", but is actually still mastered in the "RoseRT". By allowing a sub-package to exist as a read-only ghost or shadow in the new tooling, and still be mastered / edited in the legacy tooling, we can provide a one-way bridge so that the two models can interact. Then if the synchronization of the legacy tooling package is managed so that the ghost package is automatically updated when changes are made, there is little to no maintenance that you need to do to create the bridge between the two models.

Figure IV: Migration of single component using auto-synchronization

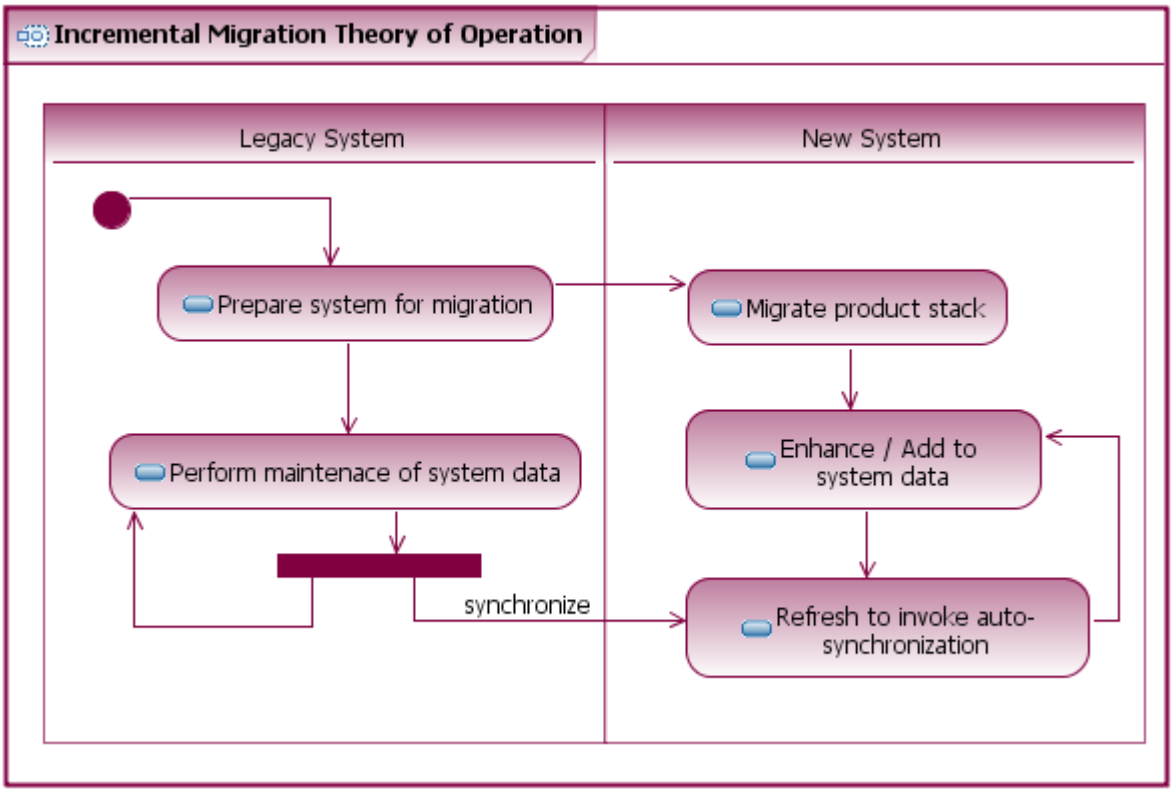


Considering the example from *Figure II* it is now possible to migrate only the "Product3" component into RSARTE by creating a shadow component to "Core1". "Core1" is still mastered in RoseRT and has no knowledge of the shadow component which exists in the new tooling. When changes occur to "Core1" in the legacy tooling, this invokes an auto-synchronization where the component is re-imported into the new tooling transparently.

Therefore, the "Big-bang migration" is no longer necessary since individual components can be migrated as needed. Based on release cycles or stability evaluation, specific components can be chosen to migrate given some criteria. At any given time during the incremental migration, the legacy tooling is still functional and can continue to be maintained. Auto-synchronization between the shadow components and their master ensures that changes to the legacy tooling are propagated into the new tooling.

Auto-synchronization implies that there is a discoverable mapping from the legacy format into the new tooling. To facilitate this, often assumptions will be made which are acceptable in the process of the migration. This allows for a transparent import between the two tooling environments without any user intervention or prompting. In fact, this is a pre-requisite for this paradigm to be operational. It is also possible to perform a manual synchronization from a legacy component to a fully migrated component. In this case, the migrated component may have been editing in the context of the new tooling. The synchronization will then need to merge the changes made on the legacy side into the new tooling instead of merely replacing it. This merge will invoke some UI to resolve any conflicts similar to a team based scenario where two different developers modify the same source file.

Figure V: Incremental Migration Theory of Operation



Refactoring

In software development, architecture is rarely a static entity. As projects scale larger, the requirements change related to how components get consumed which then requires new projects to be created and/or make project dependencies change. Or it's possible that it wasn't clear at import time how certain packages would be shared in the new Eclipse based modeling environment. To accommodate this, RSARTE has a rich set of refactoring capabilities that can be performed at a fragment level. New fragments can be created or existing ones absorbed and they can be moved to different projects entirely. In addition, packages that exist in the context of a model can be extracted into a "root package" that can be opened by itself, independently of any owning model. This is a powerful feature for sharing of individual packages. These root packages can then be moved (or extracted to a specific location) into a new project that can be brought into new workspaces as part of a different project set permutation.

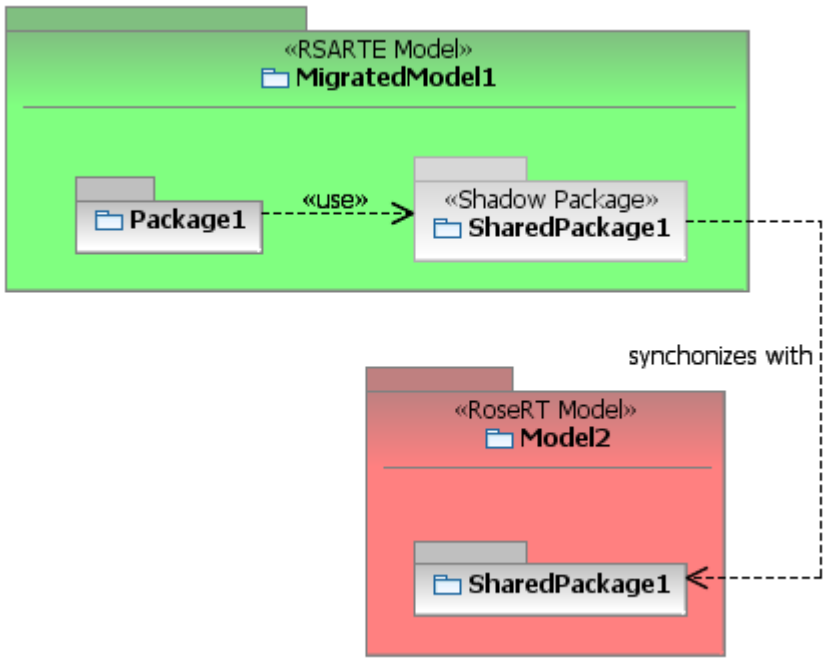
This is all very well for regular owned packages, but shadow packages are not modifiable by design so that changes in RoseRT are synchronized one way into the shadow package. Hence the default refactoring functions won't operate on them. The caveat is that synchronization relies on the package unit remaining intact in RoseRT which means refactoring will potentially break the synchronization link to the shadow package. To get around this, a specific refactoring Extract to Top-Level Package command is supported. This command – can operate on shadow packages and allows them to be made a root package and moved into a different project at the same time.

Migration Scenarios

A model shares a package from another model that isn't migrated

Since teams have different timelines that define when they can adopt new development process and tooling, one team may be ready to migrate when another isn't. When a model is imported that has dependencies to a model that isn't in the workspace, the package that is shared will be set as a "Shadow Package". If it isn't (for example, the "owned" attribute not representative), then you may change it to be a "Shadow Package" in the "Controlled Unit Conversion" page. When the original RoseRT package is modified, the "Shadow Package" can be resynchronized to bring in those changes.

Package structure of model with Shadow Package

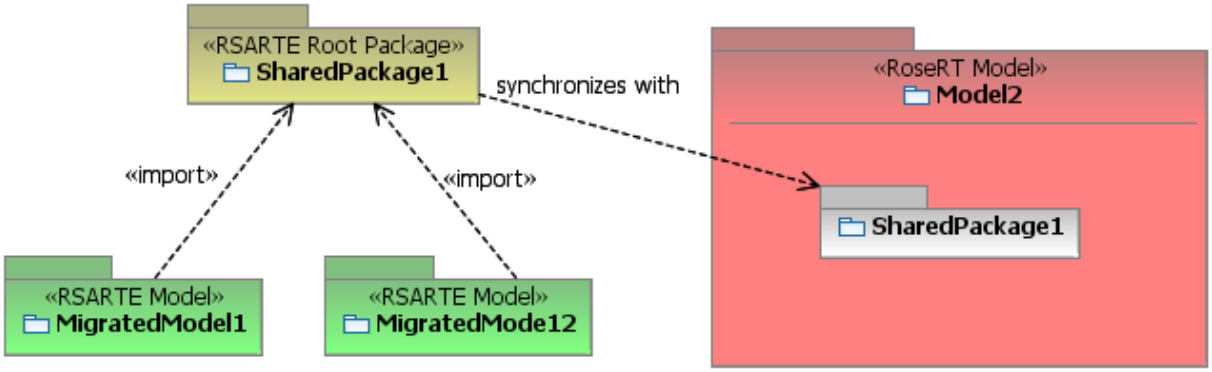


At some point, the team that owns the package being depended upon may be ready to migrate their model. Once migrated, the dependent team will bring those projects of the newly migrated model into their workspace. Next, the team can migrate the "Shadow Package" to the actual package brought into their workspace. All references to the "Shadow Package" and its contents adjusts to point to the actual package and the "Shadow Package" disappears and is replace by an element import in its place.

A package in a model that isn't migrated is shared by multiple models that are migrated

The same situation is magnified where multiple models share the same package that hasn't been imported yet. The workflow is the same as #A model shares a package from another model that isn't migrated above. However, when subsequent models are migrated the same package will be a "Shadow Package" in multiple models. This is obviously redundant and wastes memory, disk space and requires extra work to keep each of the "Shadow Package" synchronized separately. To avoid this, the originally import "Shadow Package" can be extracted out into its own project as a root package. This can be done at import time or by using the refactoring command "Extract to Top-Level". When the other models that share the package are imported, the packages should be imported as a "Shadow Package". To avoid the redundancy, these "Shadow Packages" can be migrated to the original "Shadow Package" such that all references are replaced and the packages removed with an "Element Import" relationship in their place.

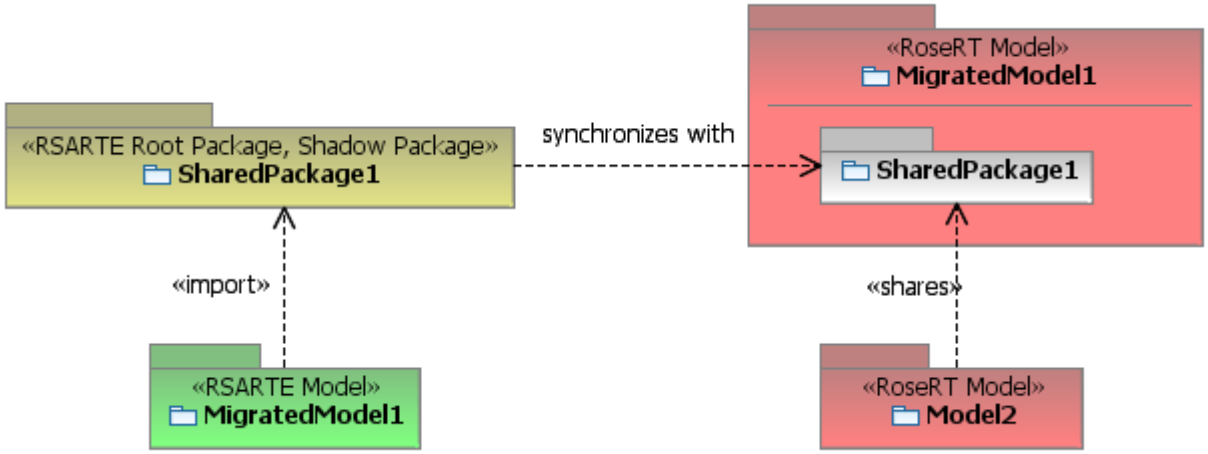
Package structure of multiple models element import same Shadow Package



A model owns a package that is shared by other models that aren't ready to migrate

Another scenario that is similar but has different ramifications is when a team has a model that has a shared owned package unit with other models not owned by that team. In this case, the team would like to migrate the package, but if they do, they need to maintain two versions of the package for the dependent teams (one in RoseRT and one in RSARTE). Alternatively, they can make the package a "Shadow Package" until dependent teams are ready to migrate themselves. This allows them to change the package only in RoseRT.

Package structure of model that owns a package shared by models not migrated yet.

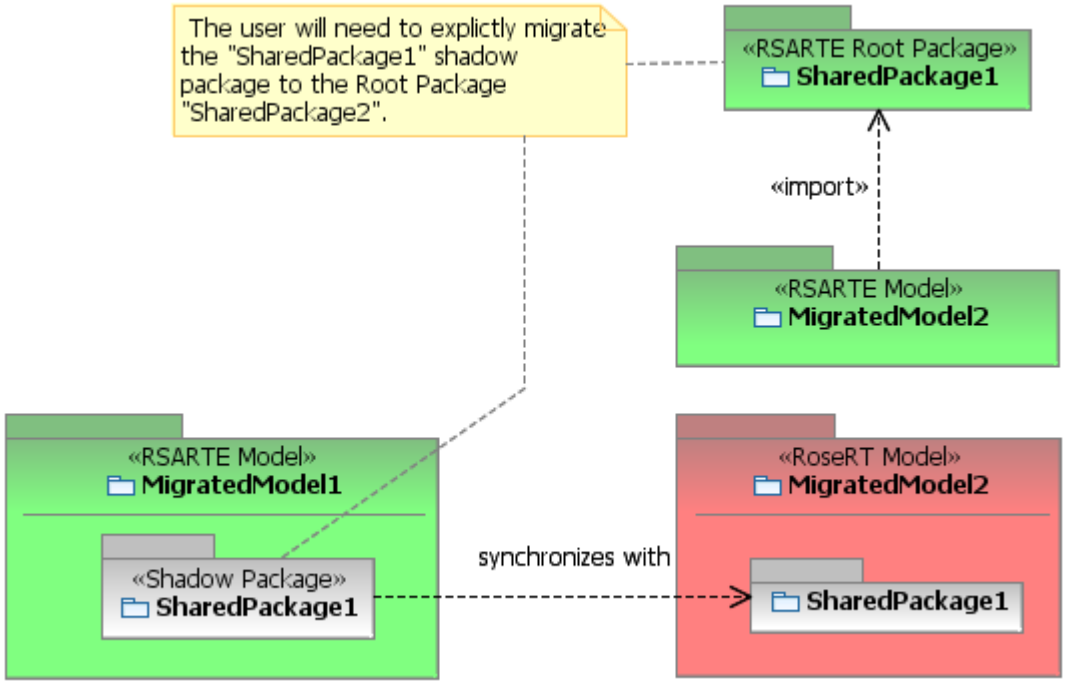


Since this package is owned by the team, it makes sense to extract this package into its own project so that team can eventually include this project in their project set after migration. This is convenient because they don't have to bring in the whole other model into their workspace; they only have to bring in the dependent package. Once all the dependent teams are ready to migrate, it is no longer necessary for the package to remain a "Shadow Package". It can be migrated to itself, so that it becomes a regular package. Then the other teams can either proceed with their migration and their reference to that package will be replaced by an "Element Import" relationship. It should be noted this represents an exception to workflow described in #Decisions since the package is owned by the migrating team and is the primary editing context. However, since it is shared by other models not yet ready to migrate, it must remain a shadow package.

A model shares a package that is owned by another model that is migrating at the same time.

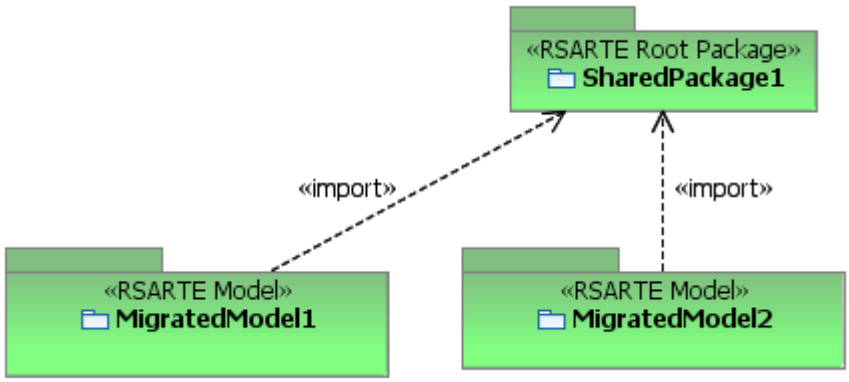
Since these models are migrating at the same time, the package doesn't need to be a shadow package to accommodate any synchronization with the corresponding RoseRT package. The order of how these models are imported is important because it could mean less migration effort. . Also, does it make sense for the models to co-exist in the same workspace or should they be separated further to isolate the specific package dependencies? If we import the model that has the shared package first, then that package will be imported initially as a shadow package because the other model doesn't exist in the workspace yet. After importing the second model, it is now possible to migrate the shadow package to the actual package in the second model.

Model that shares package is migrated first



If the second model is imported first and then the other model imports afterwards, then the package, if specified as shared, will automatically detect on import the actual package existing in the workspace and import as an "Element Import" relationship to that package. This same time and effort because the extra step of "migrating" the package to the actual package in the original model import ordering is eliminated.

Model that owns package is migrated first



In this example, since only a single package is shared by the other model, it may make sense to extract that package into its own project so that project can be referenced independently of the original owning model. This can be done either on import of the owning model on the "Controlled Unit Conversion" wizard page to specify the project where the fragment is created or after import by utilizing the "Extract to Top-Level" refactor command.

In summary, the Enterprise architect needs to consider closely the how particular components are used currently in the RoseRT world and how they will be used in the new RSARTE tooling context. Considering these permutations will allow their software to scale well in the future and ease the migration process to RSARTE.