

Rational Rose RealTime Migration to Rational Software Architect RealTime Edition

-

Pre-migration Best Practices

Steven R. Shaw
Elena Strabykina

IBM

Last updated August 10, 2018 for RSARTE 10.2.

INTRODUCTION....3

MEMORY CONSIDERATIONS....4

Component Merging....4
Sample Transformation Configuration hierarchy after migration:....4
RoseRT Example Component Structure....5
Post Migration RSARTE Example Component Structure....6
Model Separation....7
Monolithic model....7
Separated model....8
Optional ways to separate out model parts into another model using RoseRT....9
Controlled Units....9
On-demand loading of fragments....9
Comparing Notation and Semantic fragments....9

MODEL ARCHITECTURE....11

Refactoring....11
Shared Elements....11
Example of package specification for shared package....12
Table describing migration permutations for a controlled package....13
Shadow Packages....13
Shadow Package example....13
Model Specification dialog in RoseRT....14
Shadow Mechanism for RT Classes and Protocols....15

MODELING....16

Edit Inside....16
Example of "Edit Inside" in RoseRT State diagram....16
Protocol Specification....16
Protocol Specification dialog in RoseRT....17
Sequence Diagram....17
Focus of Control....17
Message Names....17
Ordering in the Interaction....18
Sequence diagram example in RoseRT and after migration....18
Sequence diagram example after migration that demonstrates semantic equivalence to diagram above....19

BUILD / COMPILATION....20

Makefile variables....20
Component Inclusion Paths dialog in RoseRT....20

CODE GENERATION....21

SCRIPTING THROUGH RRTEI....22

Introduction

In large enterprises, there are many different teams working on various products and schedules. Each team may be working with a common development technology, but scheduling and/or dependencies may prevent migration to the newer tooling environment. When an enterprise that uses Rational Rose RealTime (RoseRT) wants to migrate to Rational Software Architect RealTime Edition (RSARTE), they may realize it is not possible to move all products at the same time. As a result, certain teams are required to continue development with RoseRT until their issues are resolved.

The purpose of this document is to provide your organization with information on how to continue work in the RoseRT tooling while minimizing future migration effort to the RSARTE environment.

Memory Considerations

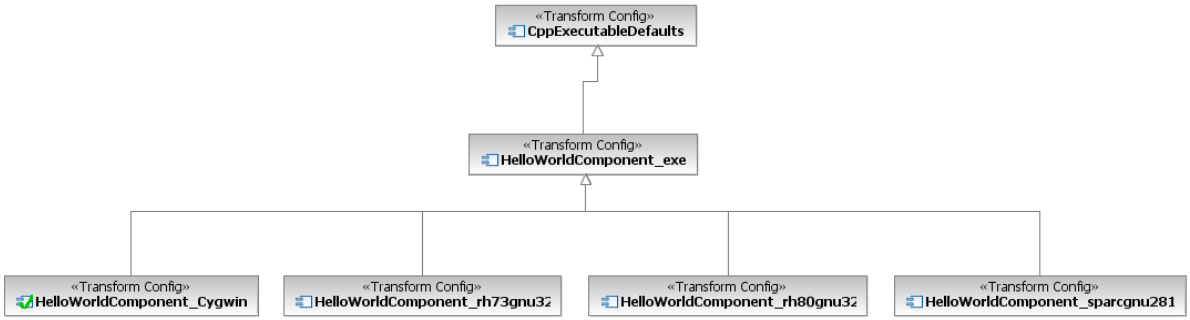
RSARTE is built on a lot of Eclipse-based technology that is reapplied and reused in other domains. The advantage of this approach is that the technology is well-tested, adaptable to different domains and has an established Application Programming Interface (API) for extensibility. However, often these components are not optimized for a specific domain. Therefore, as a result, it's possible that these products use more memory than the equivalent infrastructure in RoseRT. There are some things that you can do to your RoseRT model to minimize memory issues before migrating to RSARTE.

Component Merging

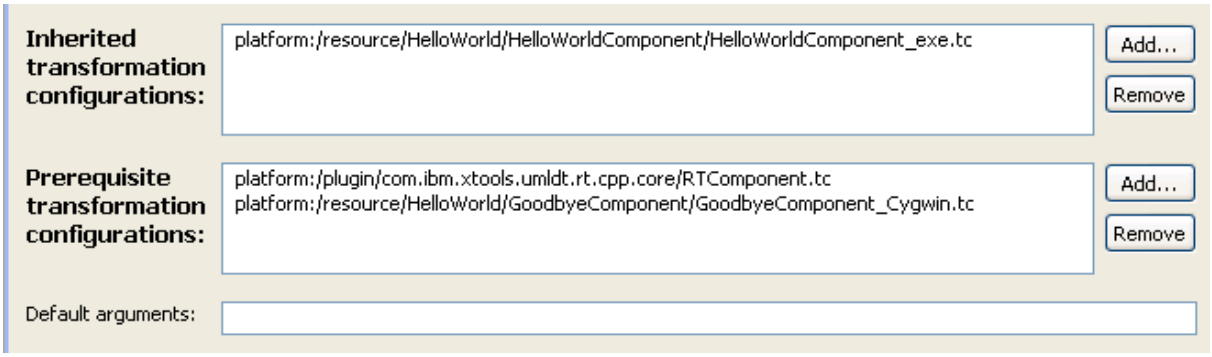
RoseRT requires a separate component for each target configuration. RSARTE utilizes the C/C++ Development Tooling (CDT) which allows a single CDT project to be responsible for multiple target configurations or compile variations (e.g. debug / release). To take advantage of this capability, the RSARTE model import wizard merges components in RoseRT that have the same code generation properties into a single abstract transformation configuration (TC) with the concrete transformation configurations inheriting from it and a single corresponding CDT project. The result is less CDT projects created (in memory), and an easier workspace to manage.

The concept of inheritance for TC files allows the property set functionality from RoseRT to be maintained after migration. Property sets is a convenient mechanism to store default properties for particular element types that are automatically inherited by each model element instance unless they are overridden locally. For models that are migrated from RoseRT, the inheritance for TC files will automatically be set based on the components' use of property sets. Default TCs are created in the project that is analogous to the Default property sets in RoseRT. Also for components that have the same properties in a component package, an abstract TC will be extracted that each component will inherit from.

Sample Transformation Configuration hierarchy after migration:



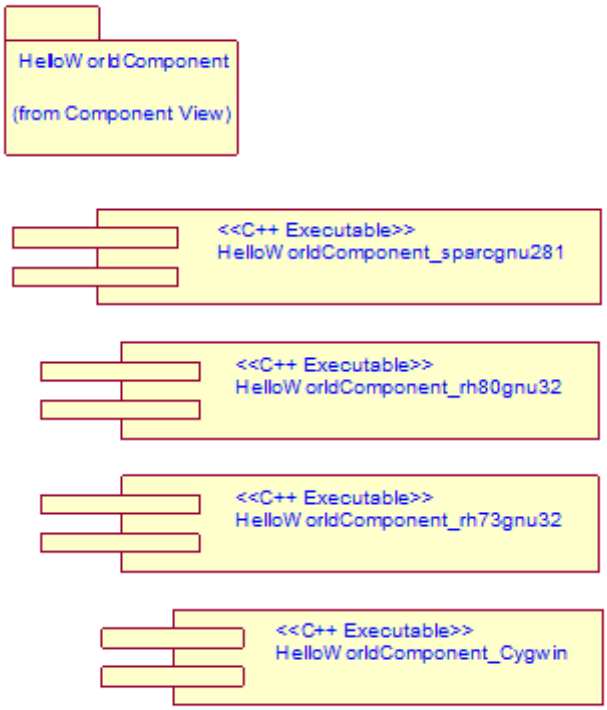
The property for TC inheritance behaves in a similar way to the prerequisite property which also specifies a list of TCs. A TC can have multiple inheritance parents, and the order in which they are specified dictates which will take priority in the case where they have the same properties.



There are visual cues in the editor to see which particular properties have been overridden from the parent properties. If the property text is bolded, then that property is local to the TC and will take precedence over the property stored in the inherited transformation configurations.

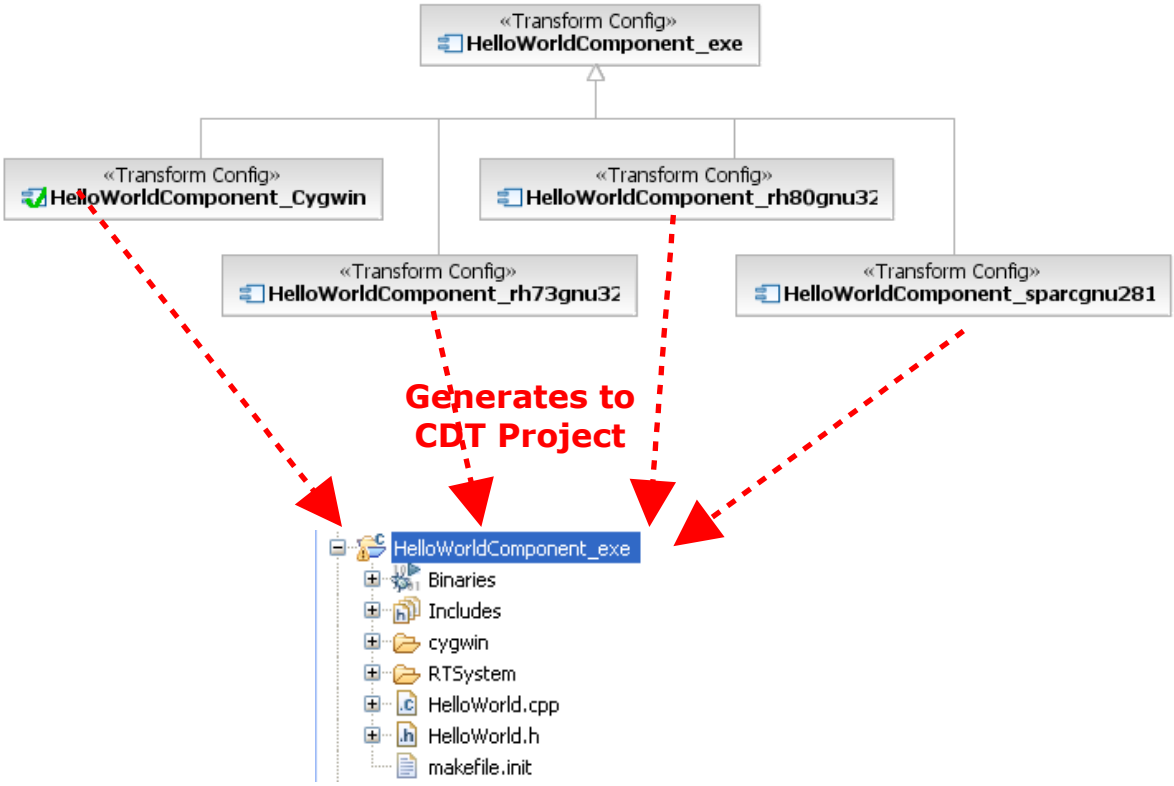
Considering this TC file inheritance and how the component merge will work on migration to RSARTE, we can take a look at a concrete example.

RoseRT Example Component Structure



Above, the RoseRT model has a package with 4 components that generate the same sources to the different target configurations they support. Since they reside in the same package and have the same generation properties, they are candidates for merging during the migration.

Post Migration RSARTE Example Component Structure



Post migration, the components now exist as TC files that inherit from a newly merged TC file that contains all the common generation properties. This means that changes to the generation can be centrally managed in that abstract TC (e.g. adding / removing sources etc.). All the concrete TCs will generate in a single CDT project that has multiple configurations – one for each target (cygwin, sparcgnu281 etc.).

There are some component storage and naming requirements in RoseRT in order for this component merging to occur:

- Components are merge candidates when they exist in the same containment (same package parent). Also, the owning package name must be a legal C++ identifier (no spaces or commas).
- The name of the merged TC comes from the owning package unless they are owned directly by the component view. In this case, the name is derived from the common suffix of prefix of the components to be merged.
- Components are merged if they have the same references and code generation properties. The following properties must be identical in components that will be merged:

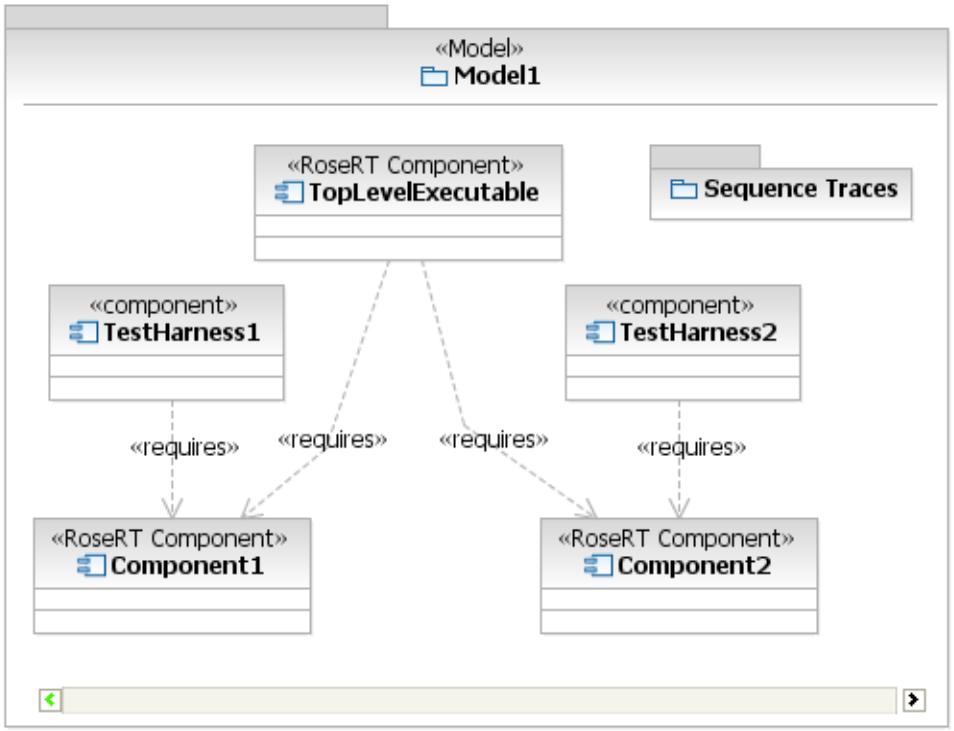
CodeSyncEnabled	Environment	SourceSubdirectory
CommonPreface	GenerateTags	Threads
CompilationMakeInsert	Language	TopCapsule
CopyrightText	References (Sources)	Type
DefaultArguments	SingleDataCompilationUnit	UnitSubdirectory

Model Separation

It is easy to create monolithic models in RoseRT because only one model is allowed in a workspace. Consequently, everything that needs to be viewed has to exist in the model at some level. This doesn't map well in the RSARTE world because Eclipse supports multiple projects in a single workspace and each project may contain multiple models as well.

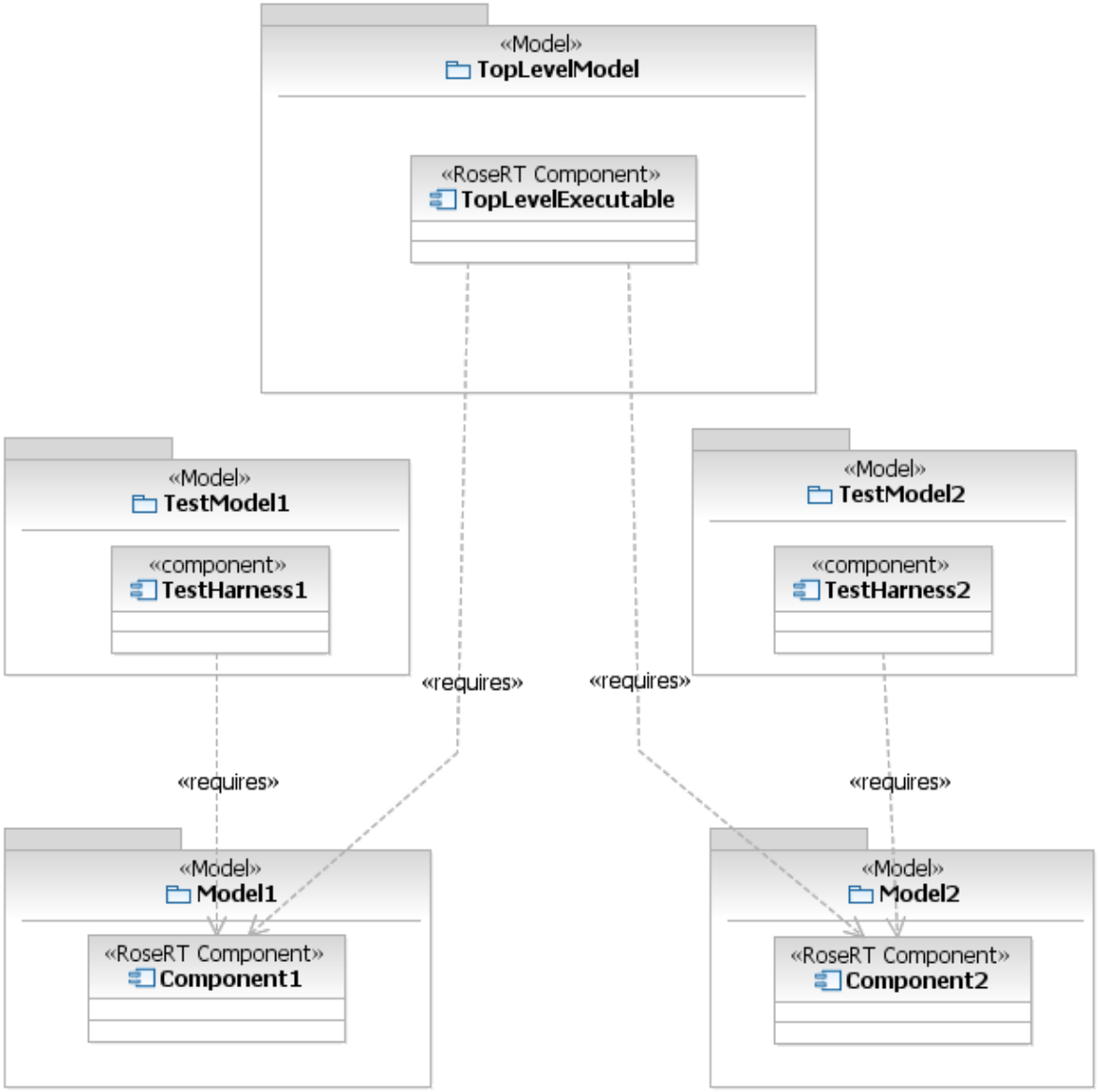
There are many different aspects to a RoseRT model; there are the production bits which are code generated into the executable or library as well as the supporting model elements which help in the construction of the production model. These are the use-cases, sequence diagrams, collaborations and other model elements that describe parts of the system but may not be involved in the code generation aspects of the model. These elements should be scrutinized to see if they are necessary to carry forward with the development model. If you can delete or separate these elements into a new model you will save memory resources during the import process. You may consider deleting old sequence diagrams that were originally generated from a trace execution. For model separation, test artifacts and harnesses that test a specific component within the top model may be good candidates for separation.

Monolithic model



In the example above, all the sub components and systems are contained in the top-level model including the associated test artifacts and residual sequence traces; therefore, this example represents a good candidate for model separation.

Separated model



In the separated model, four new models are created to isolate the test artifacts from the production code. The "TopLevelModel" shares in the components and packages from Model1 and Model2, but doesn't require any visibility of the associated test artifacts. This separation keeps the model foot print smaller and will ease the model migration process because the model import wizard requires less memory.

Optional ways to separate out model parts into another model using RoseRT

1. Export the model and then delete everything except the parts of the model you wish to separate. Afterwards, delete the separated parts from the original model.
2. Ensure the parts you want to separate are inside a controlled package. Create a new model and then share in the controlled package(s). Open the specification for the package in the new model and change it to be "Owned" on the "Unit" tab. Finally, open the original model and remove the controlled package since it now exists as an owned entity in the newly created separated model.

Controlled Units

Having the model controlled offers a number of advantages in terms of managing a project with a team of developers. This is well documented in the RoseRT Team Development Guide.

On-demand loading of fragments

There are some additional advantages when controlled units are migrated into RSARTE as fragments. A fragment is a model element that exists as a root in a separate file resource (analogous to a controlled unit in RoseRT). RSARTE has a loading strategy that states if a fragment isn't referenced by a model or set of fragments that are currently loaded, then the fragment won't be loaded. Essentially, the fragments are loaded into memory "on-demand". When working with a large model, if you are only interested in a particular subsystem, then the fragment loading strategy is very efficient because only the subsystem and it's dependencies are loaded.

Consequently, the more granular the model is controlled to, the more potential memory is saved. A granular approach saves memory resources because dependencies are broken up and won't force the load of elements that are part of a larger unit but don't have dependencies to an element being loaded. To take full advantage of on-demand loading, as a general recommendation, control down to the capsule/class level.

There is cost associated with this since refactoring operations become more cumbersome because the associated resource may need to be moved or will become out of sync with the model name and location. Also, there are more file resources being managed by the source control system which means there is more complexity or overhead with respect to updating with the repository. There is an article on DeveloperWorks that discusses some these issues around model management in a source control repository: Model Management with ClearCase.

Comparing Notation and Semantic fragments

There are two aspects to every model: the semantic elements which transform into domain language code (such as C++), and the notation elements which represent a view of the semantics that allow them to be displayed in a diagram. The transformation is only concerned with the semantic elements in the model and the notation elements are ignored. Given the fragment on-demand load capability, it makes sense to separate out the diagrams (which contain notation elements) into their own controlled units. That way, the controlled units won't be loaded unless you decide to edit or view the diagram during a work session. In

RoseRT, it is not possible to create controlled units from capsule state and structure diagrams. However, you may choose to control any class, sequence or other diagrams in the file system.

Model Architecture

Refactoring

Model architecture has to consider a lot of different concerns – memory consumption, dependencies and ability to change the model effectively. The ability to manage change means the model is separated into components that are modified by individuals or separate teams. This also means that changes from one team or component are often propagated into their dependencies during a refactoring operation. An example of a refactoring operation is where a name of an element changes and dependent elements that reference that element need to update their reference accordingly.

In RoseRT, to refactor effectively you often need to have all components and dependent components in the same model. As a result, it may be more effective to work with the system or executable model to do changes in a particular component. That way, you can be sure that changes are made to dependent components immediately. This approach inhibits working with a "block model" except for more trivial changes (a block model is a model of a single component that is designed for unit testing). When you refactor changes made in a block model, you run the risk of breaking the system model unless done in context to the system model.

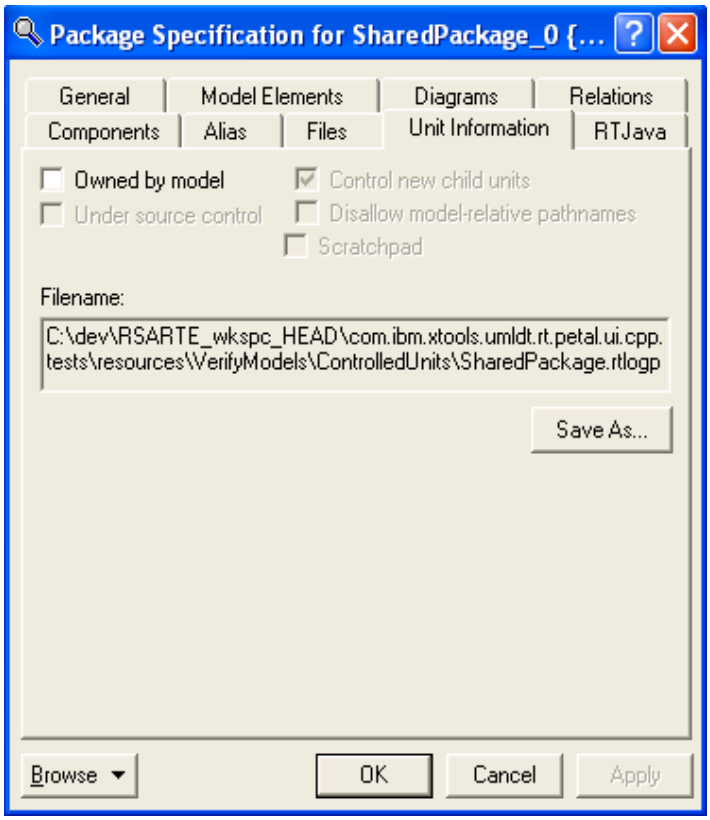
In contrast, if you import block models into RSARTE and the "system model" references these block models, it's possible to work with the "block models" independently because RSARTE refactors closed models located in the workspace. Refactoring closed models is possible because Eclipse workspaces can contain multiple projects, and the projects can have one or more models inside them. Also, models can be closed if not currently being edited, viewed, or referenced. Further, leaf dependencies or models that don't depend on other packages can be edited independent of the rest of the system.

In summary, there is value when you separate your system model into smaller component ("block") models that you can use for test or minor modification. After migration to RSARTE, these "block" models are imported into their project and are considered the primary target for editing instead of relying on the "system" model context. See Separated model.

Shared Elements

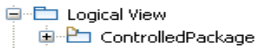
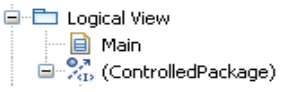
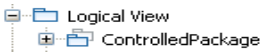
The mechanism of shared elements is a powerful concept in RoseRT because it allows such elements to be reused by multiple models. This allows your team to develop common system components which can be used to build layers of services in different applications. RSARTE supports sharing of packages or classes only. Since shared elements can reside within multiple model contexts, there should be a "master" model context where the shared element is edited and modified. The idea of "ownership" is managed through a property on the specification of the shared element. If you select the "Owned by model" check box, the element is modifiable in the owning context. If cleared, the shared element is not modifiable.

Example of package specification for shared package



During the RSARTE model import process, this property is used to determine how the package will be migrated by default.

Table describing migration permutations for a controlled package

Shared (Controlled) Package in RoseRT	**Migrated As:**	
"Owned by model" property is selected.	Package is imported as an owned fragment (directly analogous to a controlled unit in RoseRT).	
"Owned by model" is cleared and package has already been imported as owned from a different model	Package is replaced by an "element import" relationship to the previously imported package in the model workspace.	
"Owned by model" is cleared and package doesn't exist in the workspace.	Package is imported as a shadow package fragment which allows synchronization with the original package controlled unit in RoseRT.	

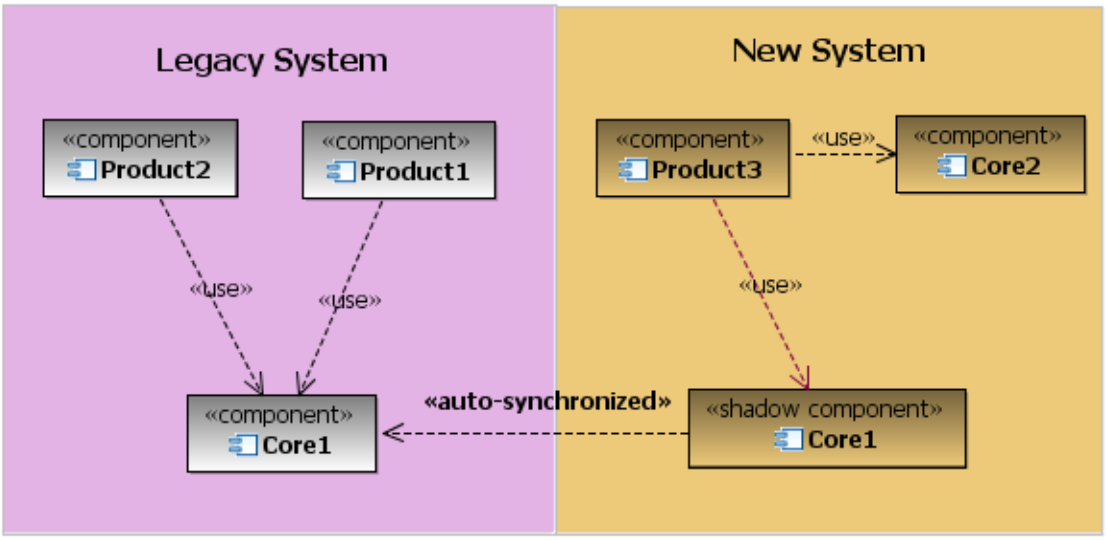
It's possible that the "owned by model" property may not have been rigorously adhered to and the shared package may be "owned" in multiple contexts. Or, the development team may rely on a configuration management (CM) system to make the real determination if a particular package is "owned" or not (in other words, the CM system only allows write access to a particular package based on user privileges and so on). In these cases, you have the ability to modify how a particular controlled package is imported into RSARTE (owned, shared, shadow) through the Controlled Unit Conversion UI page of the "Rational Rose RealTime Model" Import wizard. However, if you know about the "Owned by model" checkbox and you can keep it synchronized with the real model owner in RoseRT, you will save time and effort by modifying these options when it comes time to migrate.

Shadow Packages

Shadow packages are a concept introduced in RSARTE that accommodates large enterprise organizations in their migration effort from RoseRT to RSARTE. A shadow package is a fragment in RSARTE that is not modifiable; it can be resynchronized and re-imported from the original package in RoseRT to keep the content synchronized. As another way to understand this concept, consider it "mastered" in RoseRT and simply replicated in the RSARTE workspace.

A model or subsystem in RoseRT may have outside dependents in other subsystems that share in controlled packages from it. Those dependent models may be in a position to migrate to RSARTE and will have their dependencies to other models become shadow packages. If your model has outside dependents to owned controlled packages, it is useful to understand some restrictions with modifying these packages. If these packages are modified in RoseRT, the contents could be replicated (synchronized) into a shadow package in RSARTE.

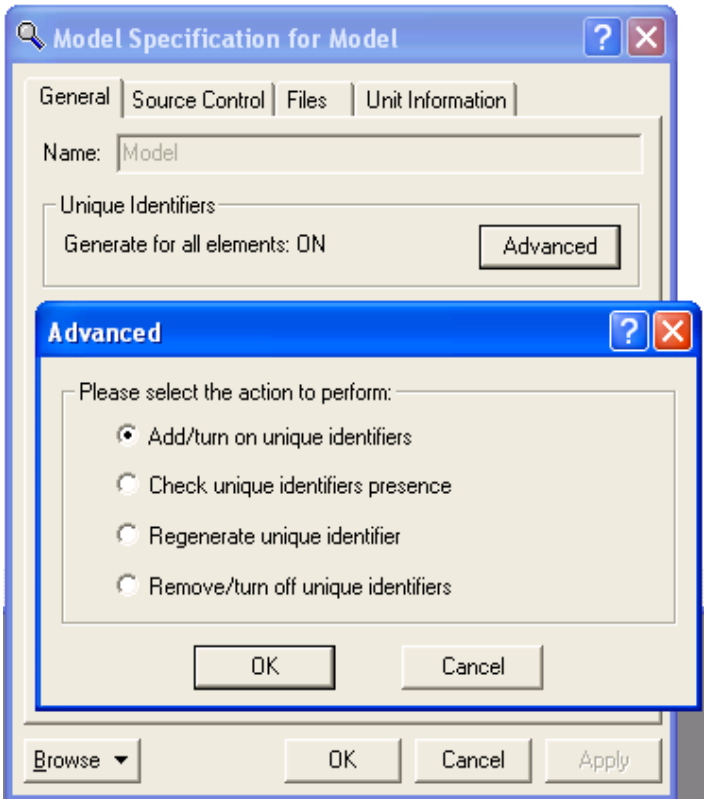
Shadow Package example



In this example, the component Core1 is used by Product1 and Product2 in RoseRT. However, Product3 has migrated to RSARTE and also depends on Core1. Core1 has been imported as a shadow package and replicates or synchronizes with Core1 in RoseRT.

Considering this synchronization, there are some complexities and/or limitations that cannot be avoided due to the nature of the meta-model transformation. To avoid some of these issues, it is best to turn on quid support in RoseRT for all models that own shared packages that will exist as shadow packages in RSARTE.

Model Specification dialog in RoseRT



When you turn quid values (i.e. unique identifiers) on, the synchronization process is able to resolve references to elements that have been renamed or moved. Understand that synchronization will still work if unique identifiers are turned off; however, the model import wizard will probably consider renamed elements as a new element and references to the previously named element will be considered deleted. In this case, it is best to minimize changes in these shared packages or try not to rename or move elements within them.

Shadow Mechanism for RT Classes and Protocols

In addition to packages, the shadow concept is partially available also for real time classes/capsules and protocols, which are separate control units in RoseRT. The difference from the shadow packages is that synchronization is not available for them. However, there is a capability to import units containing classes and protocols, which are not owned by the current model, and once the model that owns the units has been imported, there is a capability to migrate imported classes and protocols. This allows read-only access to shadow elements until the owning model is imported and their conversion to short-cuts after the owning model is imported.

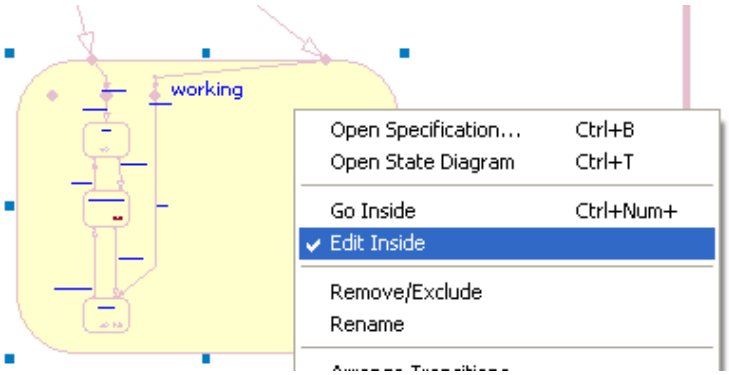
Modeling

The modeling environment in RSARTE has tried to maintain the look and feel of RoseRT diagramming experience (except for the differences introduced by UML2 and the integration with the Eclipse tooling). These differences should not interfere with the migration process, but some of the features have changed significantly which may affect how people decide to model in RoseRT before migration.

Edit Inside

RoseRT has the capability to "Edit-Inside" a state or structure element to see and/or edit its contents. On evaluation of this feature in the RSARTE context, the development team decided that this feature was not very practical as the window into the sub-diagram was difficult to edit and see; further, it didn't scale beyond one level. Consequently, this feature was not migrated over explicitly. There is a way to see the contents of a state / part in RSARTE through the "Region Compartment" and "Structure Compartment" commands respectively. These commands display larger size versions of the elements and can be used indefinitely through the containment hierarchy and are limited only by the space available in the diagram editor.

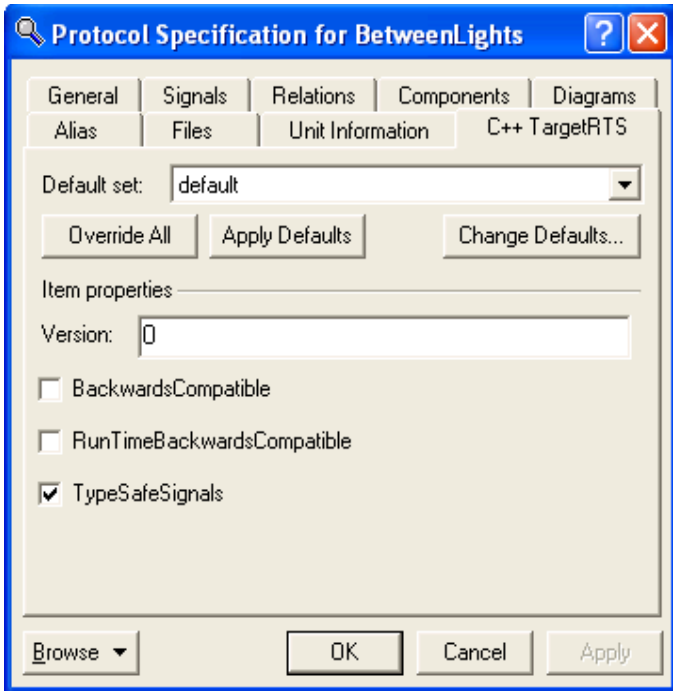
Example of "Edit Inside" in RoseRT State diagram



Protocol Specification

RoseRT was originally a next generation tooling for a tool called "ObjecTime Developer". In the process of that migration, some APIs changed within the Target RunTime System (TargetRTS). To ease this migration, an option was introduced in the Protocol specification to support the old API in backwards compatibility mode. This backwards compatibility mode is not supported in RSARTE so you must ensure that you clear this option in all protocols before beginning the migration.

Protocol Specification dialog in RoseRT



Sequence Diagram

RoseRT and RSARTE are based on a UML (Unified Modeling Language) semantic meta-model. RSARTE is based on a newer version of the UML specification called UML2. The UML2 specification has changed significantly to accommodate more functionality. Interaction semantics is one of the areas that has changed the most. As a result, there are some differences after migrations that may introduce differences in the diagram appearance.

Focus of Control

In RoseRT, you can delete the focus of control (FOC) for asynchronous messages. In RSARTE, the behavior execution specification (analogous to an FOC in RoseRT) is mandatory. After migration, the deleted/missing FOCs will appear with a default minimum size. To avoid diagrams having a slightly altered appearance after migration, you might want to keep the FOC elements in sequence diagrams with asynchronous messages.

Message Names

Message names are not imported when the message is assigned to a signal or operation.
The name of the message is set to the signal name due to the following from the UML 2.2 specification:

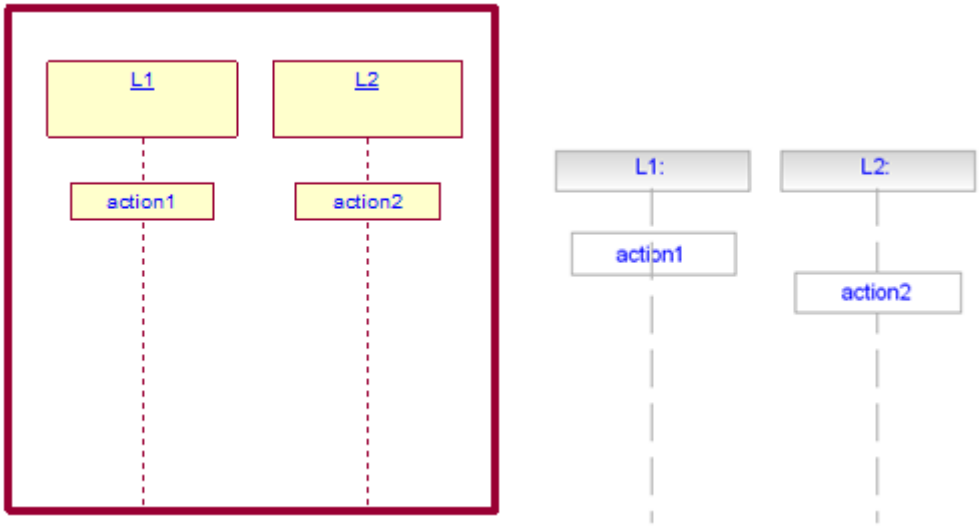
> *[2] The signature must either refer to an Operation (in which case messageSort is either synchCall or asynchCall) or a Signal (in which case messageSort is asynchSignal). The name of the NamedElement referenced by signature must be the same as that of the Message.*

Message names should be kept the same as the signal or operation to ensure that they will appear the same after migration.

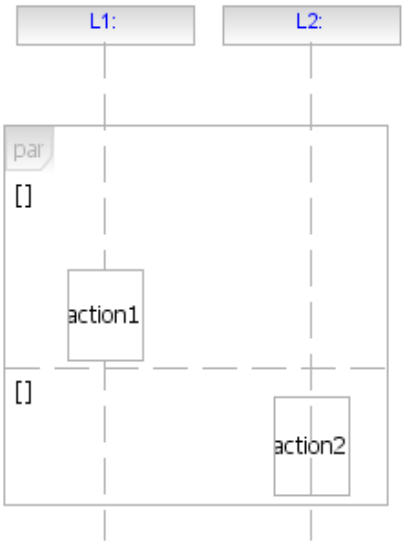
Ordering in the Interaction

The layout of the sequence diagrams usually signifies ordering of the messages in the interaction. In RoseRT, the ordering is only specific to a particular lifeline. This allows for cases where a local state and/or local action are co-located at the same vertical location on different lifelines. In RSARTE, the ordering in an interaction is global, meaning that all elements have an implied ordering relative to all other elements even if they aren't on the same lifeline. This is because RSARTE is built on the UML2 meta-model which has these semantics. Therefore, the local state and local actions are adjusted by the diagram layout algorithm to be offset from each other to reflect that. This is somewhat unavoidable, but you can layout your diagrams with this offset in mind to accommodate the global ordering paradigm. There are ways to represent random ordering in UML2 through the use of a "Parallel Combined Fragment" (see Sequence diagram example after migration that demonstrates semantic equivalence to diagram above). It is not possible to decide if this was the intention in the RoseRT sequence diagram since the vertical alignment is the only indicator. You can add the "Parallel Combined Fragment" after migration, if desired.

Sequence diagram example in RoseRT and after migration



Sequence diagram example after migration that demonstrates semantic equivalence to diagram above

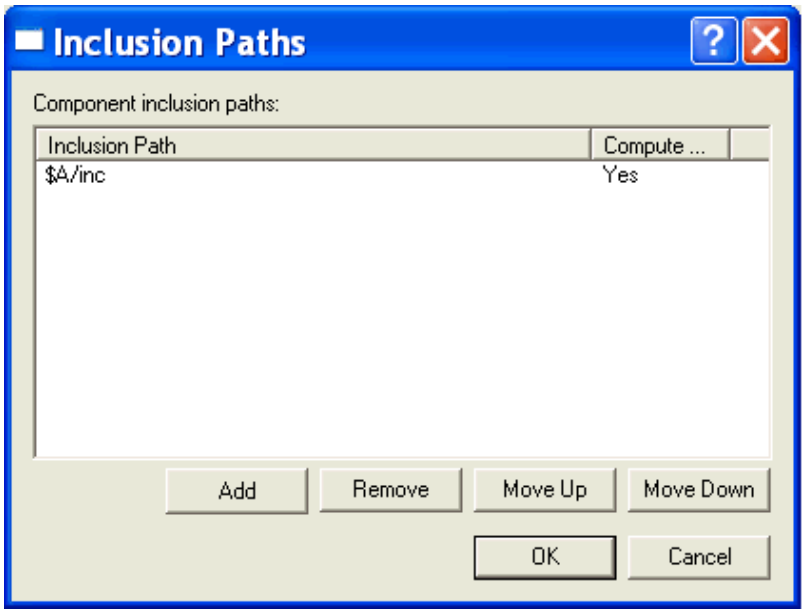


Build / Compilation

Makefile variables

Since build in RSARTE is managed through the CDT (for details, see http://www.eclipse.org/cdt/) there are some differences in how build or makefile variables are supported. In the CDT, the makefile is generated and the assumed variables are defined through environment variables and/or path maps. RoseRT allows for makefile variables which are defined explicitly in the makefile fragments. These work fine for the compile and link because the make system resolves the variables. After migration to RSARTE, this is still true, but the CDT can't resolve them dynamically so warning and problem markers are created (indexing isn't functional). To solve this problem, on import, all makefile variables are replaced with an Eclipse value variable. The problem with this is that it is a workspace variable and is not easily contributed to a CM system. The variables must be shared by exporting the variables through the Eclipse "File->Export (preferences)".

Component Inclusion Paths dialog in RoseRT



Above is a RoseRT model with inclusion paths that assumes makefile variables are defined.

Instead of using makefile variables, use PathMaps or Environment variables to define variables instead. That way, there are no issues in a team environment after migration of the model since the PathMaps exist in the model and the environment variables aren't tooling specific.

Code Generation

Code generation is a critical part of the RoseRT and RSARTE tooling. Therefore, a lot of work has gone into making sure all aspects of code generation are semantically equal in both RoseRT and RSARTE. The major difference is that RSARTE code is generated into a CDT C++ project which is then compiled and built through the Eclipse build facility. RSARTE supports C++ and C code generation, but not Java.

Since the code is generated into a CDT project, not all of the "C++ Generation" properties may be useful or relevant anymore. Any of the properties for specifying directories for code generation may not provide value after import. For instance, you can specify the "OutputDirectory" property for a component to be any directory in your file system. By default, it will output into your model directory. In RoseRT, it might be more useful to redirect this outside of your model directory especially if it conflicts with the CM system in place. In RSARTE, the CDT project where the source is generated to is a peer to the model project in the workspace so this conflict won't exist anymore. If you do specify and override this property, after migration, the CDT project still appears in the workspace, but it will have a location value to the overridden directory.

Scripting through RRTEI

Scripting is a powerful way to query the model for custom information or extend the model capabilities of the tool. In Rose RT, this is done through the RRTEI which has Summit Basic syntax with a rich set of commands and API. In Eclipse (RSARTE), the extensibility is done in a completely different way with a Java-based technology integrating with the plug-in facility of OSGI. Consequently, it is not practical or feasible to provide a migration path for the RRTEI scripts. Any scripting in Rose RT must be rewritten using the new extensibility API of RSARTE and all its underlying components. This should be considered in the case of a future migration to RSARTE for any new scripting projects since the effort would have to duplicate in the new tooling. This is worth the effort in the long term since these extensions are built on open source components that are reused in many contexts. The expertise gained is re-applicable across most Eclipse applications that build on the same open source components.