

Guide to Team Development

RATIONAL ROSE® REALTIME

VERSION: 2002.05.20

PART NUMBER: 800-025114-000

WINDOWS/UNIX

IMPORTANT NOTICE

COPYRIGHT

Copyright ©1993-2002, Rational Software Corporation. All rights reserved.

Part Number: 800-025114-000

Version Number: 2002.05.20

PERMITTED USAGE

THIS DOCUMENT CONTAINS PROPRIETARY INFORMATION WHICH IS THE PROPERTY OF RATIONAL SOFTWARE CORPORATION (“RATIONAL”) AND IS FURNISHED FOR THE SOLE PURPOSE OF THE OPERATION AND THE MAINTENANCE OF PRODUCTS OF RATIONAL. NO PART OF THIS PUBLICATION IS TO BE USED FOR ANY OTHER PURPOSE, AND IS NOT TO BE REPRODUCED, COPIED, ADAPTED, DISCLOSED, DISTRIBUTED, TRANSMITTED, STORED IN A RETRIEVAL SYSTEM OR TRANSLATED INTO ANY HUMAN OR COMPUTER LANGUAGE, IN ANY FORM, BY ANY MEANS, IN WHOLE OR IN PART, WITHOUT THE PRIOR EXPRESS WRITTEN CONSENT OF RATIONAL.

TRADEMARKS

Rational, Rational Software Corporation, Rational the e-development company, ClearCase, ClearCase Attache, ClearCase MultiSite, ClearDDTS, ClearQuest, ClearQuest MultiSite, DDTS, Object Testing, Object-Oriented Recording, ObjecTime & Design, Objectory, PerformanceStudio, ProjectConsole, PureCoverage, PureDDTS, PureLink, Purify, Purify'd, Quantify, Rational, Rational Apex, Rational CRC, Rational Rose, Rational Suite, Rational Summit, Rational Visual Test, Requisite, RequisitePro, RUP, SiteCheck, SoDA, TestFactory, TestFoundation, TestMate, The Rational Watch, AnalystStudio, ClearGuide, ClearTrack, Connexis, e-Development Accelerators, ObjecTime, Rational Dashboard, Rational PerformanceArchitect, Rational Process Workbench, Rational Suite AnalystStudio, Rational Suite ContentStudio, Rational Suite Enterprise, Rational Suite ManagerStudio, Rational Unified Process, SiteLoad, TestStudio, VADS, among others, are either trademarks or registered trademarks of Rational Software Corporation in the United States and/or in other countries. All other names are used for identification purposes only, and are trademarks or registered trademarks of their respective companies.

Microsoft, the Microsoft logo, Active Accessibility, Active Channel, Active Client, Active Desktop, Active Directory, ActiveMovie, Active Platform, ActiveStore, ActiveSync, ActiveX, Ask Maxwell, Authenticode, AutoSum, BackOffice, the BackOffice logo, BizTalk, Bookshelf, Chromeffects, Clearlead, ClearType, CodeView, Computing Central, DataTips, Developer Studio, Direct3D, DirectAnimation, DirectDraw, DirectInput, DirectMusic, DirectPlay, DirectShow, DirectSound, DirectX, DirectXJ, DoubleSpace, DriveSpace, FoxPro, FrontPage, Funstone, IntelliEye, the

IntelliEye logo, IntelliMirror, IntelliSense, J/Direct, JScript, LineShare, Liquid Motion, the Microsoft eMbedded Visual Tools logo, the Microsoft Internet Explorer logo, the Microsoft Office Compatible logo, Microsoft Press, the Microsoft Press logo, Microsoft QuickBasic, MS-DOS, MSDN, Natural, NetMeeting, NetShow, the Office logo, One Thumb, OpenType, Outlook, PhotoDraw, PivotChart, PivotTable, PowerPoint, QuickAssembler, QuickShelf, Realmation, RelayOne, Rushmore, SourceSafe, TipWizard, TrueImage, TutorAssist, V-Chat, VideoFlash, Virtual Basic, the Virtual Basic logo, Visual C++, Visual FoxPro, Visual InterDev, Visual J++, Visual SourceSafe, Visual Studio, the Visual Studio logo, Vizact, WebBot, WebPIP, Win32, Win32s, Win64, Windows, the Windows CE logo, the Windows logo, Windows NT, the Windows Start logo, and XENIX are trademarks or registered trademarks of Microsoft Corporation in the United States and other countries.

FLEXIm and GLOBEtrotter are trademarks or registered trademarks of GLOBEtrotter Software, Inc. Licensee shall not incorporate any GLOBEtrotter software (FLEXIm libraries and utilities) into any product or application the primary purpose of which is software license management.

Portions Copyright ©1992-2002, Summit Software Company. All rights reserved.

PATENT

U.S. Patent Nos. 5,193,180 and 5,335,344 and 5,535,329 and 5,835,701. Additional patents pending.

Purify is licensed under Sun Microsystems, Inc., U.S. Patent No. 5,404,499.

GOVERNMENT RIGHTS LEGEND

Use, duplication, or disclosure by the U.S. Government is subject to restrictions set forth in the applicable Rational Software Corporation license agreement and as provided in DFARS 277.7202-1(a) and 277.7202-3(a) (1995), DFARS 252.227-7013(c)(1)(ii) (Oct. 1988), FAR 12.212(a) (1995), FAR 52.227-19, or FAR 227-14, as applicable.

WARRANTY DISCLAIMER

This document and its associated software may be used as stated in the underlying license agreement. Rational Software Corporation expressly disclaims all other warranties, express or implied, with respect to the media and software product and its documentation, including without limitation, the warranties of merchantability or fitness for a particular purpose or arising from a course of dealing, usage, or trade practice.



Contents

Chapter 1 Team Development 1

Goals of Team Development 1

Sharing Within a Team Environment 3

Protecting Configuration Items From Unintentional Changes 5

 Overwriting A Modification 6

 Adding Dependency Issues 9

 Changing Language Semantics 10

Managing Relationships Between Configuration Items 12

Managing and Delivering Configuration Items 14

Improving Efficiency in Team Development 17

 Team Development Roles 18

 Architect Role 19

 Developer Role 19

 Product Tester Role 20

 Integrator Role 20

 Source Control Administrators 21

 Configuration Managers 21

 Project Managers 21

 Customer Role 22

Recommendations	22
Delivering the Product	22
Source Control Fundamentals	23
Preempting Conflicts	25
Packaging Strategy	26
Managing Dependencies	27
Labeling	27
When Merging is Necessary	28
Merging Detail Code Before Using Model Integrator	29
Artifact Freeze	29
Building and Executing a Rational Rose RealTime Model	30
Advanced Concepts and Heuristics	31
Moving Controlled Model Elements	31
Renaming a Controlled Model Element	32
Primary and Secondary Edits	32
Understanding Blue Deltas	35
Parallel Development	36
Model Integrator	37
Using Rational ClearCase Multi-Site	38
Using Rational ClearCase UCM	38
Unique Ids	39
Rational Quality Architect - RealTime Edition	43
Additional Heuristics for Team Development	43
Additional Recommendations	45

Chapter 2 Storage of Model Data 47

Storing Model Data	47
What is a Controllable Element and a Controllable Unit?	48
What Elements Can Be Controlled?	50
Parent and Child Controlled Elements	51
Directory Structure for Model Data	52
File Names for Controlled Units	55
Controlled Units are Saved when Building	56
Unit Information Tab	57
What Level of Granularity Should I Use?	59

Sharing Controlled Units	61
Overview of Import, Add, and Share	62
Creating Sharable Controlled Units	64
Sharing Model Properties with Controlled Units	64
Working with Controlled Units	65
Controlling a Subset of the Controllable Elements	65
Controlling All of the Controllable Elements	66
Changing the Granularity of Controlled Units	66
Moving Controlled Units	67
Moving Controlled Units Between Model Directories	67
Moving Elements Between Controlled Units	67
Synchronizing Models with the File System	68
Export Controllable Elements from a Model to a File	68
Services Library packages	68
Import Controllable Elements from a File to a Model	69
Add an Existing Controlled Unit to a Model	70
Share an Existing Controlled Unit into a Model	71
Produce a Single Model File from a Model with Many Units	73
Virtual Path Maps	73
How Do Virtual Paths Work?	74
Defining Virtual Paths	74

Chapter 3 Source Control Fundamentals 79

Fundamentals	79
Source Control in Rational Rose RealTime	80
Source Control Status	80
What are Primary and Secondary Edits?	81
Source Control Operations	87
Types of Source Control Systems	91

Source Control Development Concepts 92

Development Activity 92

Integration 92

Lineup 92

Working in Isolation 93

Versioning Strategies 93

Single Stream Versioning 93

Parallel Stream Versioning 94

Chapter 4 Organizing a Model (Architect Activities) 97

Packages, Models, and Subsystems 97

One Model versus Multiple Models 99

Getting Started 100

Mapping the Architecture to Subsystems 100

Decomposing a Model into Subsystems 100

Splitting a Model 101

Checking Package Dependencies for Completeness 101

Show Access Violations 101

Determine the External Dependencies for a Package 102

Check if a Subsystem is Self-contained 104

Define Subsystem Interface 104

Best Practices 104

Scratch Pad Packages 105

Setup Subsystem Components 107

Background 107

Components in Subsystems 108

Support for Unit Testing 110

Use Property Sets for Build Settings 110

	Processors and Component Instances	111
	Project Level Processors	111
	Subsystem Level Processors	112
	Component Instances	112
	Preparing and Releasing Subsystems	113
	Splitting a Model into Subsystem Models	114
	Should You Split a Model Before Adding to Source Control?	114
	Splitting a Model Not in Source Control	115
	Splitting a Model Under Source Control	118
Chapter 5	Working with a Model Under Source Control (Developer Tasks)	123
	Setting up your Source Control Tool	123
	Configuring Work Areas	124
	Getting a Specific Lineup of a Model	124
	Opening a Model Under Source Control	125
	Adding a new Controlled Unit into Source Control	125
	Check Out Parent Package	125
	Checking Controlled Units In and Out of Source Control	126
	Checking Out Controlled Units	126
	Checking In Controlled Units	126
	Submitting All Changes to Source Control	126
	Undoing a Check Out	128
	Building and Running Locally	129
	Reusing Build Settings	129
	Probes and Inject Messages	130
	Unit Testing within a Subsystem	130
	Best Practices	130

Set up Private Components 130
Differencing and Merging Model Elements 131
Synchronizing Models with Source Control 132
Promoting Changes for Integration 132

Chapter 6 Building and Integrating (Integrator Tasks) 133

Building using Automated Scripts 133
 Virtual Path Map Symbols 135
Building within a Larger Build Procedure 135
Reuse of Build Artifacts 136
 Creating Reusable Build Artifacts 136
 Using Build Artifacts 137
Integrating Changes 137
Automating Model Validation 137

Chapter 7 Source Control Administration 139

Set up a Source Control System and Repository 140
Control Appropriate Model Elements as Units 140
Create a Local Work Area 140
Save Model to Local Work Area 141
Configure the Workspace Source Control Options 141
Add the Model to Source Control 141
Make Default Workspace Available to Project Members 141
Defining Developer Work Areas 142
Creation of Labels and Lineups 142
Manipulation of the Source Control Repository 142

Chapter 8 Source Control Tools 143

- Rational ClearCase 144
 - General Recommendations 144
- UCM Integration 146
 - Activity Selection Combination Box 146
 - Run Project Explorer 146
 - Rebase 146
 - Deliver 147
- Snapshot Views 147
- ClearCase Workstation Setup 150
 - Command Line Access to the Source Control Tool 150
 - Element type setup: type manager 150
 - ClearCase Options 151
- ClearCase Repository Setup 151
- ClearCase Work Area Setup 152
- Microsoft Visual SourceSafe 152
 - General Recommendations 153
 - Source Control Operation Behavior with SourceSafe 153
 - Label 153
- SourceSafe Workstation Setup 153
 - Command Line Access to the Source Control Tool 153
 - Set Project Mapping Option 154
 - Let Visual SourceSafe Know Which Database to Use 154
 - SourceSafe Repository Setup 154
 - SourceSafe Work Area Setup 155
- RCS and SCCS 155
 - Repository Mapping Files (.rmf) 156
 - Source Control Operation Behavior with SCCS 157
 - RCS/SCCS Repository Setup 157
 - RCS/SCCS Workstation Setup 157
 - RCS/SCCS Work Area Setup 158

PVCS	159
Source Control Operation Behavior with PVCS	159
PVCS Workstation Setup	159
PVCS Repository Setup	160
PVCS Work Area Setup	161

Chapter 9 Model Validation 163

What is a Model Inconsistency?	164
What is an Unresolved Reference?	165
What do the Errors/Warnings Mean?	167
Validating Names	169

Chapter 10 ClearCase Parallel Development: Sample Process 171

Parallel Development Overview	172
Making Design Changes in Parallel	174
Using View Templates	175
ClearCase Entities	176
Views	176
View Template	176
Labels	176
Initial Setup	176
Create the Integrator View	177
Create Project Labels	177
Create Initial Lineup	177
Creating the Developer View Template	178
Automated Builds	181
Create the Build View	181
Label Build Files	182
Perform Build	182
When the Build Completes Successfully	182

Developer Process	184
Creating a Developer View	184
Starting a Development Activity	185
Working on a Development Activity	185
Finishing a Development Activity	185
Integration Process	186
Integrating Intermediate Changes	187
View Template Script Usage	187
vtadmin	187
vtsetview	188

Chapter 11 Customizing Source Control Interface Scripts 189

Customizing Scripts	190
Input Parameters	190
Output Expected	190
Output Format	190
Script Return Code	190
Notes	190
Script Parameters	191
cm_getcaps	193
cm_status	195
cm_get	196
cm_add	197
cm_checkout	198
cm_checkin	199
cm_uncheckout	200
cm_history	201
cm_extract	202
cm_label	203
cm_diff	204
cm_merge	205

Index 207

Figures

UML Class Diagram of a Shared and Isolated Implementation	4
Overwriting a modification	6
Check-out and Check-in Scenario	6
Checking Out an Artifact After it is Checked In	7
Merging Changes Prior to Check-In	8
Comparison Between Versions	8
Removing Required Dependencies	9
Code Example Showing Changes to Language Semantics	11
Resulting Artifact After Merging Changes	11
Comparing Dependency Reports	13
Labelling Configuration Items	14
Example of Labelling Items	16
Comparing Reports	17
Parallel Stream Versioning Strategy	24
Packaging Strategies	26
Incorrect Merge Scenario	41
A Correct Merge Scenario	42
Browser Icons for Controlled Units	49
Browser Icons Example	50
Sample model structure	52
Directory structure for sample model	53
Sample directory after granularity is reduced	54
Filename Selection dialog	55
Directory Name Selection dialog	56
Unit Information tab	57
Unique id conflict dialog	70
Export shared package dialog	73
Virtual Path Map dialog	75
Controlled Unit Icons with Source Control	81
Model Validation Example	82
Source Control Settings	83
Tools > Source Control Menu	86
Source Control in the Browser context menu	87
History dialog example	90

Example Version Tree 95
Model, Packages, and Subsystems 99
Show Access Violations dialog 102
Package Dependencies Diagram Example 103
Scratch Pad Package Unit Information Tab 106
Example Subsystem Components 109
Add to Source Control dialog 127
Check In Dialog 128
Undo Check Out Dialog 129
Version Tree Example 173



Chapter 1

Team Development

Contents

This chapter is organized as follows:

- *Goals of Team Development* on page 1
- *Sharing Within a Team Environment* on page 3
- *Protecting Configuration Items From Unintentional Changes* on page 5
- *Managing Relationships Between Configuration Items* on page 12
- *Managing and Delivering Configuration Items* on page 14
- *Improving Efficiency in Team Development* on page 17
- *Team Development Roles* on page 18
- *Source Control Fundamentals* on page 23
- *Building and Executing a Rational Rose RealTime Model* on page 30
- *Advanced Concepts and Heuristics* on page 31

Goals of Team Development

Developing complex systems requires that groups of people, such as analysts, architects, developers, and testers, coordinate their efforts to produce the finished product. Consequently, they must ask themselves the following questions:

- What are we trying to accomplish in team development?
- What are the goals of team development?

- How does Rational help implement strategies and best practices to meet those goals?
- What do I need to do to have efficient and effective team development?

The purpose of this book is to outline the goals of team development, and recommend some best practices when using Rational Rose RealTime to help ensure success.

Team development touches on development, testing, configuration management, project management, and other disciplines such as engineering, analysis and design.

This overview of team development helps provide the entire team with an overview of the challenges associated with team development, while specifically outlining the tools and mechanisms Rational Rose RealTime supports to aid in implementing a team development strategy.

To support teams of analysts, architects, and software developers, Rational Rose RealTime:

- Allows team development of a single model by supporting decomposition of the model into versionable units, called controlled units.
- Permits moving or copying controlled units between work areas using virtual path maps.
- Permits sharing subsystems and layers among project members and external projects through shared packages.
- Allows you to generate C++ libraries in a development model, and share these libraries into user models.
- Enables teams to manage their model in concert with other project artifacts by integrating with source control systems, such as Rational ClearCase.
- Provides a tool called Model Integrator, to compare and merge controlled units.
- Enables teams to build their models in concert with other project artifacts by integrating with standard build environments, such as Rational ClearCase clearmake.

The Guide to Team Development provides an overview of the basic team development concepts in Rational Rose RealTime and specifies how to configure and use Rational Rose RealTime in a team environment.

The goals of team development are to:

- Allow team members to share their work with a team. See *Sharing Within a Team Environment* on page 3.
- Protect configuration items from unintentional change. See *Protecting Configuration Items From Unintentional Changes* on page 5.
- Manage the relationship between configuration items. See *Managing Relationships Between Configuration Items* on page 12.
- Deliver specific versions of configuration items to interested parties. See *Managing and Delivering Configuration Items* on page 14
- Reduce or eliminate disruptions to team activities. See *Improving Efficiency in Team Development* on page 17.

Sharing Within a Team Environment

After a developer completes an activity (work), they require a mechanism to share that work with others. Integration is the mechanism that permits the integration of changes made by a team member into what is currently being shared.

A version control system can facilitate the work flow of team members. A team member working on a shared artifact acquires some type of implicit or explicit permission to check-in their work by performing a check-out prior to working on the artifact.

The check-out status for the artifact indicates to other team members that work is currently being done to change the artifact. A configuration manager or configuration system can monitor these operations and enforce any policies. The mechanism can involve the use of a version control system, or it may be an unsophisticated implementation whereby the check-in is a simple copy, and the communication is verbal between developers. Regardless of the mechanism used, an awareness of a change at the appropriate levels must be achieved, and you must assess the implications of the change.

A check-in does not necessarily imply that the artifact is immediately available to team members. Typically, it is useful to work with older versions of shared artifacts until such time as the team is ready to access the latest version.

A version control system allows the team to return to previous versions of work, while providing an audit trail of changes. The desire to associate work with specific requirements is a type of policy the Integrator can enforce at integration time.

Work produced by a member of a team can affect other members of the team; therefore, those effects must be intentional. A copy of the work is made available to a team member in an environment isolated from other team members.

The environment is only isolated one-way. The work environment can see shared team artifacts, but other environments are not effected by the isolated environment. Figure 1 is a UML class diagram that shows a typical implementation of how work is shared and isolated. The Work class in Figure 1 is not available to other team members.

Implementation

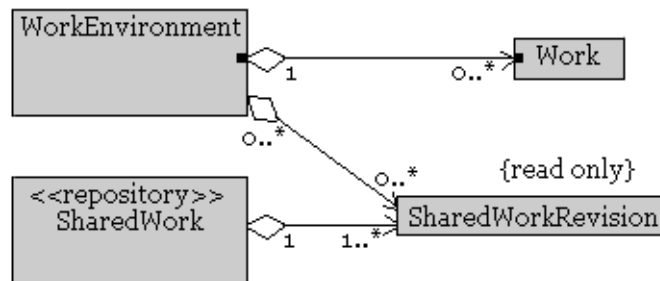


Figure 1 UML Class Diagram of a Shared and Isolated Implementation

Note: Some version control tools may implement a strategy where the multiplicity between the SharedWork class and the SharedWorkRevision would be 0..*

The benefits of this type of implementation are:

- Development team members can produce builds in their isolated environment in an iterative, non-intrusive way. It also allows team members to see a read-only version of shared work.
- Testing teams can perform a series of tests on a specific lineup of work in their own test environment. A lineup is a collection of specific versions of files from a version control repository.
- Production users can use a particular lineup of work that has met quality control criteria.




Protecting Configuration Items From Unintentional Changes

There are several ways a revision can cause unintentional changes to configuration items:

- Direct conflicting change where one change overwrites another. See *Overwriting A Modification* on page 6.
- The source from one change conflicts with another change by removing a dependency that one of the changes relies on. See *Adding Dependency Issues* on page 9.
- A second modification changes the language semantics of the first change.

Table 1 shows the legend that explains some images found in Figure 2 through Figure 6.

Table 1 *Image Legend*

Image	Description
	Represents an unintended change
	Represents movement
	Represents a unit of work or configuration item

Overwriting A Modification

If a team member shares their work with the team, not realizing that someone else produced or edited some work with the same name, they may overwrite the changes of the other team member.

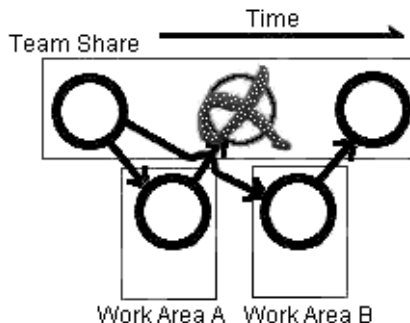


Figure 2 *Overwriting a modification*

Most version control tools provide adequate protection from this type of unintentional change through a process of obtaining permission to make modifications, called a check-out. The version control tool grants implicit permission when there are no check-outs currently in place. When one team member has an artifact checked out, other team members are denied permission to check out that same artifact until it is no longer required by the first team member. Figure 3 shows a scenario where a check-out is followed by a check-in, allowing the sequence of events to iterate.

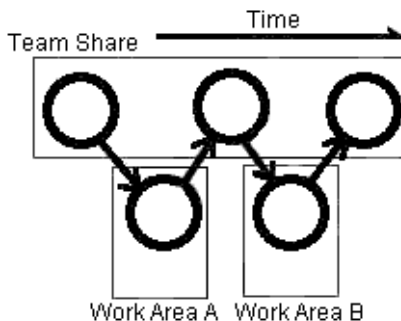


Figure 3 *Check-out and Check-in Scenario*

This type of scenario may cause contention that is unacceptable for high traffic work items. The diagonal lines in Figure 4 indicate that a check-out cannot occur until the previous check-in process completes.

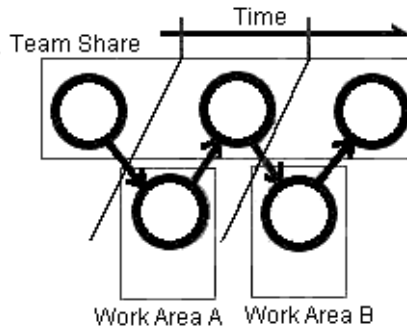


Figure 4 *Checking Out an Artifact After it is Checked In*

The problem illustrated in Figure 2 commonly occurs in strategies that do not use a version control system. Because previous versions of configuration items are always available to developers, the possibility of having this type of unintended change always exists. A developer may make changes to a private copy of an artifact without permission to do so. Subsequently, they may acquire the appropriate permission and check-in the changes of the local copy that may not represent the latest version of the configuration item.

You can use a merge tool to apply a combined set of changes in situations when multiple team members have permissions to make changes to a single artifact. Figure 5 shows how you can merge two changes made to the same artifact.

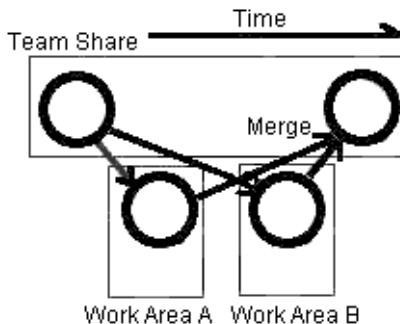


Figure 5 Merging Changes Prior to Check-In

It may be difficult to remove a set of changes that occurred in a previous version of an artifact. The situation in Figure 6 shows us three versions of an artifact. If you want to remove all changes applied to the second version (the changes occurring between the two diagonal lines), you may encounter difficulties.

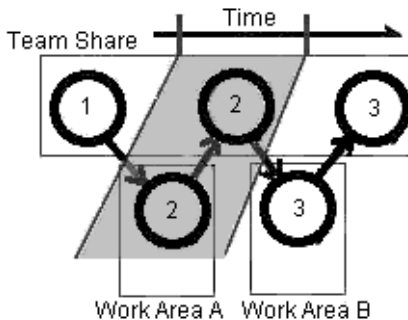


Figure 6 Comparison Between Versions

For example, the changes between version 1 and version 2 must be compared to the changes between version 1 and version 3.

Obtaining adequate permission to modify artifacts helps to ensure that unintentional changes do not occur. Configuration management can choose to implement and enforce this type of policy.

Adding Dependency Issues

Modifying an artifact may cause a conflict with another change if it removes a dependency that one or more other artifacts rely on.

Figure 7 shows how this type of problem can occur.

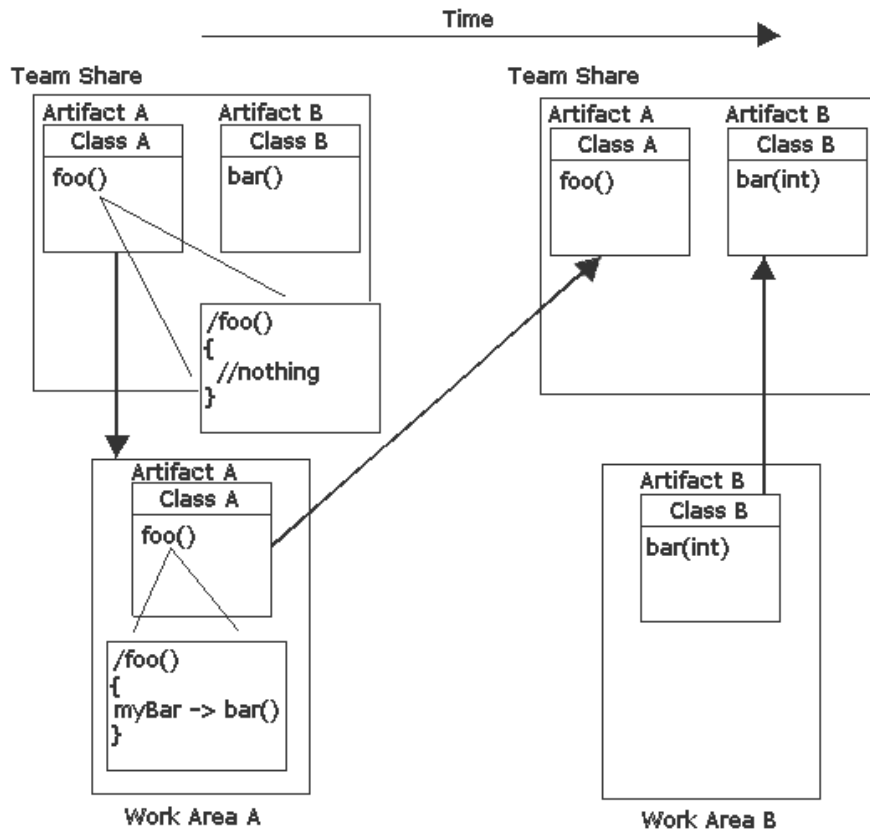


Figure 7 Removing Required Dependencies

Developer A and B individually check out artifacts A and B respectively, and have access to the shared version of artifact A and B respectively.

Developer A creates a new dependency in `foo()` by adding `myBar-> bar()`.

Developer B makes changes to `bar()` in class A by changing the parameter signature to `integer`.

Changes to `bar` - from `bar()` to `bar(int)` - cause any references to this function to fail. The changes made by Developer B (to artifact B) that are referenced by `foo` in artifact A are not valid.

Note: *Most merge tools are unable to identify a conflict here because they compare items of work individually, and not against all referenced work.*

This type of change is common and may have serious implications. Often, when product maintenance is underway and feature development is concurrently managed, the maintenance person or developer may be unsure or unaware of all dependencies involved in a proposed change. Rather than research all the dependencies associated with the artifact, they do not modify the original item. Instead, they create a new item with the proposed changes.

Changing Language Semantics

This type of change is common and may have serious implications. Often, when product maintenance is underway and feature development is concurrently managed, the maintenance person or developer may be unsure or unaware of all dependencies involved in a proposed change. Rather than search all the dependencies associated with the artifact, they do not modify the original item. Instead, they create a new item with the proposed changes.

When modifying an artifact, team members must be aware that subsequent changes can affect the language semantics of the artifact. Figure 8 shows an example of how changes can produce unexpected results.

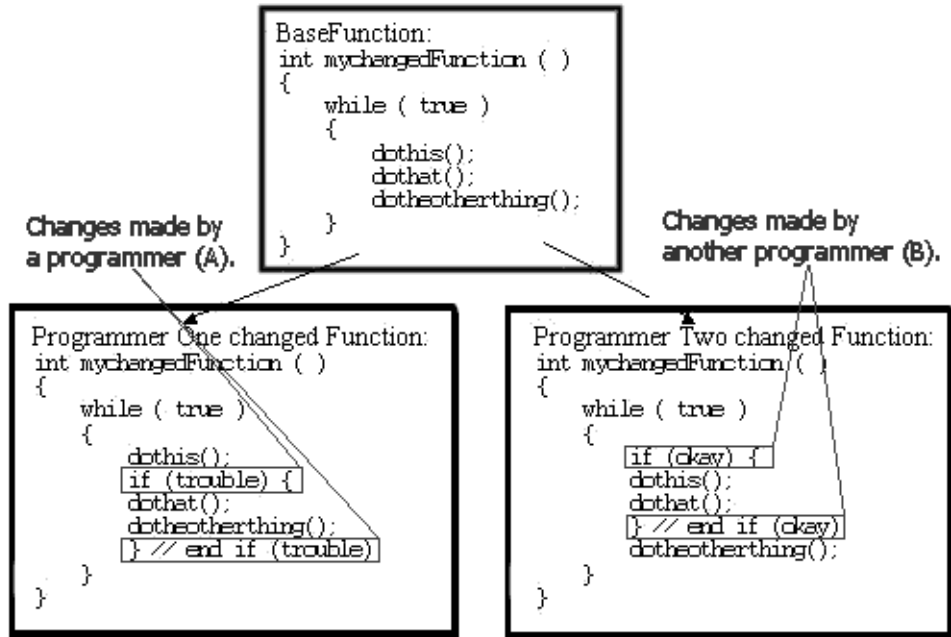


Figure 8 Code Example Showing Changes to Language Semantics

Unless your merge tool knows something of the language semantics used, it may produce a file like that in Figure 9.

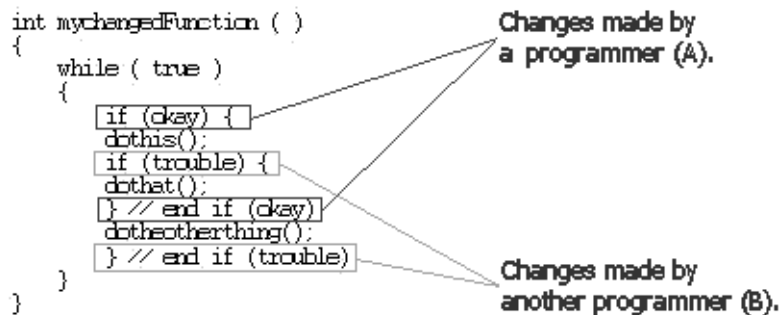


Figure 9 Resulting Artifact After Merging Changes

The function `dothat()` is called only when there is `trouble`, and the function `dotheotherthing()` is called only when `okay` is true. Since most developers do not comment ending braces, it is difficult to identify the problem created by merging. Figure 9 only illustrates a small example. A much more complex code block may present a situation where it is difficult to see the unintended change.

Rational Rose RealTime Model Integrator is aware of the language semantics/model syntax of model files.

Note: Note: *Although Rational Rose RealTime files are text files, the standard text file merge tool is not aware of the Rational Rose RT language semantics or model syntax, and it will corrupt the model files when it attempts to merge them.*

Model Integrator is aware of the Rational Rose RealTime language semantics/model syntax, but not of the language semantics for any language add-ins, or the UML.

Managing Relationships Between Configuration Items

Note: Team members must understand and use the dependencies between configuration items to reduce or prevent unintended changes in the system.

Because most configuration items do not work in isolation from other configuration items, a set of particular versions of configuration items has a set of dependencies. When a set of versions of configuration items changes, the possibility exists that the set of dependencies also changes. It is useful to compare the set of dependencies from one set of versions to a previous set to ensure that dependency changes are intentional.

A set of versions of configuration items is also known as a lineup. Figure 10 shows a generated dependency report for the lineup identified by the label called ALabel. Later, a comparison is made between ALabel and another dependency report generated for the lineup identified by BLabel. Although the dependency reports themselves may be too large to be of any use, a good differencing tool can make it easy to see dependencies modified since a previous stable lineup of the project artifacts.

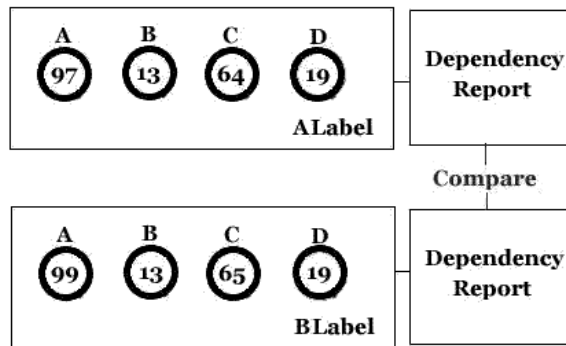


Figure 10 Comparing Dependency Reports

Specific to Rational Rose RealTime, there are several levels of dependencies that must be understood and managed:

- Dependencies between control units in a model. See *Storage of Model Data* on page 47 for more information on model files.
The Rational Rose RealTime Toolset interprets what is loaded into memory as the entire model. When loaded from separate configuration items, the model elements stored on secondary storage must be loaded such that it creates a model where elements are consistent with any corresponding relationships.
- Model element relationships

Managing and Delivering Configuration Items

A specific set of configuration items in their appropriate version (a lineup) must be accessible and reproducible. Test teams, packaging teams, and production end users must be able to work with a release of the entire system that is not in flux. Most version control tools use labelling to produce an environment that contains the desired set of configuration items. Labelling allows you to identify a version of a configuration item with a retrieval marker through association.

Protection of these version sets is important. For example, in a test environment, a small change to a single configuration item can render an entire set of test results unreliable. Often, testing teams only have enough time to perform a specific group of tests once. When an element changes in the test environment while a set of tests are underway, the schedule may not allow for regression testing, and the level of confidence in the test results is downgraded.

Like most one-to-many relationships, a label is often stored many times; once with each configuration item.

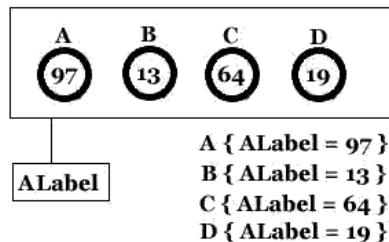


Figure 11 Labelling Configuration Items

Figure 11 shows the following:

- The full set of configuration items are not all labelled at the same time.

Note: *If the label is applied while the lineup changes, this may create an inconsistent state.*

- A configuration item may be overlooked or may not be associated with the label. Sometimes, it is better if the configuration item is not associated with the label. The label associated with a previous version of the configuration item would make the problem difficult to find.

A fixed label is the first primary use of a label, forever identifying a version of a configuration item with a specific label. An example identifier of this type of label is “Build 2000.10.04 night” or “Release 1.0”. It is also useful to include naming convention details, such as the date and time in a label name.

The two types of floating labels (logical and explicit) become associated with different versions of a configuration item.

Over time, a logical floating label is arbitrarily associated with the latest version of a configuration item on a particular branch or stream. For example, “LatestDevelopment” or “JanesLatest”.

An explicit floating label is explicitly assigned to different versions over time, and it is almost always based on the associations of another label and not with the latest versions on a branch or development stream. This means that it is not necessary to “freeze” the configuration items to associate a label with versions already assigned to another label; only the state of the base label must be frozen. For example, Figure 12 shows that the SYSTEMTEST label is associated with version 3 of this particular configuration item.

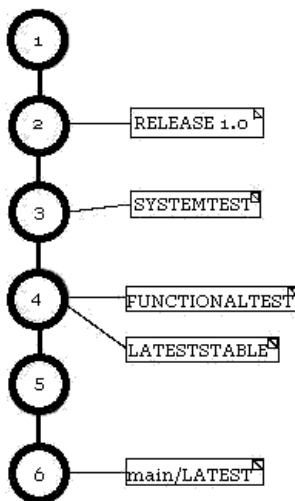


Figure 12 Example of Labelling Items

When the test team for the system is ready, they can associate the label with all the versions associated with FUNCTIONALTEST. No changes should occur to the FUNCTIONALTEST label until the SYSTEMTEST label change is complete. However, assigning LATESTSTABLE with the current versions of all the files on the main branch of development requires that no new main branch versions are added to any of the configuration items until the LATESTSTABLE label change has completed the operation. Since labels can be moved, it is good practice to produce and keep a dated report on the versions associated with important labels for milestones.

Creating and comparing label reports of different dates on a regular basis can reveal trends and areas that require additional testing to ensure quality of volatile areas of the system. Figure 13 shows label reports for two consecutive weeks.

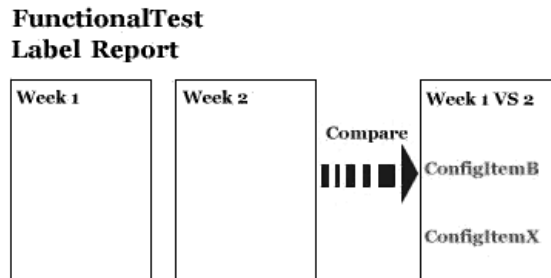


Figure 13 Comparing Reports

Teams looking at a particular lineup of configuration items should retrieve artifacts solely on the selection of configuration items associated with a specific label. Testing in this type of environment quickly identifies overlooked configuration items because of a missing association. It also ensures that all necessary configuration items are included as they are made available to other teams.

Improving Efficiency in Team Development

The implementation of some team development practices can hinder the implementation of other team development goals. Planned activities may be part of the strategy to deal with implementation issues in a team environment.

You can reduce unplanned activities by using an effective strategy that promotes handling conflicts up front. Your configuration management plan should implement a strategy that promotes team development goals with as little impact to team activities as possible. See *Goals of Team Development* on page 1 for more information about specific team development goals.

The stakeholders of the configuration management plan are almost everyone, and their needs vary significantly. The description of the roles and tasks in this document is general and must be customized to suit your particular development organization.

Team Development Roles

This section provides an overview of the typical development roles played by team members in a software project. The organization of the remaining sections elaborate on the logical activities associated with these roles.

Typical Roles

A role is a named behavior of an entity participating in team development, and each role has assigned tasks to complete. There are typically seven roles to consider in your team environment:

- Architect
- Developer
- Integrator
- Tester
- Administrator (for source control)
- Configuration Manager
- Project Manager
- Consumer

Roles Vary Based on Team Size

In a large team environment, several people can be responsible for different team tasks associated with the same role, whereas smaller projects can have only one person responsible for most or all of the tasks for a specific role.

A single person can play multiple roles. A user can perform Architect tasks while working on the initial architecture of the system. Later, they can perform Developer tasks when they are performing detailed implementation. After they make changes, the user can perform Integrator tasks to promote this change to the integration branch of their source control system.

Architect Role

The Architect establishes the overall structure of the model: the grouping of elements into packages, the separation of models into subsystems, and the interfaces between these major groupings. The Architect adapts the structure of the model to reflect the organization of the team.

Architect Tasks:

- *Packages, Models, and Subsystems* on page 97
- *Decomposing a Model into Subsystems* on page 100
- *Splitting a Model* on page 101

See *Organizing a Model (Architect Activities)* on page 97 for a description of these tasks.

Developer Role

A Developer is anyone given check-in and check-out privileges for ongoing system development or system maintenance.

Developer Tasks:

- *Configuring Work Areas* on page 124
- *Getting a Specific Lineup of a Model* on page 124
- *Opening a Model Under Source Control* on page 125
- *Checking Controlled Units In and Out of Source Control* on page 126
- *Building and Running Locally* on page 129
- *Unit Testing within a Subsystem* on page 130
- *Promoting Changes for Integration* on page 132

See *Working with a Model Under Source Control (Developer Tasks)* on page 123 for a description of these tasks.

Product Tester Role

The Test Designer is the principal role in testing, and is responsible for planning, designing, implementing, and evaluating the test.

Tester Tasks:

- *Configuring Work Areas* on page 124
- *Getting a Specific Lineup of a Model* on page 124
- *Opening a Model Under Source Control* on page 125
- Sharing controlled units
- *Building and Running Locally* on page 129
- Generating a test plan and test model
- Implementing test procedures
- Evaluating test coverage, results, effectiveness
- Generating a test evaluation summary

The Tester is responsible for:

- Setting up and executing the test.
- Valuating test execution.
- Recovering from errors.

Integrator Role

An Integrator combines changes from multiple developers to produce an internal build that they can use as the basis for the next set of development activities.

Integrator Tasks:

- Working with an integration model
- Sharing controlled units
- *Building using Automated Scripts* on page 133
- *Building within a Larger Build Procedure* on page 135
- *Reuse of Build Artifacts* on page 136
- *Integrating Changes* on page 137

See *Building and Integrating (Integrator Tasks)* on page 133 for a description of these tasks.

Source Control Administrators

The Source Control Administrator provides the overall source control infrastructure and environment for all required members of the team.

Source Control Administrator Tasks:

- Configuring the source control system for use with Rational Rose RealTime
- Placing a model under source control
- Creating a default workspace file
- Defining work areas
- Defining lineup policies
- Enforcing all other configuration management plan policies

Depending on your team organization, the Integrator role can perform one or more of these tasks.

Configuration Managers

The Configuration Manager provides the overall Configuration Management (CM) infrastructure and environment. The CM function supports the product development activity so that developers, integrators, and testers have:

- Appropriate workspaces to build and test their work.
- All artifacts are available for inclusion in the deployment unit, as required.

The Configuration Manager must ensure that the CM environment facilitates product review, change, and defect tracking activities. The Configuration Manager is ultimately responsible for a comprehensive plan that identifies and deals with pitfalls to team development in the most efficient way for the project.

Project Managers

The Project Manager allocates resources, determines priorities, coordinates interactions with customers and users, and generally keeps the project team focused on the right goal. The Project Manager also establishes a set of practices to ensure the integrity and quality of project artifacts.

Customer Role

The result of the development effort is customer consumable. It is usually impossible to tell from the product the specific versions of source files used to create it.

Most products have some way of providing information to the customer that could assist in determining the versions of source files used to create it. Sometimes, a combination of product version label and build numbers are reported to the customer in an **About** dialog box, much like the Rational Rose RealTime Toolset. Label the particular lineup of source files used to create the product delivered to the customer for possible retrieval. Customers need to know that any bug fixes provided by the maintenance team will contribute to the stability of the product.

Recommendations

Protection of configuration items and the ability to deliver a consistent set of configuration items are the main priorities of the configuration management plan. An implementation of a plan to achieve the other goals should support this ideal.

Use the source control operations supported through Rational Rose RealTime to facilitate the implementation of a greater configuration management plan. For complex projects, a large part of the configuration management strategy that deals with Rational Rose RealTime models may be strict ownership of shared packages.

You may think of shared packages as the building blocks of the system. One Rational Rose RealTime model brings all the building blocks together in a coherent system. Many working models are used with the sole purpose of creating and testing those building blocks.

Delivering the Product

Associate the creation of configuration items and subsequent changes with an activity under the approval of a single point of contact. This single point of contact is sometimes implemented as a Change Control Board (which may have an alternate name within your organization) that is aware of all the requirements and activities that change the system. Their awareness of the changes to the system at a high level can help them identify functional or esthetic conflicts, or nonconformance.

Source Control Fundamentals

Chapter 3, called *Source Control Fundamentals* on page 79, specifies the source control operations supported from Rational Rose RealTime. It outlines some of the differences in view-based and file-based source control systems. There is also a discussion on versioning strategies.

The ability to associate labels and create a lineup exists in both types of source control systems. Using a parallel stream versioning strategy while maintaining a single stream versioning policy, provides the safety inherent in single stream versioning strategies, and also the ability to control parallel development of the same artifacts among different teams.

Any source control tool that allows branching is capable of supporting a parallel stream versioning strategy. An example of appropriate streams of development are:

- Development streams, where developers make changes to the configuration items.
- Integration stream (implementing requirements and features) managed by Integrators.
- Product version maintenance streams (providing fixes for bugs/defects identified after release date) also managed by Integrators.

Include a maintenance stream for every product version currently supported by your organization. When support for the specific product version is concluded, these streams should end.

Note: *You can use merge tools, such as Model Integrator, for merging simple, non-conflicting changes. However, because of their limited semantic support, we do **not** recommend that you use automated merge tools when there are many conflicting changes.*

Bugs and defects reported against a version of the product should be evaluated against the product under continued development in the new development stream. Other versions of the product that may be affected by the bug/defect are under continued support. Apply corrections to all affected versions through a manual merge, or through focused merges.

If you implement a parallel stream versioning strategy, maintain virtual single streams within the parallel streams. For example, Figure 14 shows a version tree history for a configuration item. A branching of development effort occurs at version 1.0, and version 2.0 of the configuration item.

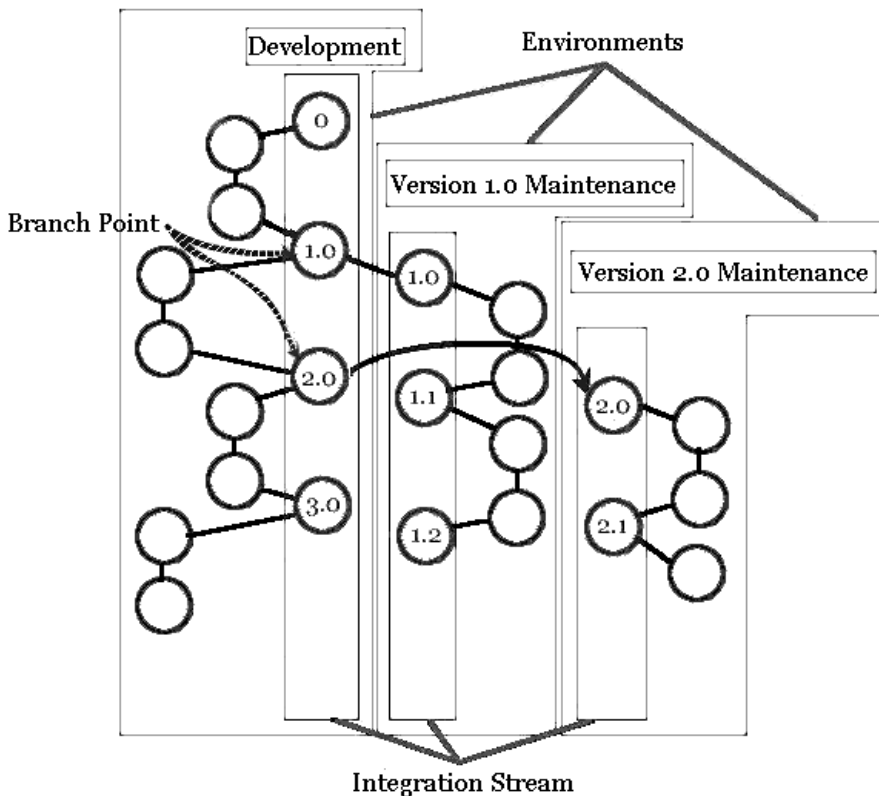


Figure 14 Parallel Stream Versioning Strategy

Only one side of the branch is checked back into that integration stream. The Integrator uses the main streams of development and may be unaware of the details of individual changes. Therefore, from the perspective of these streams, they are a single stream of development only receiving updates from one source that has permission to modify the next version in the stream. If you require merging, perform it outside of these integration streams, and sanity test it before integrating it as a new version.

Do not associate product verification labels and packaging or deployment labels with versions outside these main integration streams of development. When working with files such as test scripts that are version controlled, consider these files as if they were in a separate project.

You may have separate streams for the development and maintenance of these scripts as well, but this should be thought of as a different project than the one it supports from a version control perspective. That supporting system may have logical ties or parallels with the product under development.

Preempting Conflicts

You want to minimize more than one concurrent check-out of a configuration item. If this strategy results in unacceptable contention for a configuration item, or a dead-lock occurs, put overrides in place to deal with the contention.

A dead-lock occurs when Developer A requires a configuration item checked out to Developer B to finish his work, and Developer B requires the configuration item that Developer A currently has checked out. Because this is done up front, there is an awareness that changes are being concurrently made to the same configuration item, and these changes can be managed to minimize the likelihood of unintended change.

This type of concurrent work must occur outside the main development streams. When it occurs, resolve this type of situation as quickly as possible and provide adequate testing of the configuration item following the period of concurrent change, to ensure no unintended changes occurred as a result.

The Rational Rose RealTime shared package capability, defined in chapter 2, *Storage of Model Data* on page 47, can implement an ownership strategy to limit the scope of implicit permissions to change configuration items.

Packaging Strategy

Think of a model as a structure containing packages, that in turn contain the design of the system. Work models create packages that contain the particular part of the system a team or developer is responsible for. These work models can also see other required packages on a read-only basis by using the Rational Rose RealTime shared package facility. An integration model can then reference the work contributions by team members exclusively as shared packages.

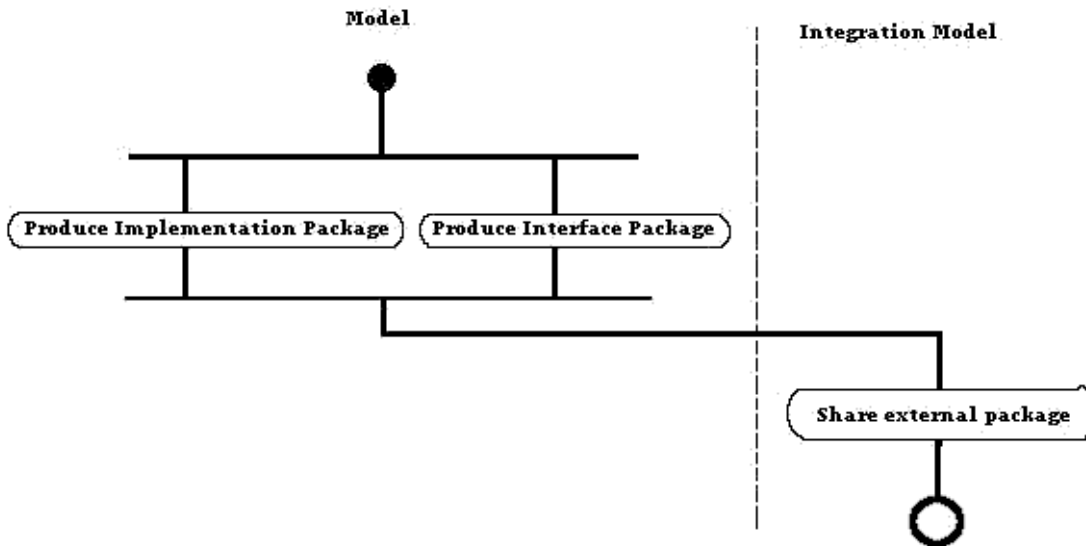


Figure 15 Packaging Strategies

Managing Dependencies

To effectively manage changes to the dependencies in your system, you must create and enforce your own team processes.

Note: *For Rational Rose RealTime projects, you must identify the dependencies between control units in a model.*

See chapter 2, *Storage of Model Data* on page 47 for more information on model files.

Additionally, it is good practice for you to identify your model element relationships. And, it would be beneficial if you also had an understanding of the generated source dependencies and data dependencies.

If you do not have a formal reporting mechanism that automatically identifies these dependencies, every change must be addressed to ensure that dependencies are researched and assessed as a result of the change.

Labeling

When considering labelling, we recommend the following:

- Establish an environment for each group or individual that will work with a specific set of configuration items in isolation from other changes for *any* length of time. For most version control tools, this is established with directories containing a copy of the appropriate version of the configuration item identified through a movable label. The team member performs work on artifacts in these directories, and this set of directories is also called the sandbox.
- When using file-based version control tools in Unix systems, developers can configure a directory that references the shared work through soft links. When team members modify the reference in the directory, the link is broken and it is replaced in the sandbox by the modified file.
- Create dated reports for each floating label on a regular basis, listing all configuration items associated with the label and the associated version. We recommend that you add the report to your version control system. You can use the data from the report to identify how the set of configuration items changed over time, and

to help you identify volatile and stable elements of the system. Fixed labels do not require this type of report. For a label associated with a set of configuration items that do not change often, you can reduce the frequency to some appropriate interval, or on an ad-hoc basis.

- Define your labeling strategy as much as possible before you begin. Use a naming convention so that everyone can understand the labels.
- Identify labels that may require protection from modification, and those labels that may require restricted access.

When Merging is Necessary

Merging is necessary when an awareness exists that concurrent development may result in conflicting changes. Perform the merge as often as possible. Each developer involved in a concurrent change must regularly work with a merged version of the ongoing work to identify adverse or unintended change.

The intention is to reduce the amount of lost work that can occur when conflicts arise. A conflict identified early reduces the amount of re-work necessary. This kind of concurrent work on the same artifacts must be done in isolation from other work.

The way ClearCase facilitates integration branches, it is wise to choose a special integration stream for the concurrent changes to a configuration item. This isolates the remaining artifacts in your system (which uses mutual exclusion) from these changes until the configuration item can go through extensive quality verification.

With other sandbox type systems, one developer merges other developer's work, and then provides the merged version to the other developer.

After every merge, assess changes to semantic relationships and other dependencies.

Merging Detail Code Before Using Model Integrator

In Model Integrator, when models include detailed code, you must select one contributor over another.

When comparing models, Model Integrator looks at the model elements, which it then compares with other elements based on properties. For example, for a base model, called **ModelX**, there are two contributors, Contrib1 and Contrib2. If property A of element A from Contrib1 is equal to element A of property A for Contrib2, then all code associated with the transition is in a single property; the base model has element A.

When property A of element A from Contrib1 is **not** equal to element A of property A for Contrib2, Model Integrator detects the difference and allows you to select a contributor. Selecting a contributor causes the base model to change. The result is a merged model with the changes from a single contributor.

To merge before using Model Integrator:

1. Abandon the merge.
2. Export the code from Contrib1 to a file.
3. Export the code from Contrib2 to another file.
4. Use another merge tool, such as Rational ClearCase, to merge the source code from the two files.
5. Import the merged source code into Contrib1 in Rational Rose RealTime Model Integrator.
6. Use Model Integrator to merge Contrib1.

Artifact Freeze

Occasionally, a set of configuration items may require protection from more than one change. We recommend that all changes currently underway be completed, verified, shared, and re-verified. Do not allow any additional changes to that set. This is sometimes referred to as a freeze. Following the freeze period, refresh the work areas that use that set of shared work.

The artifact freeze allows testing to occur in one environment using only shared configuration item versions before promoting those versions to more formal testing environments. The duration of the freeze must be long enough to label all configuration items with a fixed label, such as “Delivered to Test on day 58”. This labelling occurs prior to the movable “Test” label assigned the same versions as those assigned to “Delivered to Test on day 58”.

Because “Delivered to Test on day 58” is a fixed label, it is not necessary to produce a label report at this time. However, if reports on the dependencies between these configuration items is available, generate and check in the set of configuration items with the fixed label.

A Special Type of Artifact Freeze

If a change to the model results in a change between the dependencies of a model’s control units, check in all model-related configuration items before making the change. Ensure that the model is checked-out to allow for the change, and checked-in after the change. Then, refresh work areas with the new copy of the model files. Configuration items checked-in as a result of this special type of freeze may now be checked-out to continue the work. This type of a change is a change to the model’s architecture, and not necessarily a change to the architecture being modeled. Possible scenarios include:

- Changing how to store a model file, adding control units, or reducing the number of control units. You can make these changes by clicking **File > Control Unit** or **File > Uncontrol Unit** from the pop-up menu on model elements.
- Moving an element from one package to another package where this represents a move from one control unit to another control unit.

Building and Executing a Rational Rose RealTime Model

Rational Rose RealTime models are executable. To execute a model, a user must compile/build a component in the model to produce the executable.

On a large project, we recommend sharing build artifacts, such as generated code and object files, to reduce the build time for developers. Rational Rose RealTime has features to support build reuse, and the ability to integrate with other tools to leverage their features (for example, the wink-in capabilities of ClearCase clearmake).

The Integrator is responsible for many of the ‘infrastructure’ build tasks, while the Architect and Developer roles typically participate in local build tasks.

On large projects, you can generate, then share an external library interface. This feature allows you to reuse builds; you only need to rebuild when there are changes. For information on external library interfaces, see *Generating and Sharing an External Library Interface* in the *C++ Reference*.

Advanced Concepts and Heuristics

This section includes additional information about advanced concepts and heuristics in the following areas:

- *Moving Controlled Model Elements* on page 31
- *Primary and Secondary Edits* on page 32
- *Understanding Blue Deltas* on page 35
- *Parallel Development* on page 36
- *Model Integrator* on page 37
- *Using Rational ClearCase Multi-Site* on page 38
- *Using Rational ClearCase UCM* on page 38
- *Unique Ids* on page 39
- *Rational Quality Architect - RealTime Edition* on page 43
- *Additional Heuristics for Team Development* on page 43
- *Additional Recommendations* on page 45

Moving Controlled Model Elements

When a model element moves from one package to another, Rational Rose RealTime does not move the file corresponding to the model element into its new directory.

When a UML package is assigned a CM system label, it later performs the operation on the directory and all its contents. However, if the controlled unit moved, its corresponding element will not be labeled correctly.

Considerations

In ClearCase, the relationship between a file element and directory elements is such that an element may be in multiple directories at the same time, possibly even in the same view. This does not necessarily complicate things for the toolset, but requires careful consideration.

A Rational Rose RealTime model element may be saved as two distinctly named Rational ClearCase elements.

Heuristics

Until your system architecture is stable, use package-level granularity rather than class-level granularity. When the system architecture is stable, use Class-level granularity. This level of granularity reduces the probability of having to merge units later.

Renaming a Controlled Model Element

When the name of a controlled package, diagram, or classifier changes, the storage unit file is not changed.

Primary and Secondary Edits

When a change occurs in a model, it affects the immediate controlled unit and Rational Rose RealTime requires that you check-out the model. Often, more than one controlled unit may be affected by the original change and Rational Rose RealTime also requires that you check-out these controlled units. If you cannot check-out all of the affected controlled units, the original change is not permitted.

The original change and all required (affected) changes are called primary edits. Primary edits must be made effective at some point otherwise, they will cause inconsistencies in the model that cannot be resolved by Rational Rose RealTime as it loads the model.

Secondary edits involve changes as a result of primary edits, but do not have to be completed at the time as the primary edits. Rational Rose RealTime can resolve secondary edits as it loads the model, but the fixed model only persists in memory. You must save the affected secondary edit control units for the changes to persist between model loading. Since we assume a highly controlled environment, this means that the affected controlled units must be checked out, then saved.

In summary, in a highly controlled environment, a single edit can often affect other controlled units. As a result, some controlled units may require immediate check-out, and some can be resolved later to have a consistent model across all its controlled elements.

A complication to this process may occur in a project exercising best practices; where the person making the original primary edit may only own the controlled unit that is immediately affected. The other primary edit controlled units may belong to another team member. Additionally, the same holds true for the secondary edit controlled units.

To handle the secondary edits, let the owner accept the changes and make them persistent. All other users can let Rational Rose RealTime resolve secondary edits when loading the model, and they can choose to ignore prompts to save the changed controlled units they do not own.

Primary edits that do not impact more than one controlled unit are trivial and only require that the user making the change be the one who owns the affected controlled unit.

Primary edits that involve more than one controlled unit are the most troublesome, and is more common in projects where specific Best Practice guidelines are not followed. When these situations arise, there are typically two approaches to implementing change:

- The user making the primary edits performs a private check-out of all affected controlled units. The affected controlled units are then later merged into another stream, possibly at integration time. Unfortunately, the type of merging that must be performed is less predictable and planned. It is difficult for any tool to properly and completely address the complexities of these merges in a reliable and robust manner. Such is the case with the Model Integrator.

- The user making the primary edits coordinates with the owners of the other affected controlled units to implement a change. Ultimately, to avoid the necessity for a merge later. This approach is difficult to do and does not take advantage of the change management features of the tools in the tool chain. An important consideration for this approach to implementing a change, is who will do the changes, and when.

Model Conversion

The following guidelines may help minimize the occurrence of problems when dealing with primary edits:

- Every controlled unit must have an owner.
- **Assign one user a number of controlled units that are related and may involve a lot of coupling, particularly inheritance coupling.**
- Use components and layers in the architecture to reduce coupling and minimize the number of dependent elements.
- The interaction between unsaved secondary edits and blue delta syndrome affects the ability to build the model. Therefore, it is very important to resolve secondary edits as soon as possible. Blue deltas represent a changes that cannot be resolved by Model Integrator.
- Do not over populate diagrams with information; try to focus on model elements having the same ownership. For example, focus on Class X and its subclasses, or Class Y and its dependencies. Focused diagrams help reduce the effects of Primary Edits and merge conflicts in Rational Rose RealTime Model Integrator.

Possible Solutions:

- The owners of the affected controlled unit must save secondary edits, or save the controlled unit with the permission of the owner of the affected controlled unit.

Heuristics:

Some model and architecture characteristics can make secondary edits better or worse. Whenever possible, use loosely coupled architectures as in the access and resource manager patterns. Also, avoid reusing packages from a production model into a consumer model (shared external packages). Secondary edits are not recognized until the consumer model is refreshed/loaded, which may not occur until integration time.

Solutions:

- There are consequences associated with not saving secondary edits in a persistent manner. For controlled units that have not had their underlying representation updated for consistency with the associated primary edits, Rational Rose RealTime prompts the team member with warnings when the control unit(s) are first loaded. These changes do not cause irreparable model inconsistencies; however, unsaved secondary edits have a consequence on builds. If blue deltas are not saved, then a build is not possible.
- Rational Rose RealTime lets you build without committing changes to source control.

Note: *The worst case scenario is when an edit performed in one model forces a primary edit in a controlled unit in another model (or other models), but not in the model where the initial edit was performed. For this case, the owner of a model that is affected by a primary edit in another model, must perform the integration of changes.*

Understanding Blue Deltas

Blue Deltas occur when changes are made to a model, and due to the control over the unit by the CM system, the file cannot be saved. Rational Rose RealTime attempts to save all controlled units where the memory image differs from the image on the file system, but is unsuccessful because the controlled unit is read-only. This is one of the negative effects of unsaved secondary edits.

Parallel Development

Parallel Development is a term that sets high expectations regarding collaborative development, where there is a need for multiple users to work together on a common set of artifacts to achieve the same goals.

When collaborating on a common set of artifacts, consider the following approaches to collaborative development:

- **When more than one user needs to make changes to the same artifact, they must share the artifact; the changes are made serially, one after the other. Although this is the most reliable approach, it is perceived by most users as not being most efficient. This approach can be managed using the check-in and check-out features of most CM systems.**
- When more than one user needs to make changes to the same artifact, they can make the changes at the same time. The changes are merged back into one artifact at a later date. The benefit of this approach is that work goes on in parallel, and it saves time. The problem is that arbitrary and uncoordinated changes on the copies of the same artifact can be difficult to resolve during the merge process. In fact, they may never be resolved, and the changes from only one contributor are accepted and from the other, discarded.

***Note:** We recommend that these types of changes be coordinated and merged often.*

The development process and tool chain can have a significant impact on the opportunity to use and the effectiveness of the second approach. The second approach is known as Parallel Development. For the purpose of this discussion, the term Parallel Development refers specifically to this second approach to collaborative development.

It is unrealistic to expect to employ parallel development without any constraint or guidance. Too often, this technique is used without coordination or planning. Sophisticated tools, such as Rational ClearCase, may not be properly used and can lead to this misperception. The design artifacts at the center of collaborative development have complex interrelationships within them, and between them. These higher level abstractions and concepts are not easily, and cannot arbitrarily, be merged without some experience. Fortunately, when team members are working within a well-defined process, and there is a clear definition of roles and responsibilities, most changes made in parallel are done in a complementary manner.

A certain amount of conflicting changes are inevitable. You can resolve the changes by choosing one or the other. These conflicting changes must be expected and their frequency should be minimized. If they are unexpected, it may be counterproductive and time is being wasted by changes that will not be discarded.

The following guidelines will help maximize the efficiency and productivity of a process that employs parallel development:

- Scrutinize and minimize the occurrence of every conflicting change in the merge.
- Create a well documented and communicated development plan to help ensure that every developer knows how they are contributing and what they will implement. This helps minimize duplication of effort, even at the lowest level of detail.
- Establish clear ownership of design artifacts, and use source control to enforce it.
- Invest time into understanding what the Rational Rose RealTime Model Integrator will and will not do during a merge.
- Follow all guidance specific to the Rational Rose RealTime Model Integrator regarding the types of changes that it can reliably merge.
- Resolve all issues relating to merging parallel changes prior to integration.

Model Integrator

The Rational Rose Model Integrator is a powerful tool that manages the merging and differencing of models at the Rational Rose RealTime meta-model level. It is not a visual model or UML semantic-level merge tool, therefore it lacks a number of features that can make the merging of models more efficient and more accurate.

For every use-case of Model Integrator that fails to do what you may expect, there are many other use-cases that do add value or do what is expected, and will save time. When using Model Integrator, you must understand what it can do efficiently and properly, and what should be avoided.

When you plan for a graphical change (a layout change) to a diagram within a model, only one person should make this change. This ensures that during the merge process, all of the graphical changes are accepted by one contributor and merging at a lower level of detail is not allowed.

Using Rational ClearCase Multi-Site

When a team follows best practices, (for example, being careful about artifact ownership) they can use Rational ClearCase Multi-Site to work on separate branches.

Rational ClearCase Multi-Site is a powerful tool that can help you with the challenges of a distributed team development. When using Rational ClearCase Multi-Site, you must consider the following:

- Rational ClearCase Multi-Site has a restriction that a branch is owned by a site.
- Only developers on that specific site can check out to that particular branch.

Using Rational ClearCase UCM

Unified Change Management (UCM) is an activity-based process for managing changes to all software artifacts. It supports a change management usage model and is a key component of the Rational Unified Process (RUP). The RUP is a comprehensive framework for delivering software development best practices. You can use UCM to unify cross-functional teams and provide meaningful, common data access, along with processes and tools, that enable teams to manage change, monitor quality, and communicate more effectively from requirements to release.

For larger teams, you can use a combination of UCM and “base” ClearCase functionality on a project-by-project basis. However, you may want to consider the following facts about UCM:

- Existing Rational ClearCase users can upgrade from their current version to UCM and continue to work the same way by choosing not to use UCM. New Rational ClearCase users can use the UCM model or implement a more traditional model using base ClearCase functionality.
- If you want to use the new UCM capabilities for a subset of your current projects, it is possible for some teams to use UCM while others do not, even if they share some or all of the same code.
- Additional capabilities are available with the combination of Rational ClearCase and ClearQuest-enabled UCM.

- The UCM model provides an enhanced integration between Rational ClearCase and Rational ClearQuest that is not available outside of UCM.
- Rational ClearCase LT supports UCM and offers an activity-based process model for managing change and controlling workflow. This provides for a seamless upgrade path from the basic source control management functions provided by ClearCase LT to the enterprise capabilities featured in ClearCase. With Rational ClearCase LT, you can also take advantage of UCM to manage changes to artifacts other than source code, including requirements documents, test scripts, and design models.

Unique Ids

Unique ids are unique internal names associated with model elements. They are used internally by Rational Rose RealTime, and not all model elements require unique ids. Rational Rose RealTime includes a feature that helps Model Integrator by generating unique ids for those model elements that would otherwise not require them, for internal use. For Model Integrator, an element with a unique id is easier to merge.

RRTEI users will find traceability easier when they set this option. Unique ids improve the traceability of model elements of other tool integrations that use RRTEI.

It is necessary to plan and choose when to incorporate the new unique ids into the project model since virtually all controlled units will be modified implicitly. Additionally, the generated new ids are dependent on time and location. For example, generating unique ids for a given model at different times, or on different machines, produces different ids.

The following model elements do not have unique ids, unless you set this option:

- Protocol In Signals ()
- Protocol Out Signals ()
- States (CompositeState)
- Capsule Roles (CapsuleRole)
- Ports (Port)
- Port Roles (PortRole)

- Capsule Structure diagram (CapsuleStructure)
- Classifier Role (ClassifierRole)
- Transitions (Transition)
- Junction Point (JunctionPoint)
- Choice Point (ChoicePoint)
- Connectors (Connector)
- (Guards)
- (Events)
- (EventGuards)
- Parameters ()
- Element hyperlinks (ExternalDocument)

Caution: *We strongly recommend any team involved in parallel development use this option.*

Setting this option creates unique ids for model elements that currently do not have them. This typically affects most of the model, so you will be prompted to check out those parts when setting this option.

When saving the model, the size of the affected file increases by approximately 20%, and the time to load the model also increases.

Caution: *Do not set this option in multiple streams as shown in Figure 16; otherwise, objects with similar characteristics will be treated differently since their unique id's will differ.*

Figure 16 shows an example of an incorrect merge scenario.

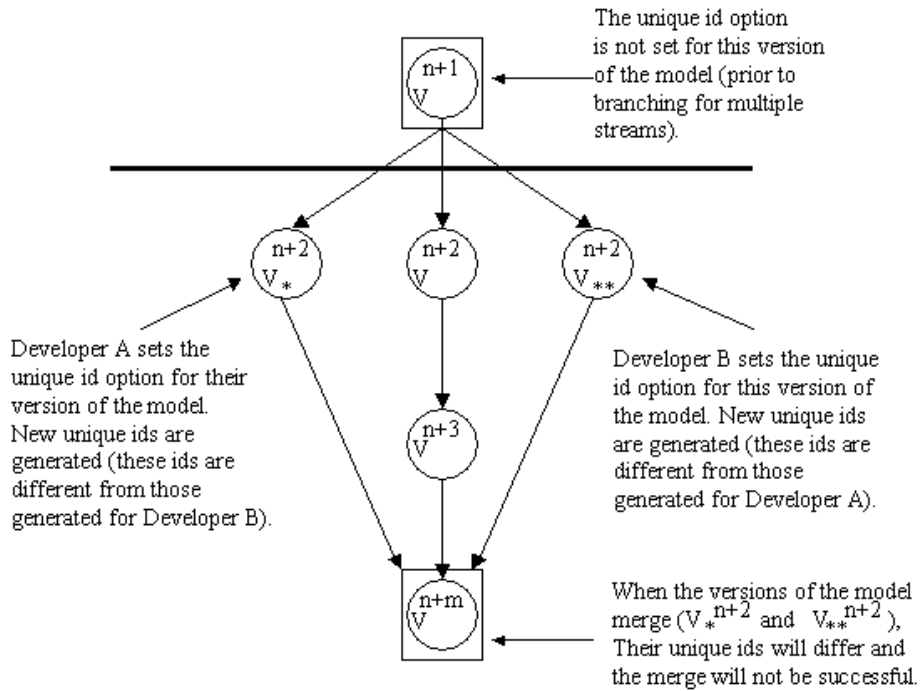


Figure 16 Incorrect Merge Scenario

An example of when to set this option is shown in Figure 17.

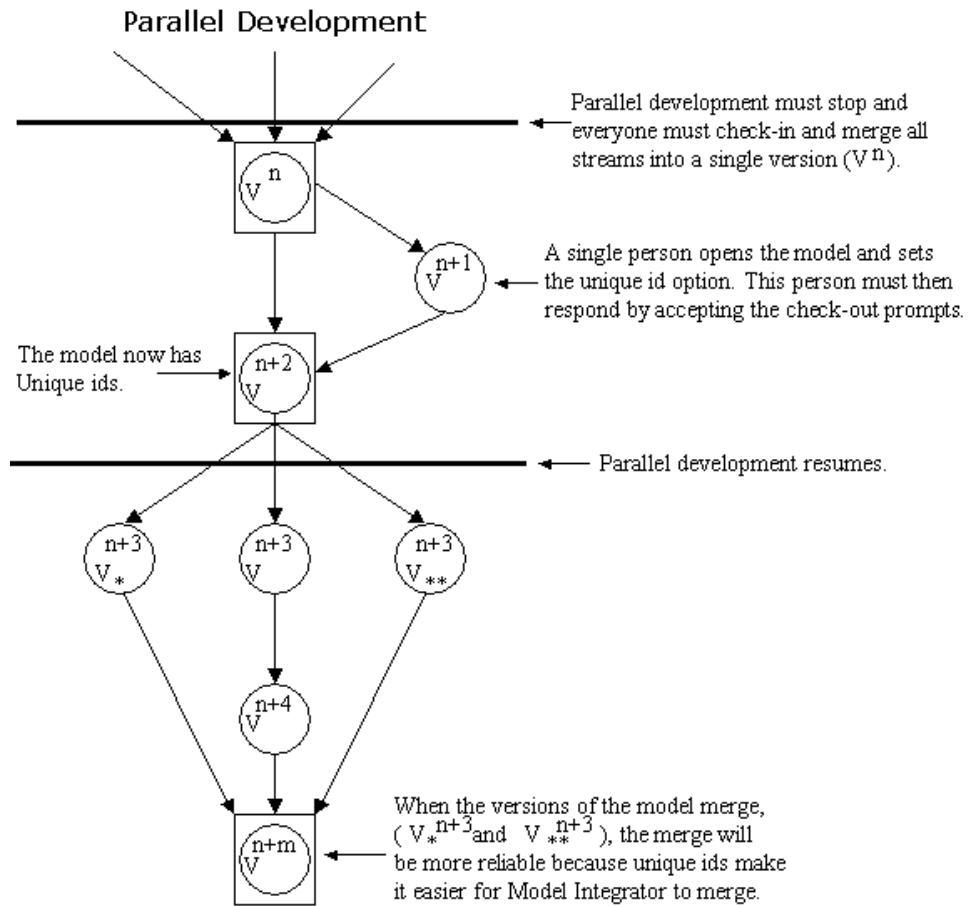


Figure 17 A Correct Merge Scenario

Note: This option must be set prior to branching.

For information on how to enable the Unique ids, see Model Specification in the online help.

To clear the unique id option, follow the same procedure in Figure 17.

Caution: If you clear this option, your merge results will not be as reliable.

Rational Quality Architect - RealTime Edition

When using the Rational Quality Architect - RealTime Edition in a CM controlled environment, ensure that you check-out the following:

- Top level model file
- Logical view
- Component view
- Deployment view

When several developers are working with Rational Quality Architect - RealTime Edition in parallel, they must have separate models or be on different branches.

You can use scratchpad packages to avoid complications in using Rational Quality Architect - RealTime Edition in a highly source controlled project. You can only specify the package that the generated model goes into; you have no control of the component or component instance.

Additional Heuristics for Team Development

- Begin with a high level of granularity for controlled units when an area of a model is immature. As the area of a model becomes more mature, then its level of granularity can be lowered, possibly to the capsule and diagram level.
- During the architecture phase, the granularity is coarse. When the architecture is released to the designers, decrease the granularity to manageable pieces for efficient team development.
- Use a layered architecture where the coupling between layers is minimal and well-defined. This kind of architecture is also called loosely coupled.
- Define the interfaces between layers of the architecture early and minimize changes to these interface elements.
- Release the interfaces and associated components at a layer boundary separately and ensure that they have their own test and release schedule. There should be one or more separately released components in each layer.
- Every controlled unit should have only one owner.
- Plan for conflicting merges and attempt to minimize them throughout the development life-cycle.

- Only merge controlled units with primary edits back into the integration stream.
- If the system is sufficiently complex, divide each layer into subsystems.
- Ensure that subsystems have a well-defined and minimal interface to other subsystems.
- Subsystems are not necessarily confined to one layer. Interfaces at lower and higher levels of abstractions should coincide with one of the architectural layers. Subsystems may encapsulate their own set of layers that satisfy particular objectives.
- Use different models (.rtmdl files) to develop different subsystems and share them in the top-level model.
- Employ at least three streams of development: release stream, integration stream, and developer stream.
- Place a new part of a model under source control after it has had some (minimal) testing.
- **As an inheritance heuristic, do not use a classifier to derive classes (for example, Capsules, Classes, and Protocols) until its superclass has a stable design element.**
- Do not make frequent or large changes to a superclass.
 - Subsystem interfaces (protocols and data classes) may need to be modified by both users, but changes should be planned, controlled, and authorized by owner (or group).
 - Appoint one responsible person for each interface. This person is the only one that can change the interface. For example, all requests for changes must be sent to this single team member for them to make the required change.

Additional Recommendations

- Perform integration at least once a week.
- Use a build coordinator to ensure that all required components make it into the build. The build coordinator has granularity to the capsule protocol level.
- The following structure represents a recommended general layout of a model, with particular focus on the Logical view:

```
+UCVP
+LVP
+Project/Model Name
  +Layer1
    +SubSystem1
    +SubSystemn
    +CommonSubSystemProtocols
    +CommonSubSystemDataClasses
    +TopCapsules
  +LayerN
    +SubSystem1
    +SubSystemn
    +CommonSubSystemProtocols
    +CommonSubSystemDataClasses
    +TopCapsules
  +CommonLayerProtocols
  +CommonLayerDataClasses
  +TopCapsules
+CVP
+DVP
```




Chapter 2

Storage of Model Data

Contents

This chapter is organized as follows:

- *Storing Model Data* on page 47
- *What is a Controllable Element and a Controllable Unit?* on page 48
- *Sharing Controlled Units* on page 61
- *Creating Sharable Controlled Units* on page 64
- *Working with Controlled Units* on page 65
- *Moving Controlled Units* on page 67
- *Export Controllable Elements from a Model to a File* on page 68
- *Import Controllable Elements from a File to a Model* on page 69
- *Add an Existing Controlled Unit to a Model* on page 70
- *Share an Existing Controlled Unit into a Model* on page 71
- *Produce a Single Model File from a Model with Many Units* on page 73
- *Virtual Path Maps* on page 73

Storing Model Data

The following sections describe how Rational Rose RealTime stores model data. This information is useful for understanding how Rational Rose RealTime interacts with source control systems, as well as for learning the capabilities that may impact the performance of Toolset operations that read and write model data to files.

By default, Rational Rose RealTime saves a model as one file. When multiple users work on the model at the same time, there is reduced contention for files if the model is stored as many small files rather than one large file. Rational Rose RealTime supports users saving models as a series of individual files, called controlled units, rather than one large file. As a general guideline, the more granular the storage, the better it is for large team development because each file is a potential bottleneck when multiple users work on it concurrently.

Rational Rose RealTime provides a great deal of flexibility in how the model data is stored as files. A very simple model may be stored in a single file and a very large model can be stored in hundreds or thousands of files. Understanding how to control the storage of model data is important for using Rational Rose RealTime successfully.

What is a Controllable Element and a Controllable Unit?

A *controllable element* is a Rational Rose RealTime element (for example, class, package, class diagram) that supports being controlled/saved to a separate PetalRT file, independent of its parent element. The term *controllable unit*, or just *unit*, is generally used to refer to the file itself as opposed to the element.

Each controllable element has a "controlled" flag which determines if it is stored in its own unit file or in the same file as its parent. This flag can be changed for a specified element using the **File > Control Unit** and **File > Uncontrol Unit** menu items in the browser context menu. See *Controlling a Subset of the Controllable Elements* on page 65 for the details of this process. The initial setting for the controlled flag is based on the setting of the "Control new child units" flag of the parent unit.

The model itself is always a controlled unit. Thus, a model is made up of a hierarchical set of controlled units where the granularity of the controlled units is fully configurable by the user.

If an element is controlled, then its Specification Dialog includes a **Unit Information** tab which lists the associated file name and other settings. See *Unit Information Tab* on page 57.

The Rational Rose RealTime browser uses the following icons for controlled units (the icons are much smaller when shown in the browser)



Figure 18 Browser Icons for Controlled Units

The first icon indicates a controlled unit that has been saved.

The second icon indicates a controlled unit that has unsaved changes. This blue triangle is often referred to as a "delta". (Delta is a greek letter drawn as a triangle and means "change".) The blue delta symbol indicates that the version of a unit in the toolset is different from the file on disk from which it was loaded.

The last icon indicates a shared external package, such as the RTClasses package.

Controllable elements that are not individually controlled do not have these icons (but they will have their respective element type icon, such as package or diagram). Additional browser icons relating to source control status are described in *Source Control Status* on page 80.

Figure 19 shows an example of these icons in the browser. If we look at the elements in the Logical View, then we see:

- RTCClasses and RTClasses are shared external packages,
- Collaboration1 is a controlled unit with unsaved changes,

- the Main class diagram and NewClass1 are controlled units that were saved,
- NewPackage1 is not a controlled unit since it does not have a controlled unit icon

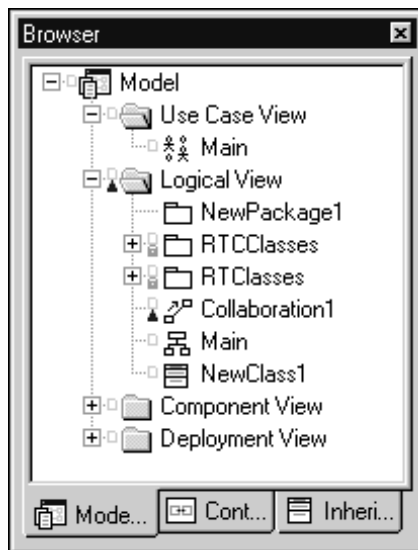


Figure 19 Browser Icons Example

What Elements Can Be Controlled?

Rational Rose RealTime supports the following controlling elements as separate units (file extensions for corresponding files are shown in parentheses):

- Model (.rtmdl)
- Package (.rtlogpkg), (includes Use Case Packages and Logical Packages)
- Class Diagram (.rtclassdgm) (includes Use Case Diagram)
- Class (.rtclass)
- Capsule (.rtclass)
- Protocol (.rtclass)
- Use Case (.rtclass)
- Actor (.rtclass)

- Collaboration (.rtcollab)
- Component Package (.rtcmppkg)
- Component Diagram (.rtcmpdgm)
- Component (.rtcmp)
- Deployment Package (.rtdeploy)
- Deployment Diagram (.rtdeploydgm)
- Processor (.rtprocsr)
- Device (.rtdev)

Parent and Child Controlled Elements

Some types of controlled elements are containers for other controlled elements. For example, a logical package is a container for classes, capsules, protocols, class diagrams, collaborations, and other logical packages. The package is often called the parent for the controlled elements it contains (which are often called children).

When you create a controlled unit from a child element and then save it, its contents are moved from its parent unit's file and stored in a new file. Thus, the original file will no longer hold the contents of the child. Instead, the original file only references the new controlled unit file.

The complete list of container controlled elements and their possible child controlled elements are:

- model: package, component package, deployment package
- package: package, class diagram, use case diagram, class, capsule, protocol, use case, actor, collaboration
- component package: component package, component diagram, component
- deployment package: deployment package, deployment diagram, processor, device

The "Control new child units" flag for a parent unit specifies whether children of that unit are controlled by default.

If a parent element is not a controlled unit, then its child elements cannot be controlled units. Similarly, when a package is uncontrolled, all children will also be uncontrolled.

A package that is a controlled unit has the following file system elements:

- a file with model specifications
- a directory created to save child units

Directory Structure for Model Data

If a package is a controlled unit, then there is a directory created to contain the saved child units. As an example, assume we have the following model structure:

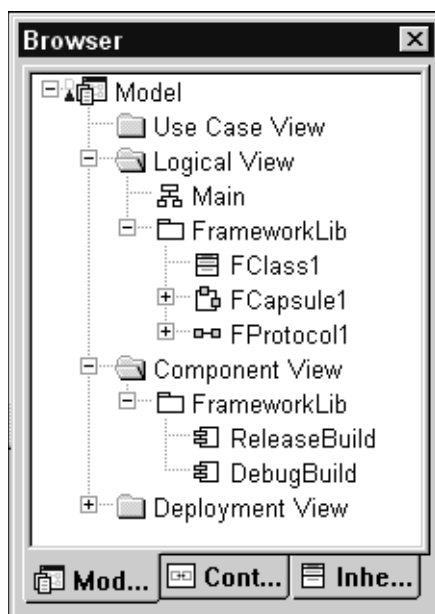


Figure 20 *Sample model structure*

What is a Controllable Element and a Controllable Unit?

If all controllable elements in this model are controlled units, then the default directory structure would look like:

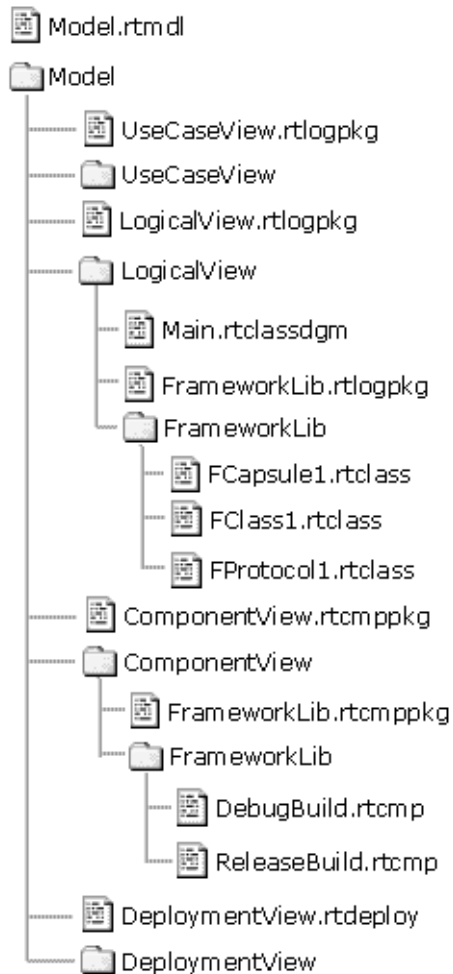


Figure 21 Directory structure for sample model

As an alternative, if we decided to reduce the number of controlled units as follows:

- FCapsule1, FClass1, and FProtocol1 should not be independently controlled; instead, they should be saved in the same unit as the FrameworkLib logical package;
- ReleaseBuild and DebugBuild should not be independently controlled; instead, they should be saved in the same unit as the FrameworkLib component package.

These changes would result in the following directory structure:

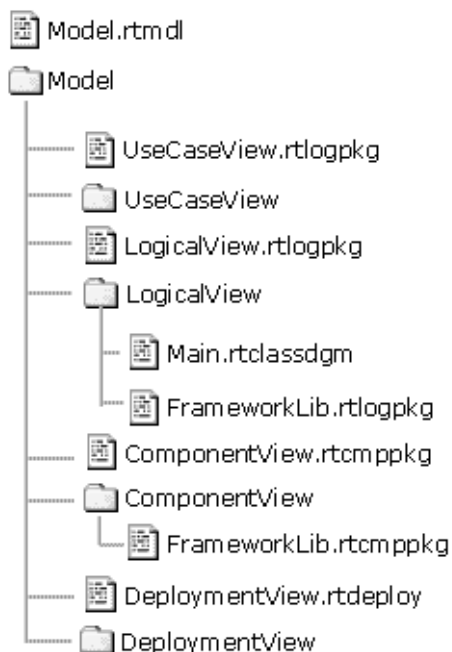


Figure 22 Sample directory after granularity is reduced

The FrameworkLib.rtlogpkg file contains FrameworkLib, FClass1, FCapsule1, and FProtocol1. The FrameworkLib.rtcmpkg file contains DebugBuild and ReleaseBuild.

File Names for Controlled Units

Rational Rose RealTime can generate a default name for the file used for a controlled unit. This default is based on the name of the controllable element and the file extension that is appropriate for its type. If a file with the same name exists in the directory, the Toolset appends a number to generate a unique name. See *What Elements Can Be Controlled?* on page 50 for a list of the file extensions.

A default name can also be generated for the directory used to store the child units for a model or package. This default is also based on the name of the controllable element (with no file extension). As before, if a directory with the same name exists, the Toolset appends a number to the directory to generate a unique name.

By default the Toolset prompts the user to determine whether to use the default name and, if not, determines a name for the file. Prompting occurs the first time the controllable unit is saved.

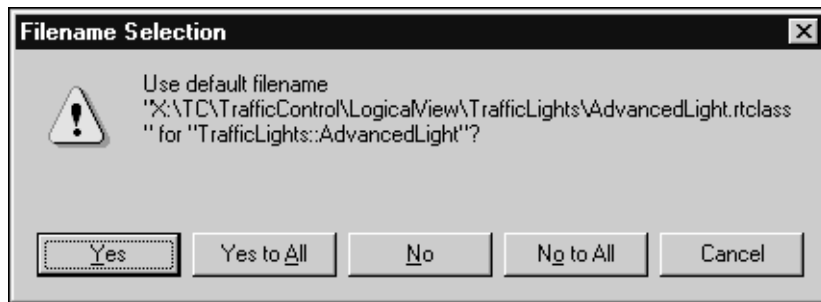


Figure 23 *Filename Selection dialog*

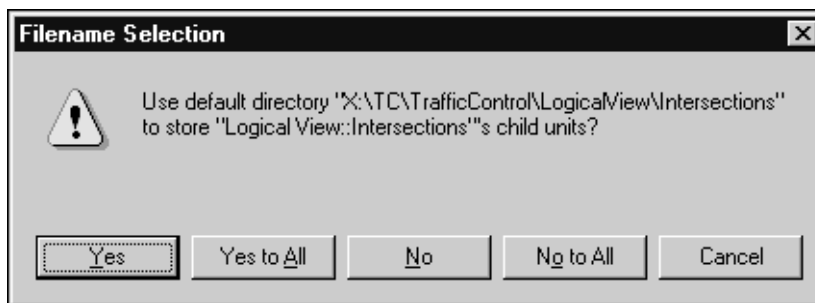


Figure 24 *Directory Name Selection dialog*

If you want to always use the generated default file names, then you can avoid these dialogs by selecting the “Always use generated file names” option in the **File** tab of the **Tools > Options** dialog.

See *Moving Controlled Units Between Model Directories* on page 67 for a description of the steps involved in moving or renaming the controllable unit file.

Controlled Units are Saved when Building

Rational Rose RealTime creates executables or libraries by generating programming language (for example, C, C++, or Java) code from a UML meta-model. It does this through the use of an external UML Model Compiler. This compiler reads a model in PetalRT form and outputs the corresponding code.

For this reason, the Toolset saves all modified units before performing a build. This can cause some issues if, for some reason, the user is unable to save a modified unit. Possible reasons for this include check-out conflicts or read-only files. The Toolset attempts to save to read-only files (after appropriate warnings and prompting) so that it can build any local changes which have been made.

Unit Information Tab

The specification dialog for a controlled element includes a **Unit Information** tab.

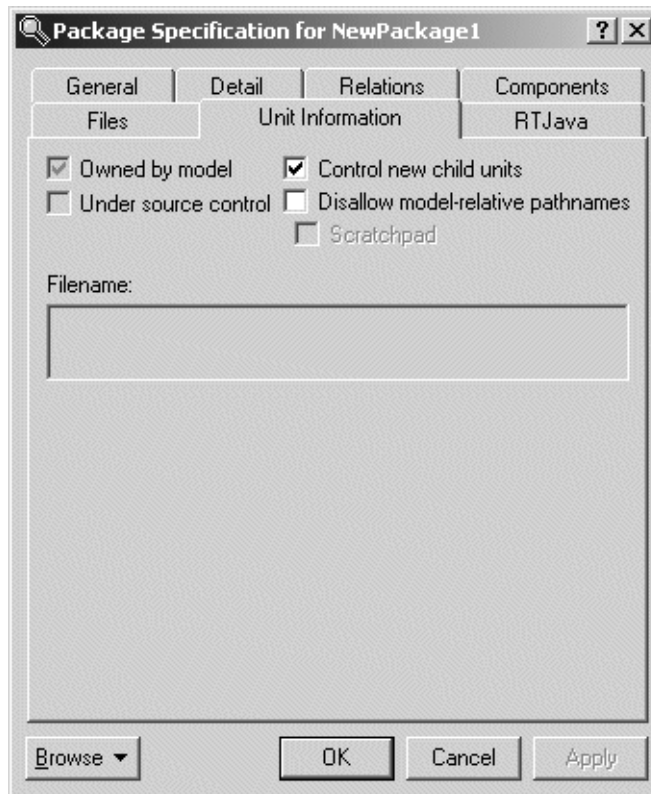


Figure 25 *Unit Information tab*

Owned by model

This check box indicates whether the unit is owned by this model, or whether it is owned by another model and shared into this model. This setting is not directly editable. See *Overview of Import, Add, and Share* on page 62 for more information on sharing.

Under source control

This check box indicates whether this element has been added to source control. This setting is not directly editable.

Control new child units

This setting controls whether newly created controllable elements in this package will be individually controlled by default. This check box is only displayed in the Unit Information tab for a package.

Disallow model-relative pathnames

This setting will inform Rational Rose RealTime to not use the implicit \$@ virtual pathmap symbol when saving units located anywhere within this package. See *Implicitly Defined Pathmap Symbols* on page 76 for details on using this setting. This check box is only displayed in the Unit Information tab for a package.

Scratchpad

This setting indicates that the package is a scratch pad. See *Scratch Pad Packages* on page 105 for a more complete description. This check box is only enabled in the Unit Information tab for a package that is not under source control.

Filename

This field displays the name of the file that is used to save this controllable unit. This field is not directly editable.

Version

This field displays the version identifier for this controlled unit. If this information is not known, then '<unknown>' displays. The ability to extract this version information depends on the source control tool being used. If a unit is not under source control, then this field will not be displayed.

What Level of Granularity Should I Use?

The primary benefit of having a fine granularity of controlled units (for example, having every controllable element as its own controlled unit) is to reduce possible contention for the file containing an element. When the model is under source control, this translates into a lower probability that a controlled unit will already be checked out when a user wants to make changes.

The granularity of controlled units determines the number of files that are used to store the model. A larger number of files may result in a degradation in the performance of the source control system. This may translate into longer times for operations such as opening a model.

The level of granularity that you use should consider the following factors:

How Stable is the Architecture?

In the early stages of analysis and design the architecture can change very frequently (and often, very drastically). During these early stages, modeling elements are created/moved/deleted often. At this point in development, it is recommended that not every possible element be controlled as individual units. Usually controlling only one or two levels of the package hierarchy will provide enough flexibility until the architecture stabilizes. Once areas of the architecture stabilize, then it is recommended that the elements in those areas be controlled down to a finer granularity before proceeding with detailed implementation.

The goal for this approach is to create an understandable directory structure for the model. When a controlled element is moved in the model, the corresponding controlled unit file is not automatically moved in the directory structure. If there is significant movement of elements in the model, the directory structure can become very fragmented resulting in situations where controlled units which are logically grouped within the model will not be physically located in the same directory hierarchy. For source control purposes it is often very useful to have controlled units for each subsystem within the same directory hierarchy since this makes it easier to perform source control operations on entire portions of the model, for example to label or search.

How Many Users Will Be Working on This Model?

If the model is only modified by a single user, and the model is not too large, then it is reasonable to store the whole model in one .rtmdl file. Otherwise, you will want to partition the model into a set of controlled units. Some teams prefer to have packages as their lowest level of granularity while others control all elements down to the class level so as to have maximum flexibility.

How Many Users Modify Elements in the Same Package?

Some teams practice strict 'class ownership' so that a single user is responsible for changes to elements in a package. In this situation the controlled elements within the package do not need to be controlled independent of the package.

How Large is Your Model?

The larger the model, the more controlled units you are likely to have. As mentioned previously, using too fine a level of granularity could cause degradation in the performance of some source control systems. Most likely, you will have to find a balance between flexibility and performance.

Implications of Changing Unit Granularity

It is possible to change the granularity of the controlled units by controlling elements that previously were not controlled (or by uncontrolling elements that previously were controlled). These actions should be taken with care since changing the granularity of a unit will move an element out of the current file and into a new file (or vice versa). If you are using a source control system to provide a history of changes for an element, then the audit trail for this element will not be easily traceable (for example, version history does not automatically include details on granularity changes).

Code Generation Performance

For the best performance from the Rational Rose RealTime code generator, every class/capsule/protocol should be stored in its own unit. This allows the code generator to parse less information when doing incremental compiles.

If multiple classes are stored in the same controllable unit, then a change to any one of these classes will cause an incremental code generation for all classes that dependent on any class in this unit, as opposed to all classes that depend on this particular changed class.

Model elements that do not influence code generation can be grouped together without influencing performance. Controllable elements that do not influence code generation are:

- Collaborations
- Class Diagrams
- Actors
- Use Cases
- Use Case Diagrams
- Deployment Packages
- Deployment Diagrams
- Processors
- Devices

Sharing Controlled Units

A large software project is typically developed by multiple teams following a layered architecture. The software produced by the "lower level" teams is used by the "higher level" teams.

Typically, an organization may have several software projects being developed at one time. Often, there is a possibility for reuse between the projects. Sometimes, this is as simple as some common 'data structure' classes. In other instances, there might be more significant framework reuse.

These sections describe the Rational Rose RealTime capabilities that support sharing and reuse in these situations.

Overview of Import, Add, and Share

Rational Rose RealTime provides three mechanisms for reuse. Each mechanism provides different capabilities and the correct mechanism to use depends on the situation.

Import a file

A collection of controllable elements may be exported to a file from one model and imported into another model. This is similar to a copy and paste of the selected elements. The imported elements are editable in both models but, since this is equivalent to creating new elements that are copies of the original elements, changes made in one model are not visible within the other model (unless the copied elements are deleted and the original elements are exported/imported again).

See *Import Controllable Elements from a File to a Model* on page 69.

Add a controlled unit

A controlled unit saved from one model may be added to another model. The elements in this unit are editable in both models and changes saved in one model are visible in the other model after the unit is reloaded. The controlled unit should be added to the same place in the hierarchy of each model.

See *Add an Existing Controlled Unit to a Model* on page 70.

Share a controlled package

A controlled package saved from one model may be shared into another model. The elements in this unit are still editable in the original model but they are not editable in the model that shares them. Changes saved for this unit are visible in the other model after the unit is reloaded. The controlled unit should be shared into the same location in the hierarchy as it is found in its owning model.

This is the same mechanism used to include the RTClasses, RTCClasses, RTComponents, and RTCComponents packages in the default model.

For models that are under source control, Rational Rose RealTime provides a setting that controls whether the source control status of shared packages should be queried when the model is opened. Since the time required to query for source control status can be significant, and is often longer than the time required to read the model files, turning off this setting can significantly improve the time required to open a large model if the model makes use of shared packages. See *Refresh shared unit status on model load* on page 85.

See *Share an Existing Controlled Unit into a Model* on page 71.

Summary of Import, Add, and Share

The following table summarizes the pros and cons of each of these mechanisms and describes some situations where each is appropriate.

Mechanism	Pros	Cons	Situations
Import	- supports unstructured sharing	- changes are difficult to propagate	- when you want to use existing elements as a basis for new elements - when propagation of changes is not required
Add	- supports structured sharing - changes are easy to propagate by reloading unit		- when an existing unit needs to be owned by a new model - supports moving an element from one model to another
Share	- supports structured sharing - changes are easy to propagate by reloading unit - enforces read-only access to shared elements - can improve opening time for a source controlled model - can generate libraries in a model, and share them into any user model - shared library contains only the required referenced elements - only have to rebuild a library interface when there are changes	- changes must be made in a model which owns the shared unit - generating multiple library interfaces from the same model may cause conflicts with guids	- useful in multiple team development where certain units should not be editable in all models

Table 2 Summary of Import, Add, and Share

Note: For additional information on issues associated with generating and sharing an external library interface, see *Generating and Sharing and External Library Interface in the C++ Reference*.

Creating Sharable Controlled Units

When a controlled unit is brought into another model, Rational Rose RealTime attempts to resolve all references contained in these elements. If an element has a reference that cannot be resolved, the problem is logged and the reference removed.

See *Model Validation* on page 163 for a description of the problems that can be encountered when references cannot be resolved.

It is best to avoid unresolved references when sharing controlled units. The simplest way to avoid unresolved references is to make sure the controlled unit is self-contained so that it does not require elements in any other controlled units. See *Check if a Subsystem is Self-contained* on page 104.

The term "external dependency" can be used to describe the relationship from an element inside a controlled unit to an element outside that controlled unit. When creating a sharable controlled unit it is important to ensure that the external dependencies are reasonable and documented.

See *Determine the External Dependencies for a Package* on page 102.

Sharing Model Properties with Controlled Units

If elements in a controlled unit use custom property sets, you must ensure that they are present in the model that will be sharing this unit. Rational Rose RealTime supports exporting the properties from the producer model and updating the properties in the consumer model. This should be done before sharing or adding the controlled unit.

See *Managing Model Properties in the Toolset Guide* for more information.

Working with Controlled Units

The following sections describe common tasks involved in defining and manipulating controlled units in a Rational Rose RealTime model:

Common Tasks:

- *Controlling a Subset of the Controllable Elements* on page 65
- *Controlling All of the Controllable Elements* on page 66
- *Changing the Granularity of Controlled Units* on page 66
- *Moving Controlled Units Between Model Directories* on page 67
- *Moving Elements Between Controlled Units* on page 67
- *Synchronizing Models with the File System* on page 68
- *Export Controllable Elements from a Model to a File* on page 68
- *Import Controllable Elements from a File to a Model* on page 69
- *Add an Existing Controlled Unit to a Model* on page 70
- *Share an Existing Controlled Unit into a Model* on page 71
- *Produce a Single Model File from a Model with Many Units* on page 73

Controlling a Subset of the Controllable Elements

There are several ways to control a subset of the controllable elements in your model as individual units. The browser context menu for a controllable element contains some or all of the following relevant menu items:

- **File > Control Unit** - the selected elements will now be controlled as individual units
- **File > Uncontrol Unit** - the selected elements will not be controlled
- **File > Control Child Units** - the child elements of the selected packages will now be controlled as individual units (this menu item is only available for packages)
- **File > Uncontrol Child Units** - the child elements of the selected packages will not be controlled (this menu item is only available for packages)

If only a small number of elements will be controlled, then it may be easiest to multi-select the elements in the browser and choose the **File > Control Unit** context menu item.

If most of the elements will be controlled, then control all the elements first, then click **File > Uncontrol Unit** on the elements that should not be controlled.

Remember:

- The controlled units are not written to disk until a save is performed.
- See *What is a Controllable Element and a Controllable Unit?* on page 48 for details on controlled unit status information available in the model browser.

Controlling All of the Controllable Elements

To control all the controllable elements in your model as individual units:

1. Select the Model in the model browser and click **File > Control Child Units**.
2. When prompted about controlling all child units recursively, click **Yes**. Also click **Yes** on the subsequent confirmation dialog.

This causes the model to be partitioned into individual files (one file for each controllable element).

Remember:

- The controlled units are not written to disk until a save is performed.
- See *What is a Controllable Element and a Controllable Unit?* on page 48 for details on controlled unit status information available in the model browser.

Changing the Granularity of Controlled Units

It is possible to change the granularity of controlled units using the **File > Control Unit** and **File > Uncontrol Unit** menu items in the browser context menu.

Changing the granularity of controlled units changes the file in which some model data is stored. If the model is under source control, then changing the granularity of the controlled unit disconnects some of the elements in the unit from their history.

Moving Controlled Units

Moving Controlled Units Between Model Directories

After a controllable element is saved as a controlled unit, moving that element in the model's package structure does *not* move the associated file in the underlying directory structure. This move causes the logical grouping of elements in the model to differ from the physical grouping of the corresponding files on disk.

If you moved a controlled unit in the package hierarchy so that the directory structure does not correspond to the model structure, then it is possible to change the location of the file by using the following steps:

1. Outside of the Rational Rose RealTime Toolset, move the file to the desired location.
2. Open the model in the Toolset.
3. When prompted about the missing model element, enter the new file location and click **OK**. You can use the Browse button to navigate to the new location.
4. After the model is open, save the package containing this model element in order to remember the new file location. If the model is under source control, check out then check in the element.

Similarly, you can rename the file associated with a controlled unit.

Moving Elements Between Controlled Units

Controllable elements can be moved from one parent package to another by dragging in the browser. If a controllable element that is not individually controlled is moved from one controlled package to another, then this will change the file in which the element is stored. If the model data is stored in a source control system, then this will disconnect that element from its history.

Synchronizing Models with the File System

It is possible to perform actions outside of Rational Rose RealTime that may cause the Toolset view of the file contents to be inaccurate. Also, in some situations you may wish to discard changes made in the Toolset in favor of the last saved versions.

To reload all the controlled units from disk, click **Tools > Synchronize Mode with File System**.

To reload a selected set of controlled units, click **File > Reload from File**.

If the model is under source control, see *Synchronizing Models with Source Control* on page 132.

Note: *Be very careful when reloading subsets of the model. Some edits to the model affect multiple units - reloading only one of the units involved in such an edit may cause undesired changes.*

Export Controllable Elements from a Model to a File

To export a collection of controllable elements (e.g., package, capsule, component, etc.) from a model:

1. Open the original model.
2. Select the elements in the browser.
3. Click **File > Export...** from the browser context menu and specify the desired file name.

This will generate an .rtpil file containing these elements.

Services Library packages

You should never export and add the Services Library shared packages. These are the packages that appear in all new models, and are prefixed with RT. For example, RTClasses, RTCClasses, RTComponents, RTCComponents.

If you export the RT packages to Rose format and then re-import, the imported RT packages are merged with the existing model and causes duplication with the default model's RT packages. This result is that the duplicate packages are renamed, which is also undesirable.

See *Produce a Single Model File from a Model with Many Units* on page 73 for details on how to export a whole model.

Import Controllable Elements from a File to a Model

Use the import mechanism to reuse existing controllable elements in another model and evolve them independently of the original versions.

These steps allow you to create a file that contains a mixture of any of the controllable element types. For example, you can export a capsule and a component into the same file. Since a Rational Rose RealTime model has restrictions on the types of controlled elements that can appear in each part of the model, when you import the elements from the file, you must place them in a valid location. In the previous example, the capsule must be placed in a logical package and the component must be placed in a component package.

If you would like the elements in the file to be imported into the appropriate top level package (for example, Logical View for capsules, Component View for components), then use the following steps:

1. Open the model.
2. Click **File > Import...** from the application menu and specify the file name in the dialog.

If there is a specific package into which you would like to import these elements, then use the following steps:

1. Open the model.
2. Select the package in the browser that should contain the elements.
3. Click **File > Import...** from the browser context menu and specify the file name.

The elements will be added to this package unless the file contains elements that are not allowed in this package type, in which case the invalid elements will be placed in the appropriate root packages. Continuing the previous example, if we tried to import a capsule and a component into a logical package, the toolset will place the capsule in the logical package and the component in the Component View package.

After the elements have been imported into the model, they will behave like any other newly created elements.

Rational Rose RealTime will not allow you to import the same elements multiple times into the same model. This would lead to unique id conflicts within the model, which may result in incorrect behavior by the Toolset. If a unique id conflict would occur as a result of the import, the following dialog will appear and the import will fail.

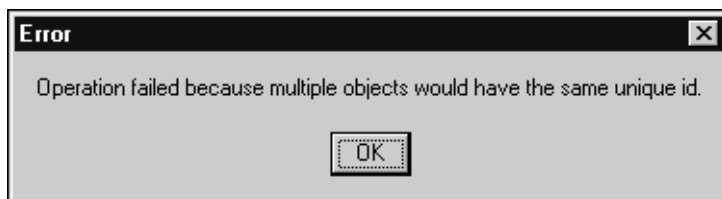


Figure 26 *Unique id conflict dialog*

Add an Existing Controlled Unit to a Model

You can add an existing controlled unit when the elements in the unit should be editable in multiple models. If the original model is abandoned, then you can also "move" this unit to another model.

To add an existing unit to a model:

1. Open the model.
2. Select the package in the browser that should contain the controlled unit.

You should add the controlled unit to the same location in this model hierarchy as in the original model.

3. Click **File > Add File...** from the browser context menu and specify the file name of the unit. You can change the filter to see the different types of files that you can add in.

The controlled unit is added to this package.

Note: *The original model must have already saved the desired controlled unit.*

After the controlled unit is added to the model, it will behave like the other units in the model.

Note: *The file contains this unit is the same as it was in the original model.*

Create a virtual path map entry to specify the location of the controlled unit added to this model. See *Defining Virtual Paths* on page 74.

If the controlled unit being added is under source control, then the model you are adding into must have source control enabled.

Share an Existing Controlled Unit into a Model

When a product involves many developers and development teams, or when multiple projects are developed, it may be very useful to make portions of a model available to other teams in a read-only format.

The shared package is indicated as "not owned" by the sharing model. See *Unit Information Tab* on page 57. This is the same facility used to include the RTClasses, RTCClasses, RTComponents, and RTCCComponents packages in the default model. The interface for the TargetRTS library is shared into a model.

With Rose RealTime, you develop your application in a high level language using state diagrams and structure diagrams. These elements are automatically converted, such as Java to RTJava, and are placed in a framework that provides critical real-time system services. The key to using the services provided by the framework is to understand how your application will integrate into the Java UML Services Library skeleton.

The ability to share packages is intended to support the development of layered models with sharing between the groups working on the different layers. For example, if a project involves a services layer with an application on top, the services layer and the application could be developed as separate models. Since the application model requires access to the services layer it could share one or more packages from the services model. In addition, sharing the elements enforces the restriction that developers working on the application layer should not modify the elements in the services layer.

It is important to properly partition the system if you wish to make use of shared packages.

Shared package producer:

The producer of the shared package must ensure that the shared package is either self contained or that all the required packages are being provided. If the package makes use of customized property sets, then the producer must also make these property sets available.

Shared package consumer:

To share an existing package into a model:

1. Open the model.
2. Select the package in the browser that should contain the shared package. For example, if you want the shared package to appear under the Logical View, select the Logical View package in the browser. Share the package to the same location in the model hierarchy as in the original model.
3. Click **File > Share External Package...** from the browser context menu and specify the appropriate file name. You can change the filter to see the different types of files that you can share.

The package is shared into this package and the containing package is marked as modified, but the shared package is not.

The original model must have previously saved the desired package as a controlled unit.

We recommend that you create a virtual path map entry to specify the location of the controlled unit added to this model. See *Defining Virtual Paths* on page 74.

Produce a Single Model File from a Model with Many Units

This can be useful for sending a model to someone at another location who does not have access to your work area.

To create a single .rtmdl file for the complete model:

1. Open the model.
2. Click **File > Export Model...** to save the model into a single .rtmdl file. You will be prompted for the file name and location.

You will be prompted about whether the file should include each of the shared packages (for example, RTClasses).

Typically, you click **No** for each package that is available to the user who will open the model, and click **Yes** for packages that are not available to that user.

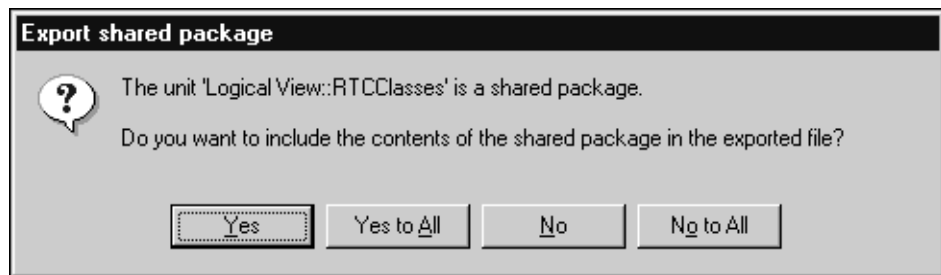


Figure 27 *Export shared package dialog*

Note: Since the resulting file contains a complete model, it must be opened (for example, click **File > Open...**) instead of being imported (for example, click **File > Import...**). Rational Rose RealTime does not allow you to import a complete model into another model.

Virtual Path Maps

When multiple users are working on the same model, there is the possibility that they may use slightly different directory paths to the model data files. Rational Rose RealTime provides virtual path maps as a general mechanism to solve this, and other similar, problems.

How Do Virtual Paths Work?

When Rational Rose RealTime saves a model element, it attempts to substitute every absolute path with a virtual path. Later, when a controlled unit is opened, each virtual path is transformed back into an absolute path.

For example, if a user has defined a virtual path:

```
$MYPATH=Z:\ordersystem
```

and saves a package as

```
Z:\ordersystem\user_services.rtlogpkg
```

the model file will refer to the package as:

```
$MYPATH\user_services.rtlogpkg.
```

When another user, who has defined \$MYPATH as:

```
$MYPATH=X:\ordersystem
```

opens the same model from their X drive, Rational Rose RealTime resolves the internal reference to the controlled unit and loads the following file:

```
X:\ordersystem\user_services.rtlogpkg.
```

Defining Virtual Paths

To define a virtual path:

1. Click **File > Edit Path Map** to open the Virtual Path Map dialog.
2. Type the name of the new virtual path in the **Symbol** field (for example, "MYPATH"), but omit the leading "\$" character.
3. In the **Actual Path** field, enter the folder location for the model file.
4. Click **Add**. You now defined a virtual path map symbol \$MYPATH.
5. To substitute the current physical paths to any existing controlled units in the model files, explicitly force a save of the units affected.

Note: Each user that is going to work on a model has to define the same path map symbols before opening the model.

The virtual paths in the dialog box are pre-defined in Rational Rose RealTime, although actual path values may be different for your system.

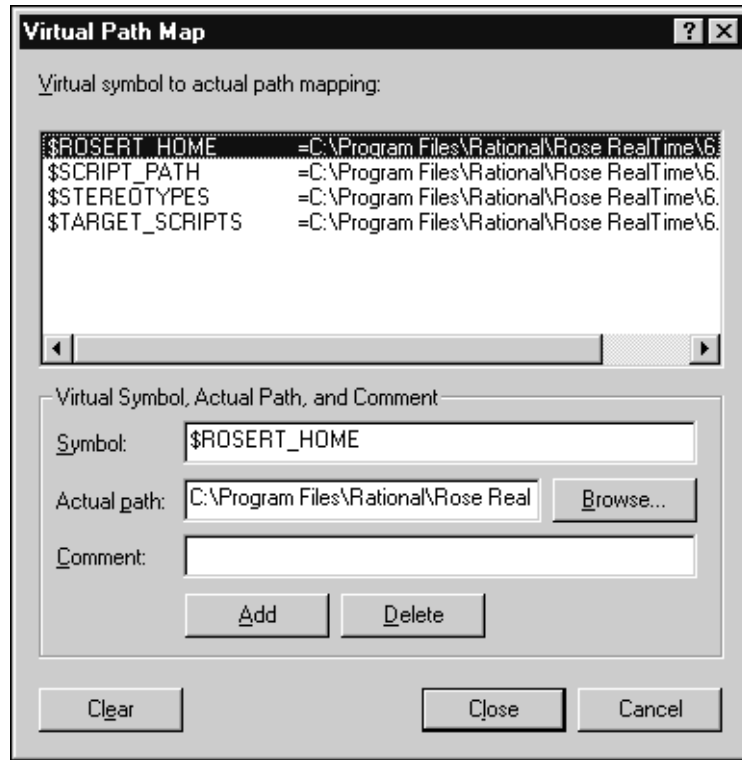


Figure 28 Virtual Path Map dialog

When anyone in the team opens or saves a model, Rational Rose RealTime attempts to match the longest possible file path to the symbols in the path map. For example, if there is a path map entry for MYPATH which is set to X:\ordersystem, and a model references the controlled unit X:\ordersystem\units\data_serv.rtllogpkg, the actual reference in the model file will be \$MYPATH\units\data_serv.rtllogpkg. Thus, when another user opens the model in their private workspace, \$MYPATH will be substituted with the path defined by that user's path map.

Defining a New Path Map Using Another Path Map Symbol

The actual path in a path map definition can contain previously defined path map symbols. For example, if there is a path map

```
$ROOT=X:\model_vob
```

you can define a path map \$MYPATH for the path

```
X:\model_vob\ordersys
```

by adding the path map

```
$MYPATH=$ROOT\ordersys.
```

Implicitly Defined Pathmap Symbols

To make simple model sharing transparent between users, two implicit pathmap symbols are defined for use by controlled unit filenames:

- \$&: the context of the unit, which is the directory of the package which contains the unit in question.
- \$@: the location of the .rtmdl file for the current model.

If you are creating a package that will be shared into other models, you should use the "Disallow model-relative pathnames" option on the package's Unit Information tab. This will ensure that filenames below the package in the unit hierarchy do not attempt to use \$@. This is important because for each model to which the shared package belongs, \$@ will have a different interpretation, leading to problems loading any files that use \$@ in their path.

Using Path Maps When Sharing Packages

When sharing and adding packages between models, it is recommended that you use virtual paths in order to avoid having explicit "hard-coded" paths saved in controlled unit files.

The controlled unit file for a controllable element is referenced by a relative file name within the directory structure of the model in which the unit was created. If another model references (for example, adds or shares) this controlled unit, the path name pointing to the shared, or added element can no longer be relative. Having these explicit paths can cause problems in team development.

This problem can be avoided by using a path map set to the root model directory of the model where the shared controlled unit was created. A model which shares this controlled unit will use the path map variable to reference the file.

You must ensure that this path map is defined when you create or open any model which shares controlled units from another model.

Using virtual paths in the value of a model property

Rational Rose RealTime does not convert actual paths in model properties to virtual paths. In order to use a virtual path in the value of a model property, you must manually enter the virtual path map symbol, including the "\$" sign - for example, \$ROOT - into the value of the model property.



Chapter 3

Source Control Fundamentals

Contents

This chapter is organized as follows:

- *Fundamentals* on page 79
- *Source Control in Rational Rose RealTime* on page 80
- *Source Control Development Concepts* on page 92
- *Versioning Strategies* on page 93

Fundamentals

Rational Rose RealTime provides source control facilities by integrating with existing source control systems to provide versioning and controlled access to model files.

Source control systems are repositories that store successive versions of files, usually with a comment attached to each version. Before a repository can begin keeping track of a file's versions, the file must be added to the repository.

Each user of a source control system typically has their own local working area that stores a copy of the files from the repository which they wish to access. Even though a repository may contain thousands of files, each user's working area need only be populated with the files from the repository that they will be accessing.

If a file is checked out to a user's working area, then it will be write-enabled; if the file is not checked out, it will be read-only. To prevent multiple users from attempting to make changes to the same file simultaneously, exclusive access is usually enforced. This is accomplished by allowing only one user at a time to check out a file version. As well, some source control systems only allow the most recent version to be checked out.

Rational Rose RealTime supports many different source control systems. For details on which systems are supported, see *Source Control Tools* on page 143.

Source Control in Rational Rose RealTime

Source Control Status

When source control is enabled, Rational Rose RealTime queries the active source control system for the status of each controlled unit. For each unit, the status indicates whether the corresponding file is present in the source control system, and if present, indicates whether the file is checked out to a specific user.

If a unit's file is checked out from source control, the browser shows a check mark next to the unit. The Unit Information tab in the specification dialog for a unit shows whether the unit is under source control. This status is also visible in the browser: units that are under source control are shown in the browser with a darkened controlled unit indicator; units not under source control are shown with a faded controlled unit indicator.

The following figure shows the different source control status options that are displayed in Rational Rose RealTime browsers:

- The lightened unit box opposite **RTClasses** and **Scratch** indicates that they are controlled units, but not currently under source control
- The checkmark in the **unit** box opposite **System** indicates that it is a controlled unit under source control that is checked out to the current user

- The empty darkened unit box opposite **TestHarnesses** indicates that it is a controlled unit under source control that is *not* checked out to the current user

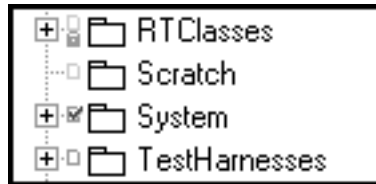


Figure 29 *Controlled Unit Icons with Source Control*

What are Primary and Secondary Edits?

Changes to a UML model sometimes require that several model elements be modified to effect the change. Some edits, such as element name changes and hierarchy manipulations, may need to modify every reference to the element being changed.

Updating all of the cross-references is not necessary to maintain the model's integrity — only direct references must be updated, such as the reference from a derived class to its superclass. Even though the model integrity is maintained in this way, code generation may not work properly unless all references are updated.

Due to the many cross-references in a model, it is often infeasible to check out all of the units that are affected by an edit so that all references can be updated immediately. To prevent excessive check-out contention for units, Rational Rose RealTime enforces that only the elements absolutely required for an edit be accessible in order to allow the edit to proceed. The changes that will affect these required units is called the "primary" edit. If these units are not accessible and cannot be checked out, then Rational Rose RealTime will not allow the edit to proceed.

All other changes, such as references that will be modified as a result of the edit, are lumped together and called "secondary" edits. Rational Rose RealTime optionally prompts the user to check out secondary edit units after the operation has completed.

It is important that secondary edits be updated as soon as possible. Otherwise model validation problems may arise.

As an example, if we have a model with the classes shown in Figure 30:

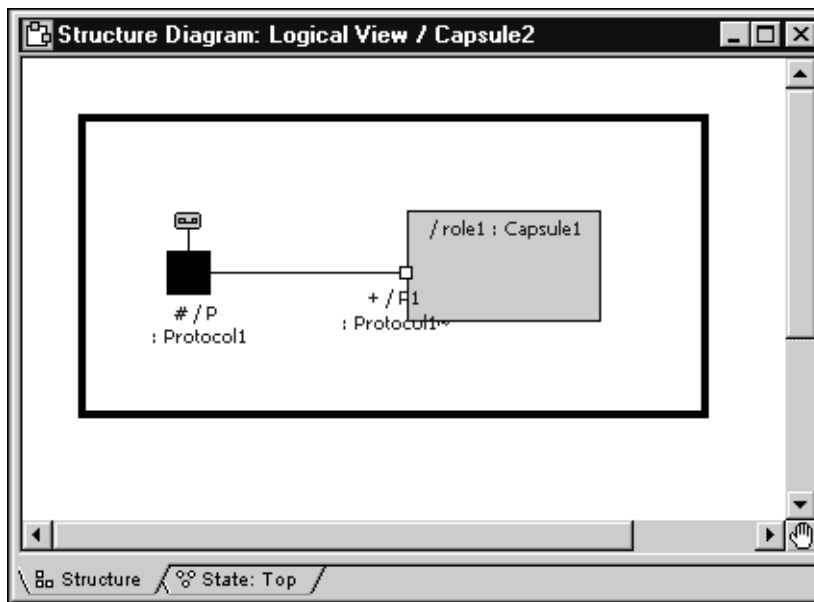


Figure 30 Model Validation Example

If a user tries to delete port P1 from capsule class Capsule1, then this would cause the port role P1 on capsule role role1 in Capsule2 to be deleted. This, in turn, would cause the connector to be deleted in Capsule2. Therefore:

- Capsule1 is a primary edit and so it must be checked out to proceed with the edit
- Capsule2 is a secondary edit and so it should be checked out but, if not, the edit can proceed

If Capsule2 is not checked out and the edited Capsule1 is checked in to source control, then users who open a model with those versions of Capsule1 and Capsule2 will encounter a model validation error that corresponds to the deletion of the connector in Capsule2. In other words, model validation has performed the secondary edit for this user. See *Model Validation* on page 163 for more information.

Source Control Settings

All source control settings are stored in the workspace file. Source control settings are located on the **Source Control** tab on the **Model specification** sheet. The **Source Control** tab can also be accessed via the **Tools > Source Control > Configure...** menu item.

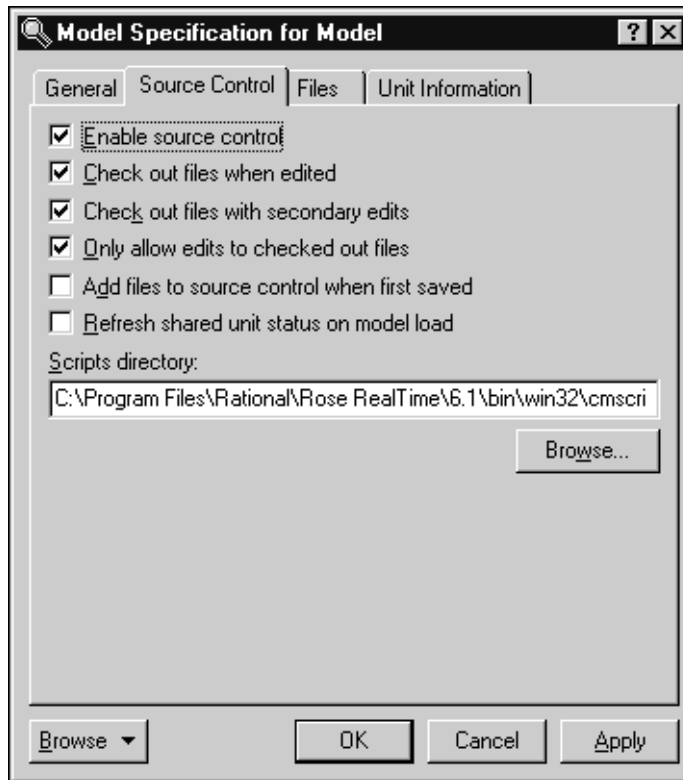


Figure 31 Source Control Settings

Enable source control

Allow model elements to be checked into and out of a source control system.

Check out files when edited

Forces the tool to automatically check out a model element if the user tries to edit it.

It is recommended that this option be selected if the model is under source control. If this option is not selected, you may have difficulty saving the changes you have made, which can also lead to problems when building.

Check out files with secondary edits

Forces the source control tool to automatically check out a model element if an edit to some other model element causes a change in the element.

It is recommended that this option be selected if the model is under source control. If this option is not selected, you may have difficulty saving the changes to the affected model elements, which can also lead to problems when building.

Only allow edits to checked out files

Prevents edits to model elements unless the element is checked out.

It is recommended that this option be selected if the model is under source control. If this option is not selected, then you may have difficulty saving your changes, which can also lead to problems when building.

Add files to source control when first saved

Causes all model elements to be placed in the source control when the model is saved.

This options is usually not selected. Instead the **Tools > Source Control > Submit All Changes to Source Control** menu item is used when submitting additions/changes.

Refresh shared unit status on model load

Indicates whether the Toolset refreshes the source control status of shared controlled units when a model is first loaded.

Clearing this option can significantly improve the time it takes to open a model with source control enabled. The status of a unit can always be refreshed later should it be required.

Scripts directory

When working with source control, Rational Rose RealTime must know the location of the scripts that interface with your source control tool.

Click **Browse** to select the directory that contains the appropriate scripts. The Browse button opens a directory browser dialog showing a subdirectory for each of the supported source control tools. The directory names corresponding to the source control systems directly supported by Rational Rose RealTime are listed below:

- cc - Rational ClearCase (Unix and Windows)
- msyss - Microsoft Visual SourceSafe (Windows only)
- rcs - Revision Control System (Unix only)
- sccs - Source Code Control System (Unix only)

Note: Source control interface scripts are located in
\$ROSERT_HOME/bin/<host platform>/cmscripts

Accessing Source Control Operations

In Rational Rose RealTime, you can access source control operations in several ways. The first is by clicking **Tools > Source Control**. These operations generally apply to all controlled units in the entire model, and include several add-in helpers and convenience operations.

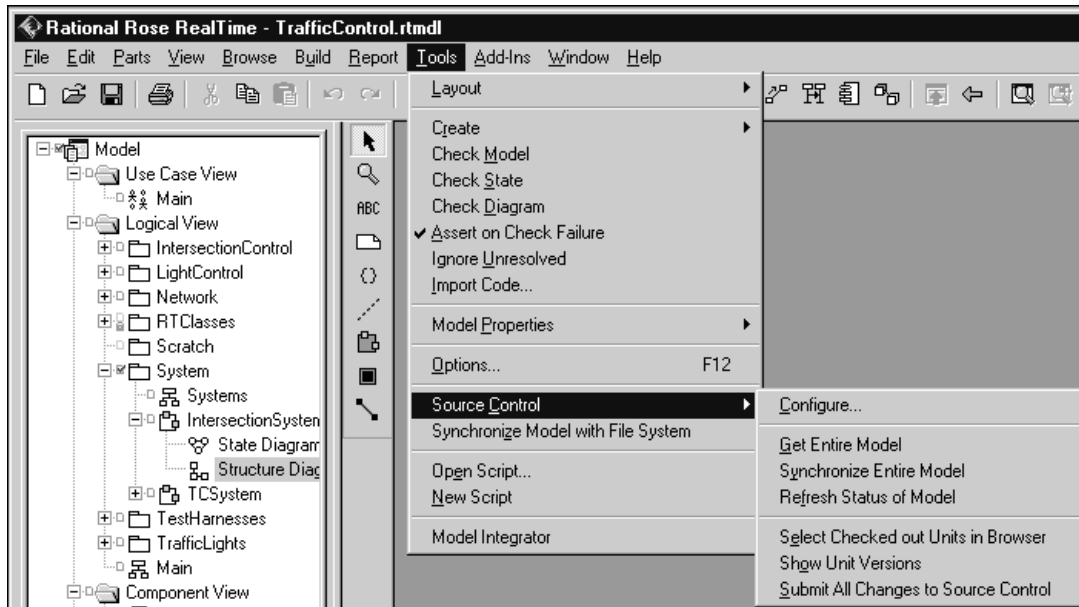


Figure 32 Tools > Source Control Menu

The second way to access source control operations is through context menus in browsers. When you select a controlled unit from the browser, the context menu contains source control operations. To apply an operation to multiple units at the same time, select all the desired units, and access the source control operation through the context menu, as shown in the following figure.

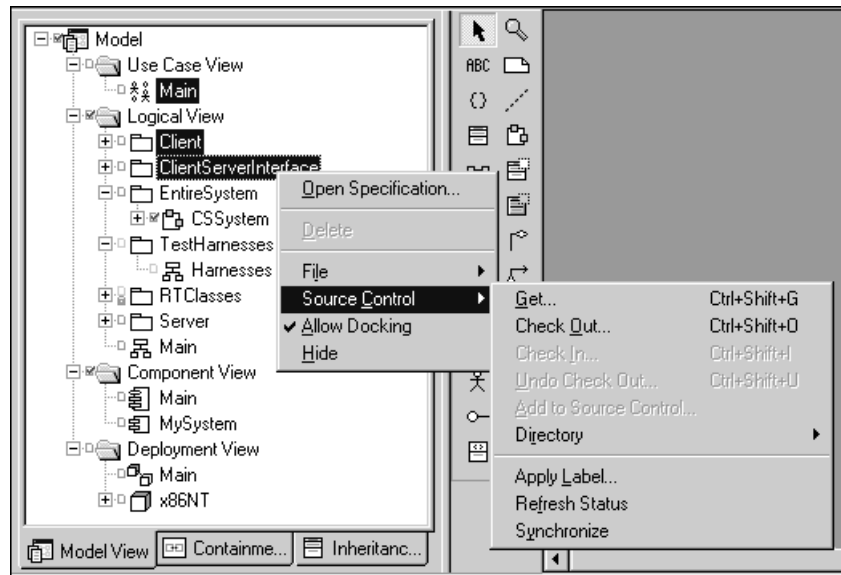


Figure 33 Source Control in the Browser context menu

Source Control Operations

The following source control operations are available from within the Rational Rose RealTime toolset with all supported source control systems. Some operations are handled slightly differently in some source control systems. See *Source Control Tools* on page 143 for more information.

Unless otherwise noted, the following operations are all enabled for any selection of units.

Refresh Status

Refresh Status queries the active source control system for each unit selected and determines whether the unit's file is under source control, and if so, whether the file is checked out. Refreshing status does not retrieve new versions of files, nor does it reload files if they have been changed outside the toolset.

Synchronize

Synchronize does the same status updating that Refresh Status performs. However, Synchronize also determines if the file on disk has changed since the file was loaded into the toolset. If the underlying file has changed, it will be reloaded into the toolset.

Get

Get interfaces with the active source control system and requests the latest version of the files corresponding to the selected units. If a new version is retrieved, Rational Rose RealTime reloads the file.

Check Out

Check out asks the source control system to lock the specified files so that the user may edit and change them, and in the future submit a new version via Check In. If the specified file is currently checked out to another user, the operation will fail. Check out will retrieve the latest version of the files being operated on.

Uncheckout

Uncheckout is available for any file that is currently checked out to the user. Uncheckout will remove the lock that the user holds on the file in the source control system and will replace their local file with the most recent file from the repository.

Add

Add attempts to place the selected units under source control. After a unit has been added to source control, it can be versioned via check out and check in. Unless a file needs to be added to source control without submitting other changes at the same time, **Submit All Changes** should be used rather than explicitly clicking **Add**.

Check In

Check In submits a checked out file to the repository so that a new version will be stored. Unless a file needs to be checked in without submitting other changes at the same time, click **Submit All Changes** to submit changes to the repository.

Submit All Changes

Submit All Changes is only available from the **Tools > Source Control** menu. This command performs the following actions:

- Determines which units are not under source control, and queries the user to add them.
- Determines which units are checked out from source control, and prompts the user to check them in.

After **Submit All Changes** successfully completes, the repository is updated with all changes made by the user.

Apply Label

Apply Label instructs the source control system to apply a specified label to the selected units. Directories may also be labelled, with the option of working recursively on the directory contents.

Some source control systems do not support labelling. See *Source Control Tools* on page 143 for more information.

Show Differences

Show Differences compares the local version of a unit with the latest version stored in the source control repository. See the Rational Rose RealTime Model Integrator documentation for details on using the merge/differencing tool.

Show Differences is only enabled when a single unit is selected.

Show History

Show History displays the version history of a unit based on the revisions of the file that are in the source control repository.

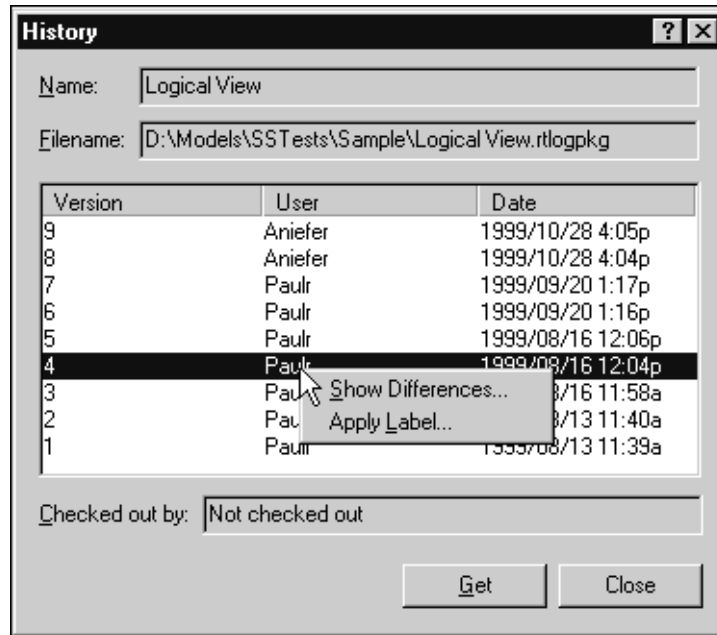


Figure 34 History dialog example

Most source control systems support the retrieval of a specific version of a file. For such systems, the **Get** button is enabled when a version is selected in the list.

To compare the local version of the unit with a specific version, right click on the version to compare with and click **Show Differences...**

For source control systems that support applying a label to arbitrary versions of elements, the context menu will also include **Apply Label...**

Show History is only enabled when a single unit is selected.

Types of Source Control Systems

There are two types of source control systems: **file** based and **view** based. Each type of system has different features and methods of supporting the source control process. Consequently there are features of each type that are not supported with the other.

File Based Source Control Systems

Source control systems in this category include Microsoft Visual SourceSafe, Rational ClearCase with snapshot views, RCS, and SCCS.

File based source control systems require each user to have a copy of the files in a local folder and use the file system's read-only attribute to control writing to files.

While working in a file-based system, you will sometimes encounter a unit in the Toolset that is marked as dirty, despite it not being checked out (see *What are Primary and Secondary Edits?* on page 81 for more information). If you are building anything that includes the dirty unit, you will need to save the changes to disk for the build to include these changes. The most desirable option is to check out the file in question.

However, if the file is already checked out by someone else, another avenue is required to make the file writable and save the changes locally. Since the file is not checked out, the read-only attribute must be changed manually. The **Make Files Writable** add-in is included with Rational Rose RealTime to make this task simpler. With this add-in enabled, the following two operations are available in the **Tools > Source Control** menu.

Make Files Writable

Performing this operation will attempt to turn off the read-only attribute on the files for units that are dirty but not checked out.

Make Files Read Only

Performing this operation changes the read-only attribute of files to match the checked out status of the corresponding unit. This means that if a unit is not checked out and the file for that unit is writable, this operation changes the file to be read only.

View Based Source Control Systems

In view based source control systems all versions of a file are stored in a versioned file system.

Users do not work with the contents of the versioned file system directly. Instead, they use a work area called a view that provides access to a set of files in the versioned file system. Moreover, a view provides access to an appropriate set of versions of those files by specifying how to choose the version of each file that will be seen in the view.

Rational ClearCase is the only currently supported view based source control system.

Source Control Development Concepts

The following concepts are helpful when designing a development process for working with Rational Rose RealTime.

Development Activity

A development activity is comprised of changes to several elements. Each activity should encompass a unit of work, such as fixing a bug or adding a new feature. When the changes for an activity are submitted to the repository, the model will evolve to a consistent new state.

Integration

Integration is the process of making changes available for use by other developers. Integration may be performed by a specific person, but it is also common for developers to play this role.

Lineup

A lineup is a collection of specific versions of files from the source control repository. Examples of lineups are:

- version 4 of every file involved in a project.
- the latest version of each file in the project that is dated before midnight, May 12.
- the version labelled “Build 6.1.112” of each file in the project

Lineups are used to represent significant combinations of files. In most development environments, the files that go into any nightly or production build form a lineup. Lineups are also valuable for reproducing specific builds of the system. The term baseline is also used to refer to a formal lineup.

Working in Isolation

It is essential that a developer's work be isolated from the work that other developers are doing. This is important for a number of reasons:

- To ensure that each developer can work without being influenced by other developers' editing, compiling, testing and debugging.
- To ensure that each developer can access the appropriate material to perform their role. This usually requires using some sort of lineup process.
- To ensure that each developer does not expose their work to the rest of the team until it is ready for integration.

To support these basic team development requirements, each developer should have their own work area. Work areas refer to private areas where developers can implement and test code, in accordance with the project's adopted standards, in relative isolation from other developers. A work area must provide private (isolated) storage for files generated during software development:

- Working (checked-out) versions of source files
- Executables
- Other work area private objects and source code, test subdirectories, and test data files

A work area private storage would be typically located within a developer's home directory on a workstation.

Versioning Strategies

Single Stream Versioning

Single stream versioning refers to having a single series of version numbers for each file. In effect, the version history for a file is a linear sequence of revisions.

While developing a project using single stream versioning, each developer always works with the most recent version of files in the repository. To edit a file, a reserved check out is performed on the latest version of the file. After changes have been made, they are submitted. This immediately makes the new version visible to other users, and will become the latest version for others to base their changes on.

This also means that only one person can work on each file at any one time since they must have the most recent version checked out in order to perform work.

Single stream versioning is not ideally suited to doing bug fixes on an existing release while doing new development for a future release.

You can use both file and view based source control systems for small projects without the need for branching or multiple stream development.

Parallel Stream Versioning

Parallel stream versioning permits each file to have a branching tree of versions. This allows many versions of the same file to be active at the same time. The following figure shows the version tree for a typical file in a parallel development project.

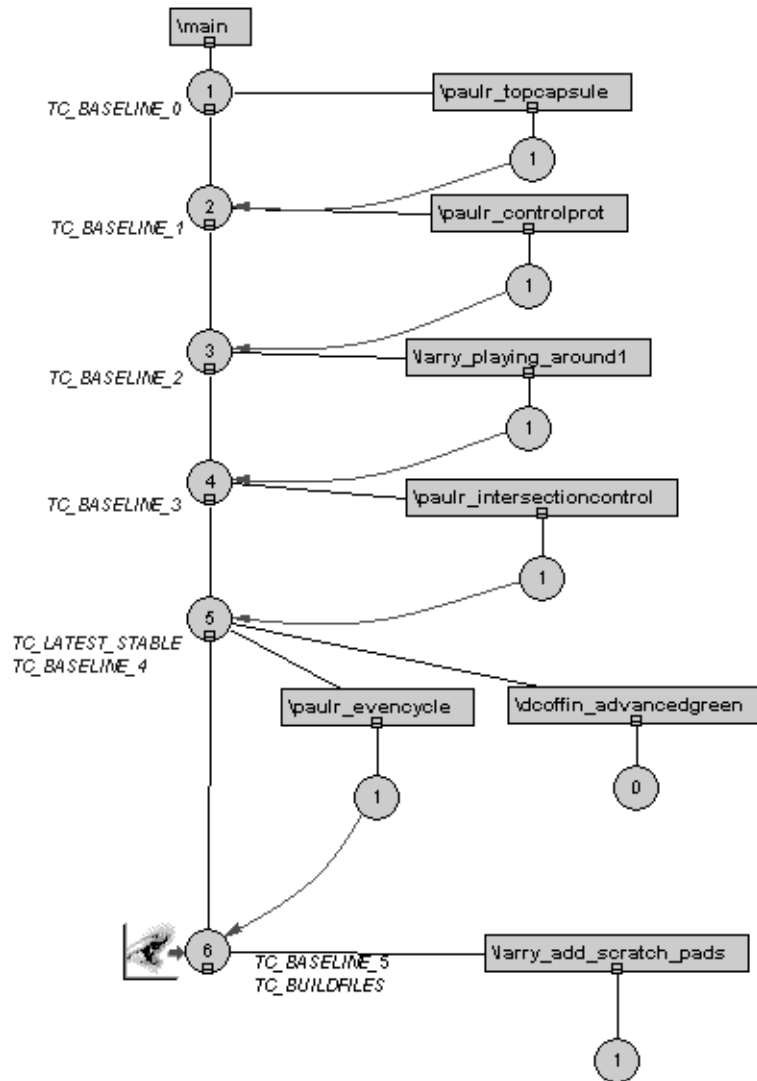


Figure 35 Example Version Tree

Most parallel development environments involve nominating a branch in the source control system as the integration branch. The integration branch is used for collecting all changes to the project (/main is the integration branch in the above diagram). Testing, release builds, and new development are all based on the contents of the integration branch.

All labelled lineups should consist of file versions from the integration branch. Once established, a labelled lineup can serve as a the basis for builds, testing, or further development. Frequently, a temporary lineup is established and built. If the build completes successfully and passes basic sanity tests, the lineup is then made available as a baseline. This process is usually automated, and should be done on a nightly/weekly basis. In the version tree above, the TC_BASELINE_<NNN> labels indicate stable baselines on the integration branch.

The lineup of file versions in the baseline is used for subsequent development. Development activities should not be performed on the integration branch, but separate from it. When a development activity is finished, the changes for that activity can be merged by an integrator back onto the integration branch. This ensures that the integration branch is strongly controlled and that only correctly working models are used to base further development on.



Chapter 4

Organizing a Model (Architect Activities)

One of the primary goals of the Architect's activities is to create an initial structure or organization of the model to facilitate team development.

Product development will often start with a small team working on one model. As development progresses, the team (and the model) will grow to a point where you should think about how to organize the model to support multiple teams working in parallel.

It is also useful to think about how sets of modeling elements can be reused by other groups. You can use Rational Rose RealTime to split parts of a model into highly cohesive layers or frameworks that can be reused in multiple models.

The actual division of a model into packages and subsystems is somewhat of an art form and we will only attempt to describe some guidelines to help you get started. Remember that once a model is well partitioned into subsystems, you can either work with one model or split the model into separate models for each subsystem.

Packages, Models, and Subsystems

Packages are used to group model elements. There are 3 kinds of packages in Rational Rose RealTime:

- Logical packages (both the Logical View and Use Case View packages are the same kind)
- Component packages
- Deployment packages.

Each kind of package can only group certain model elements. For example a logical package can group capsules and classes whereas a component package can only group component diagrams and components. Packages can also contain packages of the same kind and so it is possible to decompose your models hierarchically.

A **model** is composed of the four root packages: Use Case View, Logical View, Component View, and the Deployment View. The model is the top level model element which contains all sub-elements.

A **subsystem** is a concept and not an explicit modeling element in Rational Rose RealTime. The term subsystem represents a set of related packages that can be developed, tested, and released together.

Subsystems form the basis for reuse between models. In a layered development approach, the model for each layer will share in the subsystems for the layers beneath it.

A subsystem will typically consist of one or more logical packages and one or more component packages. The logical packages contain the classes in the subsystem and the component packages contain the components that are used to build the subsystem. Usually one of the components will be an external library in order to avoid having to compile the classes in the subsystem when it is reused in another model.

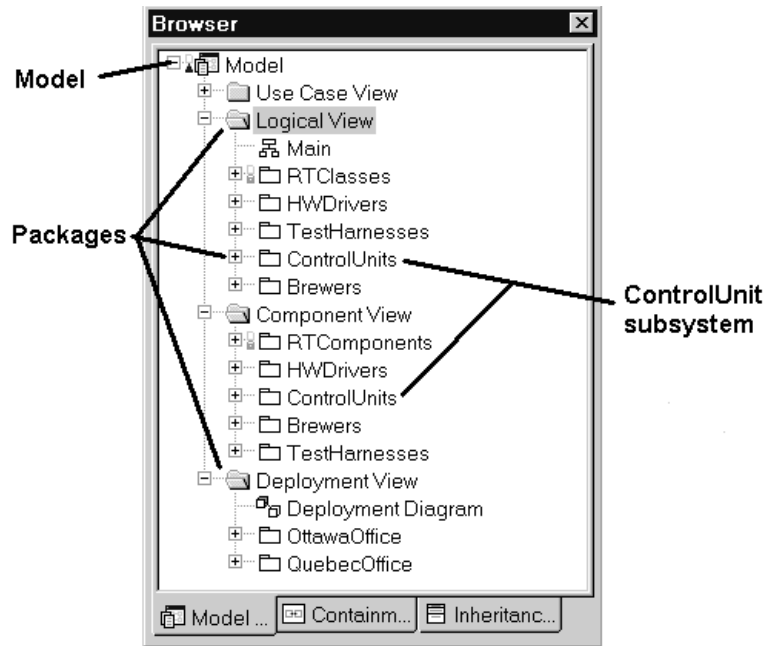


Figure 36 Model, Packages, and Subsystems

One Model versus Multiple Models

A large development project can result in a corresponding large model for the complete application. If the model has a layered architecture, then it is possible to produce a set of smaller models that follow the layering of the larger model.

One of the goals of having a separate model for each layer/subsystem is to reduce the number of developers working on the same model. This technique helps to isolate development work and reduce parallel development issues.

To build the full project, one designer, typically called the builder, opens a model referencing all the subsystems that make up the project, thus loading all the changes done to the packages in the subsystems, and build from that model.

Before splitting a model into a set of subsystem models, you should first consider the trade-offs:

Advantages of a model for each subsystem:

- Improves Toolset performance and memory footprint simply because a smaller model is opened and worked on.
- You can build, test, and release subsystems separately, reducing system complexity.
- Groups can share subsystems. Teams can share stable versions of subsystems.
- Toolset enforces ownership by not allowing developers to modify elements in shared subsystems.

Disadvantages:

- Can be more complicated to setup.
- Build process can be more involved.
- Might not be appropriate for small teams.

The following sections describe steps to perform before splitting a model to ensure that your model is well partitioned.

Getting Started

Mapping the Architecture to Subsystems

With Rational Rose RealTime, you decompose a model by grouping modeling elements into packages. You then assign a set of these packages to subsystems.

You should consider each subsystem as a distinct unit that you can build and test independently, whether the model is split or not. You must also define and enforce the interfaces between subsystems.

Decomposing a Model into Subsystems

- *Checking Package Dependencies for Completeness* on page 101
- *Check if a Subsystem is Self-contained* on page 104
- *Define Subsystem Interface* on page 104
- *Scratch Pad Packages* on page 105
- *Setup Subsystem Components* on page 107

- *Support for Unit Testing* on page 110
- *Use Property Sets for Build Settings* on page 110
- *Processors and Component Instances* on page 111

Splitting a Model

- *Should You Split a Model Before Adding to Source Control?* on page 114
 - *Splitting a Model Not in Source Control* on page 115
 - *Splitting a Model Under Source Control* on page 118

Checking Package Dependencies for Completeness

After you create packages and move the model elements into the packages (subsystems), you want to ensure that the subsystems you created have dependencies that you expect. If the interdependencies between subsystems are too complex, it will be difficult to work in teams (changes will not be isolated) and split the model.

Show Access Violations

Click **Report > Show Access Violations...** to verify that the designed dependencies between packages (subsystems) are correct and complete. For a description of this menu item, see the Report Menu in the Toolset Guide.

The Architect should revisit the package dependencies periodically to check that the detailed implementation has not violated the intended architecture.

Click **Report > Show Access Violations...** to verify that there are no violations in the logical packages and component packages in the subsystem. You should also verify that every class and logical package referenced by the components in the subsystem are also part of the subsystem.

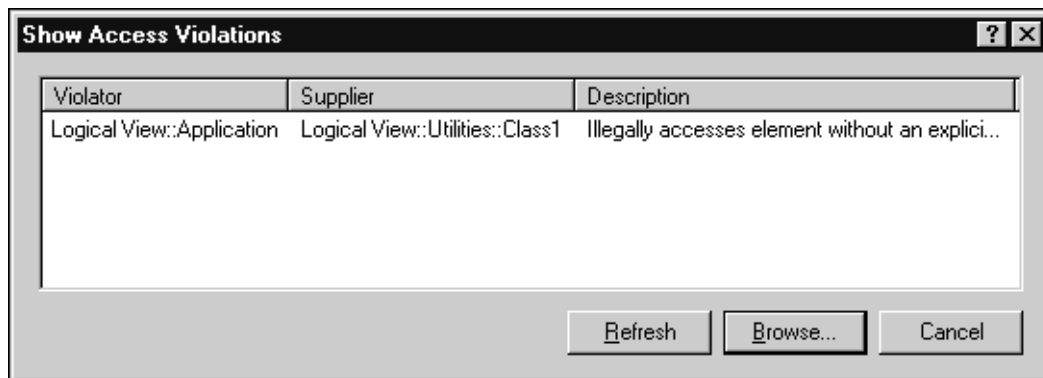


Figure 37 Show Access Violations dialog

Determine the External Dependencies for a Package

The Specification dialog for a package contains a Relations tab which shows the dependencies for this package. This is a quick way to see if a package has any dependencies but it can be difficult to visualize the dependencies if you just look at this list. In order to properly visualize the package relationships, use a class diagram.

To quickly create a class diagram showing the relationships for a specific package:

1. Open the class diagram.
2. If this package is not already on this diagram, then drag it from the browser onto the diagram.
3. Select the package in the diagram and click **Query > Expand Selected Elements**.

The resulting dialog allows you to add related elements to this diagram based on the chosen options.

4. To see the direct dependencies for this package, set the options to expand one level of suppliers. Ensure that dependency relations are chosen in the Relations dialog.
5. Click OK to add the related packages to the diagram.

The following figure shows the package dependencies in a simple traffic light control model.

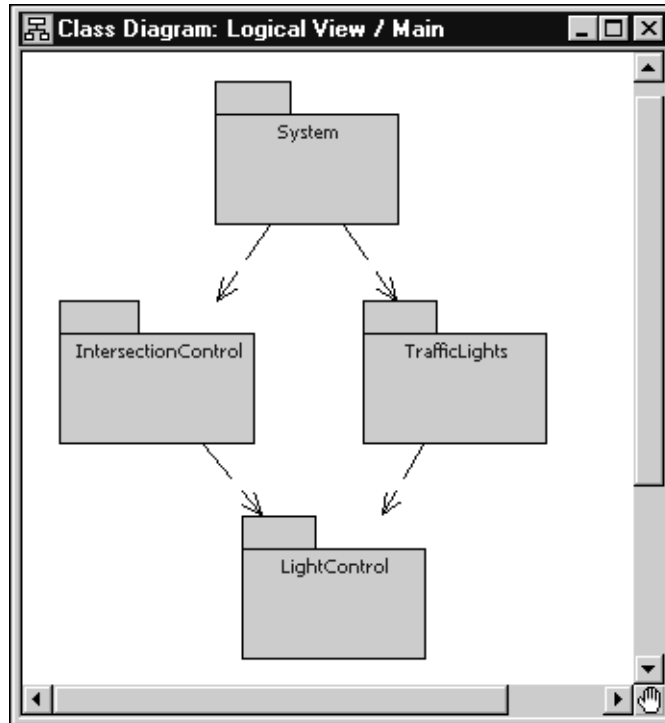


Figure 38 *Package Dependencies Diagram Example*

These steps are also supported for component packages on a component diagram.

By varying the options you set in these dialogs, you can quickly produce a diagram showing the desired information. If many packages were added to the diagram, then you can use the automatic layout mechanism to produce an initial layout for the diagram.

By reviewing the relationships in this diagram, the Architect can detect any undesirable dependencies. Resolving an undesirable dependency can involve either modifying the class(es) that caused the violation and/or moving some of these classes to another package.

Check if a Subsystem is Self-contained

A self-contained subsystem is composed of packages that do not have any dependencies to packages outside of the subsystem. A self-contained subsystem can be shared without requiring any other subsystems.

Assuming the package dependencies are complete (see *Checking Package Dependencies for Completeness* on page 101), then checking whether a subsystem is self-contained involves examining the dependencies for the packages in the subsystem to ensure that all of them are linked to other packages within the subsystem.

A subsystem does not need to be self-contained in order to be shared, provided that the sharing model contains all the other subsystems that are required.

Define Subsystem Interface

By reducing the coupling between subsystems, you can lessen the chance of having integration problems caused by using subsystems that have complex dependencies into one another.

It is important for the producer of the subsystem to pay close attention to the classes in a subsystem that are public (for example, is visible and usable outside of the subsystem) and which are private. For ease of use, it is also recommended that the subsystem contain a set of class diagrams that illustrate the public classes.

Best Practices

1. Specify the visibility of each class (public or implementation).
2. Include one or more class diagrams showing the public classes. You may also use different visual clues for the public classes in a class diagram, for example, color.

Scratch Pad Packages

When working on a model in a team environment, it is common for a developer to create temporary model elements that are not intended to be shared with the rest of the team. For example, a developer may create a temporary component when unit testing a change to a capsule class. If the model is under source control, the developer may not want these temporary elements checked in to source control with the other changes they are making.

To support temporary work within a controlled model, Rational Rose RealTime supports *scratch pad packages*. A scratch pad package is a package that is not added to source control. Also, changes can be made to a scratch pad package without the Toolset requiring that package to be checked out. This allows multiple team members to make temporary changes within their own local scratch pad package without encountering any contention issues.

Elements can be moved into or out of a scratch pad package by dragging them to another package in the browser. Elements can also be copied into (or out of) a scratch pad package using control-drag in a model browser.

The controllable elements within a scratch pad package cannot be individually controlled. If a controlled unit is moved into a scratch pad package, then it will no longer be controlled.

To create a scratch pad package:

1. Create a package and give it a descriptive name, e.g., **TemporaryComponents**.
2. Select the package in the browser and click **File > Control Unit** menu item. If this menu item is not enabled, then ensure that the parent element for this package is also controlled.
3. Open the **Specification** dialog box for this package and change to the Unit Information tab.
4. Select the **Scratchpad** option and click **OK**.
5. Save the package containing the scratch pad. Optionally, you may also save the scratch pad. If the containing package is under source control, then it should be checked out and checked in.

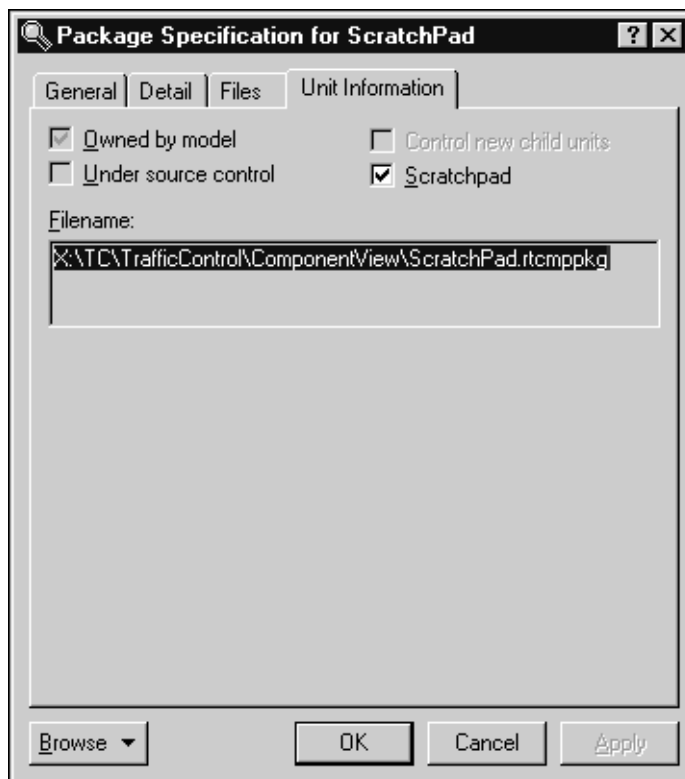


Figure 39 Scratch Pad Package Unit Information Tab

Remember

Scratch pad packages are only intended to be used for temporary work. If you initially create an element in a scratch pad package and you decide that it should be placed under source control, then it is possible to move the elements from a scratch pad package to another package by dragging them in the browser.

Conversely, if you initially create an element in a (non-scratch pad) package and you decide that it should not be placed under source control, then it is possible to move an element from another package into a scratch pad package by dragging in the browser.

When you open a model that contains a scratch pad package, the Rational Rose RealTime Toolset will try to read its file based on the file information for this package. If the file does not exist, then the Toolset will prompt to allow you to specify an alternative file location. If you do not have a local file for this scratch pad, then you may click Cancel to this dialog with no repercussions. If you wish to avoid any prompts about missing scratch pad packages, open the **Tools > Options** dialog box and select **Ignore missing scratch pad files** on the **File** tab.

Potential Problems

Since a scratch pad package is never placed under source control, you must ensure that the elements within it are not referenced by elements that are checked in to source control. For example, if you create a capsule class in a scratch pad package, this capsule class should not be referenced within a component that is checked in to source control.

Note: *Elements in a scratch pad package can reference elements either inside or outside of that package with no problems.*

If you accidentally check in an element that references an element in a scratch pad, then other developers will encounter model validation errors when they load that version of the referencing element. For more information on model validation, see *Model Validation* on page 163.

Setup Subsystem Components

Background

Rational Rose RealTime supports three general types of components:

1. *executables* - building an executable component results in an executable (for example, .exe files).
2. *libraries* - building a library component results in a static library (for example, a .lib file).
3. *external libraries* - an external library specifies the path to an existing static library so it does not need to be built.

It is also possible to create dependencies from an executable component to a library or external library component. The dependency indicates that the static library associated with the library component should be linked in with the executable created when building the executable component. See the language-specific guides for more information about components and component dependencies.

A small model may have a single executable component that is built to produce the application. A large model will have an executable component and many library components, typically corresponding to the layering in the architecture.

In addition to the components that are used to build the complete application, it is often useful to have components that build subsets of the model, for example, for unit testing purposes.

Components in Subsystems

Ideally, each subsystem will contain one or more external library components. These components are built as part of the build process of a subsystem and are referenced in models that use the subsystem. An external library component will allow the sharing model to reuse the prebuilt library, which can dramatically reduce build times for a large model.

A subsystem will often include multiple variations of each component. For example, a debug component and a release component. For ease of navigation and organization, the subsystem should group the components into packages, for example, a Debug package and a Release package containing the debug and the release components respectively.

The subsystem model will need one or more executable components that are used to test the subsystem. Typically, the executable component will only contain the testing classes and it will have a dependency on the library component for the subsystem.

The following component diagram shows three components for an example subsystem. The **BaseRelease** component is a library that contains the subsystem. The **SanityTests** and **FullRegressionTests** components are executables that use the **BaseRelease** component.

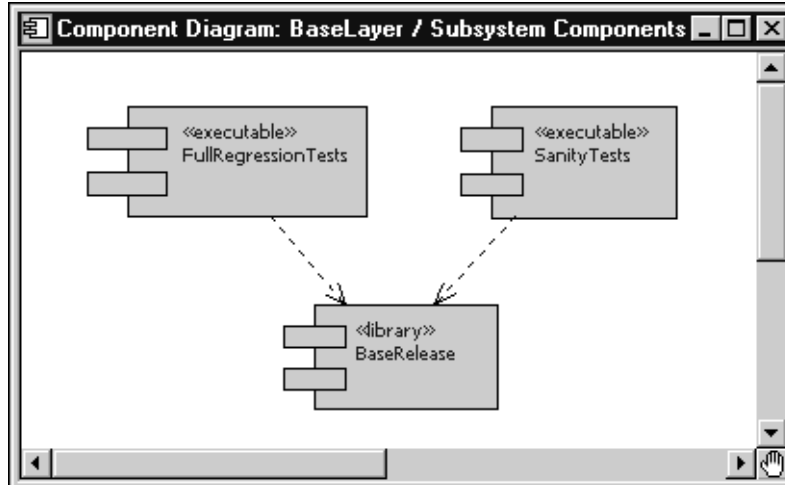


Figure 40 Example Subsystem Components

After you create the necessary components and the dependencies between them, you have to determine which classes belong to which components. Typically, this will follow naturally from the architecture of the model, but there can be some issues that arise during development. As new classes are created, they will need to be added to the appropriate component(s). If multiple developers create classes referenced by the same component, then the component can become a source of contention.

The contention for a component can sometimes be avoided, or at least reduced, if the component references logical packages instead of classes themselves. Remember that referencing a package from a component is equivalent to referencing all the classes in that package. The added benefit is that the component does not need to be updated when a new class is added to the package provided that class belongs in that component. The risk is that a component may contain classes that it does not require.

Support for Unit Testing

While working within a subsystem model, a developer may find it useful to create a component for use in unit testing their changes. If this component has lasting value, then it should be created as part of the subsystem model so that it can be reused. To support the organized storage of unit testing components, the Architect may find it useful to create component packages that can be used for grouping these components.

If this component is temporary, then it can be created within a scratch pad package. Often a temporary component is created by copying an existing component by control-dragging it in the browser.

It is recommended that the Architect creates one or more scratch pad packages in the Logical View, Component View, and Deployment View in order to support unit testing with temporary components. It may also be useful to create a scratch pad package in the Use Case View.

If many developers are creating components in the same (non-scratch pad) package, then this package can become a source of contention. If your development process requires the creation and source control of unit testing components, then you may wish to create several component packages that are used for this.

For more information on the tasks involved in developer testing, see *Unit Testing within a Subsystem* on page 130.

Use Property Sets for Build Settings

Using property sets for common build settings is a suggested method of maintaining and reusing project level configuration information for building components. See *Managing Model Properties in the Toolset Guide*.

Tasks:

- The builder or architect defines custom sets of component properties which are specific to a project. For example you can have debug and release build settings. Custom properties are stored in the .rtmdl file for this model.

- A component should be based on the appropriate properties sets by modifying the **Default set** field in appropriate property tabs of the component Specification dialog. Any local overrides should also be added.
- For each executable component, the top level capsule must be set.
- When the **loadbuilder** updates a property set then all components that use this set must be updated by opening the Specification dialogs and clicking **Apply Defaults** on the corresponding property tab. The development team should also be notified so that they can update their private test components in the same way.

Processors and Component Instances

Project Level Processors

For each project, there is usually a known set of processors that component instances are intended to execute on. Since all the subsystems in the model are intended to execute on this set of processors, these project level processors should be defined in a deployment package that is shared between the various subsystem models.

The builder should setup a deployment package containing these project level processors. For example the builder could configure processors for the labs that are available for the development teams. These deployment package(s) can then be shared in each subsystem model. Each package should be owned by one of the models so that modifications can be made to it in a controlled manner.

The processors in these project level deployment packages will typically not contain any component instances. If they did contain a component instance, then sharing them would also require the corresponding component packages which contain the required components. In turn, these components would require the referenced classes and logical packages. Unless these elements are present in all subsystem models, these processors should only be used as 'templates' in the subsystem models.

Subsystem Level Processors

A development team may choose to create additional processors for their own use, either by copying the project level processors or by creating new processors for platforms that are not shared with other teams.

The subsystem level processors can contain component instances based on the components present in the subsystem. Typically this would include component instances for regression testing the subsystem and for unit testing major classes in the subsystem.

Component Instances

Component instances provide the ability to run a specified executable component on a specified processor. A component instance is controlled with the processor. As mentioned previously, project level processors will usually not have any component instances and so they will typically be copied before they can be used to execute/test a component.

Subsystem level processors will typically contain component instances that execute/test the entire subsystem. Developers working on the subsystem can use these component instances, but they may find it easier to create specific unit testing components and corresponding component instances.

If the model is under source control, then scratch pad packages provide a way to create and execute temporary component instances.

Tasks

- A set of deployment packages can be created to hold processors that are available i- house for testing. The processors will contain **ip** addresses, host names, and other configuration information that can be re-used by all developers.
- Subsystem processors can be created by copying project level processors and creating the component instances desired for executing/testing the subsystem.
- A developer copies one of the pre-defined processors into a scratch pad package, and then creates the desired component instance to run on the processor.

Preparing and Releasing Subsystems

In a model composed of multiple subsystems, there should be policies in place which describe how new versions of the subsystems will be made available to the other models.

Subsystem Supplier

When a team is ready to release a new version of a subsystem, they must ensure that the correct version of all the necessary elements of the subsystem are available. This includes:

- logical packages containing the classes in the subsystem
- component packages containing the library components and/or external library components for the subsystem
- any other required Rational Rose RealTime elements
- any other required external (non-Rational Rose RealTime) elements including .lib files for external library components

The team which is releasing the subsystem will typically prepare the required elements using one of the following mechanisms:

1. Label Subsystem Elements

If the model is under source control, then a label can be applied to the elements in the subsystem.

2. Copy Subsystem Elements

The elements in the subsystem can be copied to a known location.

Subsystem Consumer

The architect for a model which requires this subsystem must then ensure that their model includes the new version of the subsystem. The mechanism for this depends on how the subsystem elements were made available.

If the subsystem elements were copied to a known location, the architect must ensure that this location is referenced by the model. If the location is the same as the previous version of the subsystem, then no changes should be necessary. If the location has changed, then the architect may have to recreate their model by sharing in the shared packages from the new locations and adding in the packages that are owned by this model.

If the subsystem was packaged using a source control label, then the architect must ensure that this label is used for getting the new lineup for their model.

If there are changes to the subsystem interface, then the architect of a model which uses this subsystem must ensure that the corresponding changes are made within their model.

Splitting a Model into Subsystem Models

Splitting a large model into smaller subsystem models can improve team development. A developer can now work on the appropriate model for their particular subsystem. Working on this smaller model should reduce the Toolset footprint and improve the performance of several operations (e.g., opening a model).

It is possible to split a model **before** or **after** it has been placed under source control. If a model has not been controlled, it is recommended to split the model first, then add the resulting controlled units to source control.

Before a model is split into subsystem models, you must ensure that the dependencies between the subsystems will support this partitioning. Specifically you must ensure that the subsystems form a layered architecture that will allow each subsystem to exist in a model that does not contain any of the 'higher level' subsystems. See *Checking Package Dependencies for Completeness* on page 101.

Should You Split a Model Before Adding to Source Control?

If your model is not already in source control then it is best to split the model before adding it to source control. If your model is already in source control, then it is still possible to split it into separate models but the process is a bit different.

See *Splitting a Model Not in Source Control* on page 115 or *Splitting a Model Under Source Control* on page 118 for the full description.

Splitting a Model Not in Source Control

At this point we assume that you have a base model (in this example we will call it **Base**) and that the model is not yet in source control. We also assume that you will be creating separate models for each of your subsystems.

Lastly, this description also assumes that you will want to keep the controlled units for each subsystem model together and so they will be moved into the subsystem directory tree. Moving the files is optional but it can make it much easier to manage the files that make up each model.

The section *Overview of Import, Add, and Share* on page 62 provides valuable background information that should be understood before proceeding with this task.

Tasks

1. Ensure that the base model has defined the initial controlled units, at least at the package level corresponding to the subsystem partitioning.

The base model (**Base**) directory hierarchy for the sample model would look something like:

```
Base.rtm1
<Base>
  UseCaseView.rtllogpkg
  <UseCaseView>
  LogicalView.rtllogpkg
  <LogicalView>
    SubSystem1.rtllogpkg
    <SubSystem1>
    SubSystem2.rtllogpkg
    <SubSystem2>
  ComponentView.rtcmpkg
  <ComponentView>
    SubSystem1.rtcmpkg
    <SubSystem1>
    SubSystem2.rtcmpkg
    <SubSystem2>
  DeploymentView.rtdploy
  <DeploymentView>
```

2. Click **File > Edit Path Map** to create a Virtual Path Map variable for each top level package in the model (for example, each subsystem package). In our example, we could create path map variables SubSystem1LogicalPkg, SubSystem1ComponentPkg, SubSystem2LogicalPackage, SubSystem2ComponentPkg, etc.
3. Explicitly save the **Base** model units affected by the new **pathmap** variable.
4. If the **Base** model makes use of custom property sets, then these must be made available to the subsystem models. Click **Tools > Model Properties > Export...** to create a file that can be imported to the subsystem models.
5. Create a new model by clicking **File > New**. This model will be used for the first subsystem. Ensure that the path map variables are still defined correctly.
6. If the Base model makes use of custom property sets, then ensure that these are available in the subsystem model. Click **Tools > Model Properties > Replace...** to import the file containing the property sets.
7. Control all the elements in the new model by right-clicking on the Model in the browser and clicking **File > Control Child Units**.
8. Save the model (.rtmdl) into an appropriate directory by clicking **File > Save As...** We suggest that you create a dedicated directory for each subsystem.

For example, we could name the subsystem model **SubSystem1** and store it in a directory called **SubSystem1**. Answer yes to all the prompts about file names for the control units.

9. Next, you can optionally move the packages for your subsystem from the base model directory hierarchy into the subsystem model directory hierarchy created when you saved the new model.

For each package that will be part of the subsystem, move the package controlled units (in our example this would be SubSystem1.rtlogpkg and SubSystem.rtcmppkg into the corresponding directory level in the new model) and then move the directories for each package to the corresponding location.

The resulting directory hierarchy for the new model should look something like:

```
SubSystem1.rtmdl
<SubSystem1>
  UseCaseView.rtlogpkg
  <UseCaseView>
  LogicalView.rtlogpkg
  <LogicalView>
    SubSystem1.rtlogpkg
    <SubSystem1>
  ComponentView.rtcmppkg
  <ComponentView>
    SubSystem1.rtcmppkg
    <SubSystem1>
  DeploymentView.rtdeploy
  <DeploymentView>
```

If you move the files, then edit the associated path map variables to reflect the new file locations.

10. Next you will have to add the subsystem packages into the subsystem model by clicking **File > Add Files...** in the context menu for a package. These packages should be added in at the same location in the subsystem model hierarchy as they were in the base model. In our example, SubSystem1.rtlogpkg should be added to the Logical View and SubSystem1.rtcmppkg should be added to the Component View.
11. Save the subsystem model.

Steps 5 - 11 should be repeated for each remaining subsystem with the following addition.

- Before adding the subsystem packages to the new subsystem model (for example, step 8 above), you must share in the packages from the other subsystems that are required by this subsystem. In our example, assume that SubSystem2 in the Base model depends on SubSystem1. In the SubSystem2 model we must first click **File > Share External Package...** in the browser context menu to share SubSystem1.rtlogpkg and SubSystem1.rtcmppkg into the Logical View and Component View respectively.

If we attempt to add the packages for SubSystem2 before the other required packages are present in the model, then the Rational Rose RealTime Toolset will prompt to determine the location of the required elements. If you encounter this prompt, click Cancel on this dialog and the subsequent dialog, and then share the required packages as described above before trying to add the SubSystem2 packages again.

After splitting the original model, you will typically not use that model for any further development. You may choose to create an equivalent model that shares in all the subsystems. For example, in our example we could create a new model called **NewBase** which shares in the packages in SubSystem1 and SubSystem2. This model cannot be used to edit any of the subsystems but it might be useful for building and/or testing.

Note: *If the original model is not controlled, see Controlling All of the Controllable Elements on page 66 and Controlling a Subset of the Controllable Elements on page 65.*

Splitting a Model Under Source Control

At this point we assume that you have a base model (in this example we will call it **Base**) and that the model is under source control. We also assume that you will be creating separate models for each of your subsystems.

Lastly, this description also assumes that you will want to keep the controlled units for each subsystem model together and so they will be moved into the subsystem directory tree. Moving the files is optional but it can make it much easier to manage the files that make up each model.

The section *Overview of Import, Add, and Share* on page 62 provides valuable background information that should be understood before proceeding with this task.

Tasks

1. Ensure that the base model has defined the initial controlled units, at least at the package level corresponding to the subsystem partitioning.

The base model (**Base**) directory hierarchy for the sample model would look something like:

```
Base.rtmdl
<Base>
  UseCaseView.rtlogpkg
  <UseCaseView>
  LogicalView.rtlogpkg
  <LogicalView>
    SubSystem1.rtlogpkg
    <SubSystem1>
    SubSystem2.rtlogpkg
    <SubSystem2>
  ComponentView.rtcmppkg
  <ComponentView>
    SubSystem1.rtcmppkg
    <SubSystem1>
    SubSystem2.rtcmppkg
    <SubSystem2>
  DeploymentView.rtdeploy
  <DeploymentView>
```

2. Click **File -> Edit Path Map** to create a Virtual Path Map variable for each top level package in the model (for example, each subsystem package). In our example, we could create path map variables SubSystem1LogicalPkg, SubSystem1ComponentPkg, SubSystem2LogicalPackage, SubSystem2ComponentPkg, and so on.
3. Check out the root packages in the **Base** model.
4. Explicitly save the **Base** model units affected by the new **pathmap** variable.
5. Check in the root packages in the **Base** model in order to save the modified file path information under source control.
6. If the **Base** model makes use of custom property sets, then these must be made available to the subsystem models. Click **Tools > Model Properties > Export...** menu item to create a file that can be imported to the subsystem models.
7. Create a new model by clicking **File > New**. This model will be used for the first subsystem. Enable source control for this model by opening its Specification dialog, switching to the Source Control tab, and specifying the desired settings. Ensure that the path map variables are still defined correctly.

8. If the Base model makes use of custom property sets, then ensure that these are available in the subsystem model. Click **Tools > Model Properties > Replace...** to import the file containing the property sets.
9. Control all the elements in the new model by right-clicking on the Model in the browser and clicking **File > Control Child Units**.
10. Save the model (.rtmdl) in the appropriate local working directory for your source control system by clicking **File > Save As...** (for example, /vob/SubSystem1). We suggest that you create a dedicated directory for each subsystem.

For example, we could name the subsystem model **SubSystem1** and store it in a directory called **SubSystem1**. Answer yes to all the prompts about file names for the control units.

If you choose, you may add the subsystem model to source control at this stage. Click **Tools > Source Control > Submit All Changes to Source Control** to ensure that all the controllable units are added.

11. Next you can optionally move the packages that make up your subsystem from the base model directory hierarchy into the subsystem model directory hierarchy that was created when you saved the new model.

The actual steps involved in moving the files and directories within source control are dependent on the source control tool.

For each package that will be part of the subsystem, move the package controlled units, in our example this would be SubSystem1.rtlogpkg and SubSystem.rtcmppkg into the corresponding directory level in the new model, and then move the directories for each package to the corresponding location. The resulting directory hierarchy for the new model should look something like:

```
SubSystem1.rtmdl
<SubSystem1>
  UseCaseView.rtlogpkg
  <Use Case View>
  LogicalView.rtlogpkg
  <Logical View>
    SubSystem1.rtlogpkg
    <SubSystem1>
  ComponentView.rtcmppkg
  <Component View>
    SubSystem1.rtcmppkg
    <SubSystem1>
  DeploymentView.rtdeploy
  <Deployment View>
```


If you move the files, edit the associated path map variables to reflect the new file locations.

12. Add the subsystem packages into the subsystem model by clicking **File > Add Files...** in the context menu for a package. These packages should be added in at the same location in the subsystem model hierarchy as they were in the base model. In our example, SubSystem1.rtllogpkg should be added to the Logical View and SubSystem1.rtcmppkg should be added to the Component View.

If you added the subsystem model to source control previously, then you will be prompted to check out the root packages that are affected. Click **OK** for these dialog boxes.

13. Save the subsystem model.
14. Enter the changes for this subsystem model into source control by clicking **Tools > Source Control > Submit All Changes to Source Control**.
15. We recommend that you create a default workspace for each subsystem model. See *Make Default Workspace Available to Project Members* on page 141 for more information on this task.

Steps 7- 15 should be repeated for each remaining subsystem with the following addition.

- Before adding the subsystem packages to the new subsystem model (for example, step 12 above), you must share the packages from the other subsystems that are required by this subsystem.

In our example, assume that SubSystem2 in the Base model depends on SubSystem1. In the SubSystem2 model we must first click **File > Share External Package...** menu item share in the browser context menu to share SubSystem1.rtllogpkg and SubSystem1.rtcmppkg into the Logical View and Component View respectively.

If we attempt to add the packages for SubSystem2 before the other required packages are present in the model, then the Rational Rose RealTime Toolset prompts you to determine the location of the required elements. If you encounter this prompt, click **Cancel** and on subsequent dialog boxes as well, and then share the required packages as described previously before attempting to add the SubSystem2 packages again.

After splitting the original model, you will typically not use that model for any further development. You may choose to create an equivalent model that shares in all the subsystems. For example, in our example we could create a new model called **NewBase** which shares in the packages in SubSystem1 and SubSystem2. This model cannot be used to edit any of the subsystems, but it might be useful for building and/or testing.

Note: If the original model is not controlled yet., see *Controlling All of the Controllable Elements on page 66* and *Controlling a Subset of the Controllable Elements on page 65*.



Chapter 5

Working with a Model Under Source Control (Developer Tasks)

As a developer, you work with a subsystem model under source control. Before reading the following sections, you should be familiar with the material in *Source Control Fundamentals* on page 79.

These are the tasks a developer will need to become familiar with:

- *Setting up your Source Control Tool* on page 123
- *Configuring Work Areas* on page 124
- *Getting a Specific Lineup of a Model* on page 124
- *Opening a Model Under Source Control* on page 125
- *Adding a new Controlled Unit into Source Control* on page 125
- *Checking Controlled Units In and Out of Source Control* on page 126
- *Building and Running Locally* on page 129
- *Unit Testing within a Subsystem* on page 130
- *Promoting Changes for Integration* on page 132

Setting up your Source Control Tool

Before using Rational Rose RealTime with your source control tool, you must perform any tool-specific configuration, as specified in the sections referenced below:

- *ClearCase Workstation Setup* on page 150
- *SourceSafe Workstation Setup* on page 153
- *RCS/SCCS Workstation Setup* on page 157

Other Source Control Tools

If you customized Rational Rose RealTime to work with another source control tool, ensure that the source control tool is correctly installed on each developer workstation.

Configuring Work Areas

Before working on a source controlled model you first have to get a specific lineup of controlled units onto your local disk. From there, you can start working on a model by opening the workspace file.

Your Source Control Administrator or Integrator will know how to determine the specific label or configuration used to create a local work area. Next, it is a matter of setting up a local work area before running Rational Rose RealTime.

See the following tool specific sections:

- *ClearCase Work Area Setup* on page 152
- *SourceSafe Work Area Setup* on page 155
- *RCS/SCCS Work Area Setup* on page 158

Getting a Specific Lineup of a Model

When a Developer begins a development task, they must start with the correct version of the model files. The steps involved vary depending on your team development process and the underlying source control tool.

For Rational ClearCase, the developer should be using a **config** spec that defines their view to include the correct versions of the model elements.

For Microsoft Visual SourceSafe, your team may be using labels to mark the correct versions and the developer should perform a Get based on that label by using the **Label** field available from the **Parameters...** button in the **Get** dialog box.

Similar labelling strategies can be used with RCS/SCCS.

Opening a Model Under Source Control

Opening a model under source control is no different than opening a non-source controlled model. In either case, opening the associated workspace (.rtwks) file is the recommended way to load the model into the Toolset. A default workspace will typically be made available by the Source Control Administrator, see *Make Default Workspace Available to Project Members* on page 141.

Note: *When opening the model from source control, open the associated workspace file. The workspace stores the source control configuration settings for the model. If you open the model directly (without using the workspace), the source control settings will not be set, and you will have to go to the Model Specification dialog and set them. See *Configure the Workspace Source Control Options* on page 141.*

Adding a new Controlled Unit into Source Control

After your model is under source control, any new controlled units you create in the model must be added to source control.

Check Out Parent Package

When a new controlled unit is added to a source controlled model, you will have to check out the package in which the new unit will be placed. If there is excessive contention for parent packages, then you may wish to partition the package into several smaller packages.

To add a new controlled unit to source control:

1. Add a new unit to your model.
2. If the parent package is not checked out the Toolset prompts you to check it out.
3. Click **Tools > Source Control > Submit All Changes To Source Control**.

Clicking **Source Control > Add to Source Control...** can also add selected units to source control.

Note: *Until a unit is saved to disk, it cannot be added to source control.*

The advantage of clicking **Tools > Source Control > Submit All Changes To Source Control** is that you will not forget to add any units.

Checking Controlled Units In and Out of Source Control

Checking Out Controlled Units

After a model is under source control, check out elements before you edit them. Depending on the source control settings, the Toolset may force you to check out before editing. See *What are Primary and Secondary Edits?* on page 81.

To check out an element for editing:

1. Select the appropriate controlled unit(s) in the browser.
2. Click **Source Control > Check Out...** from the browser context menu
A confirmation dialog appears. After accepting, the check out operation will proceed on all selected elements.

While editing these elements you may affect other elements that are not checked out. See *What are Primary and Secondary Edits?* on page 81.

Checking In Controlled Units

To check in a controlled unit after editing:

1. Select the unit(s) in the browser.
2. Click **Source Control > Check In...** menu item from the browser context menu.

The unit will be automatically saved to its file before the file is checked in.

If you make changes to multiple units, and/or you have several new units to add to source control, it is recommended that you use the **Submit All Changes to Source Control** menu item (described below).

Submitting All Changes to Source Control

If you have several controlled units to check in or add to source control, then it can be error prone to select them and use the **Check In...** or **Add to Source Control...** menu items. Forgetting to add new units can result in model validation errors when other users get the new version of the other units. See *Model Validation* on page 163.

To avoid these potential problems, you should add or check in all checked out and new units in your model at once by clicking **Tools > Source Control > Submit All Changes to Source Control**.

You are prompted to add any new units to source control (see Figure 41), then asked to check in any checked out units (see Figure 42). These dialogs will list all new and checked out units respectively.

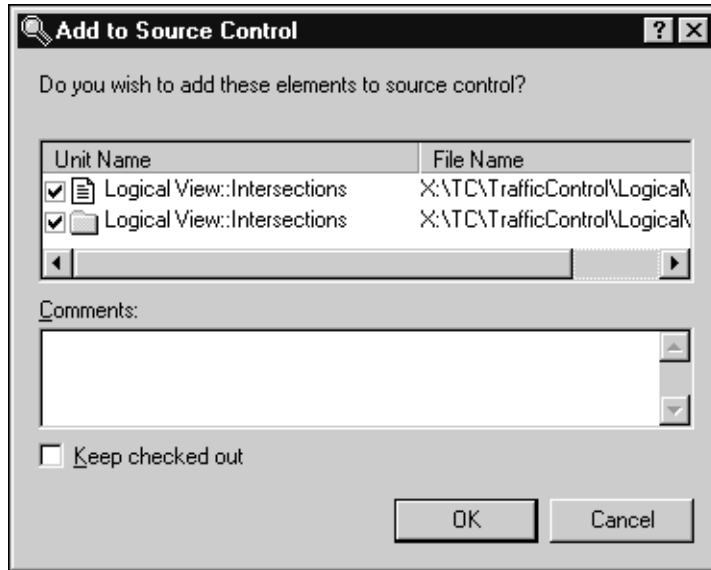


Figure 41 Add to Source Control dialog

The **Add to Source Control** dialog box has a **Keep checked out** option that automatically checks these units out after they have been added to source control.

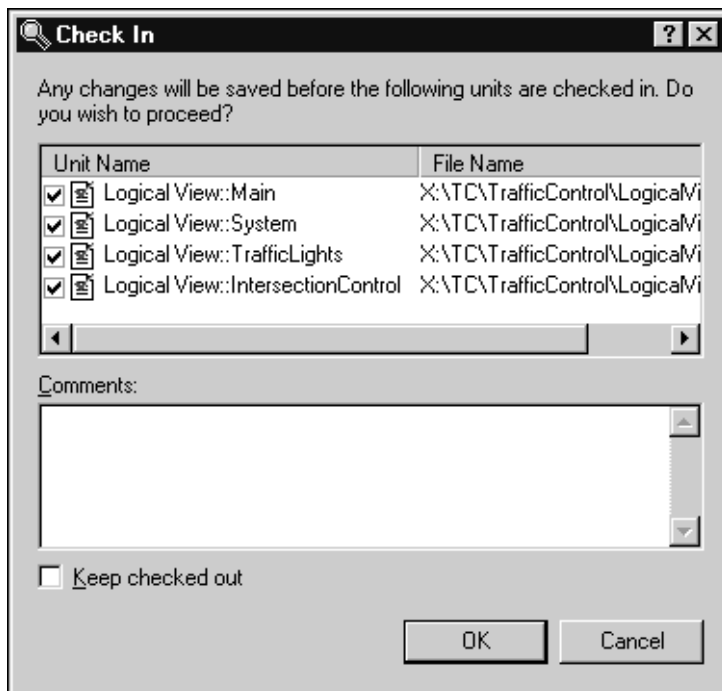


Figure 42 Check In Dialog

By default, all new and checked out units are submitted. You can use the check boxes on the left side of each unit to filter items from the list in each dialog box. The check in dialog has a **Keep checked out** option to keep these units checked out after the new version has been checked in.

Undoing a Check Out

After you check out a controlled unit, you may choose to undo the check out and not submit a new version.

To undo a check out for an element:

1. Select the appropriate controlled unit(s) in the browser.
2. Click **Source Control > Undo Check Out...** from the browser context menu. The following dialog box appears.

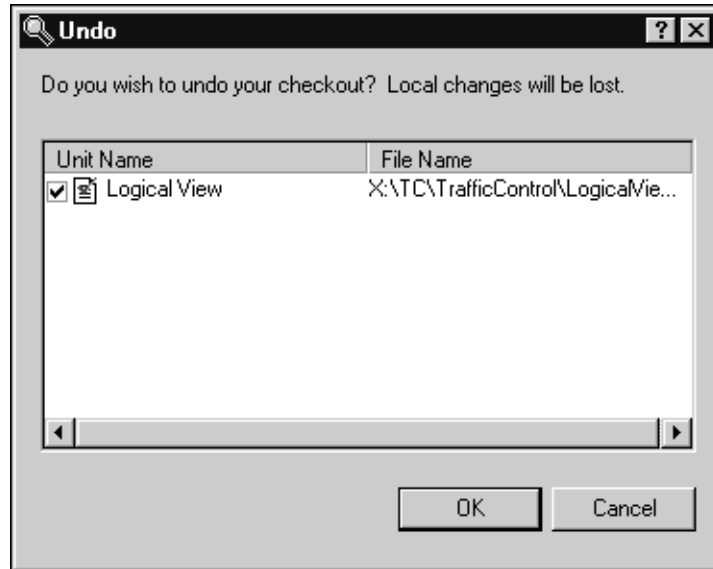


Figure 43 Undo Check Out Dialog

Note: When you undo a check out, you will lose your local changes for these units and they will not be submitted to source control.

Building and Running Locally

You can build any component that exists in a source controlled model without having to check out any files. If the component is an executable and already has an associated component instance, you can also run and observe the component instance without having to check out anything from the model.

However, if you want to create your own components, for example to change the top capsule or modify certain build settings, you will have to create a local component, processor, and component instance.

Reusing Build Settings

Typically, your development team will have a set of build properties that are used to for your components. Whenever you are creating a new component, you should try to use one of these property sets.

For a description of the steps involved in creating temporary components for testing purposes, see *Set up Private Components* on page 130.

Probes and Inject Messages

When running a component instance locally, all probes and inject messages added to the component instance are saved locally in the `<model>.rto` file. This allows a user to debug and run a component instance without having to check it out of source control.

See User-specific Working Environments (.rtus and .rtwks) in the *Toolset Guide* for more information on the .rto file.

Unit Testing within a Subsystem

It is possible to unit test a capsule by building and running it. You may need to create a new component that has this capsule defined to be its top capsule, and you may also need to create a new component instance to run. If there are existing unit test components for this capsule, they can be reused directly.

Best Practices

It is recommended that your Architect setup a package for saving useful test harnesses that would be of interest to developers working on the subsystem. *Support for Unit Testing* on page 110

If you are creating your own unit testing components, see *Use Property Sets for Build Settings* on page 110.

Set up Private Components

Each developer will want to create their own components during day to day development, for example, to unit test the changes they are working on. Often these private components are not meant to be released or added to source control. Through the use of scratch pad packages, Rational Rose RealTime provides each developer the option of creating local modeling elements which are not checked into source control.

Tasks

- The developer creates any required testing classes in a scratch pad package.
- The developer creates a component in a scratch pad package. If the testing component is only a variation of an existing component, this can be done by copying (for example, control-dragging) the existing component into the scratch pad.
- The developer can set the build properties for the component by applying a property set. This will ensure that the build settings default to the desired values, typically defined by a project level build settings property set.
- The developer adds/modifies references to the required classes in the component and sets the top level capsule.
- The developer can now build the component.
- The developer should copy (for example, control-drag) the desired processor into a scratch pad package and drag the test component onto this processor to create a component instance.
- The developer can now run this component instance and test their changes.

Differencing and Merging Model Elements

The local version of a unit may be compared to its previous versions that may exist in your source control tool. Click **Source Control > Show Differences...** to compare the local file with the most recent version under source control. To compare with an earlier version, see *Show History* on page 90.

Similarly, if a unit is checked out, then a Get performed on that unit will prompt the user if a merge should be performed. To merge from the most recent version under source control, perform a Get on the desired checked out unit. To merge from a previous version, use the Get facilities provided in the **Show History** dialog box.

See the Rational Rose RealTime Model Integrator documentation for a complete description of how to use the merge/differencing tool.

Synchronizing Models with Source Control

To synchronize the status of units displayed in the model browser with the status as reported by source control, click **Tools > Source Control > Refresh Status of Model**.

To synchronize and reload any elements that are different from what is loaded in the Toolset, click **Source Control > Synchronize**.

Note: *This is different from Synchronize with File System, which ignores source control information.*

To extract the latest version of all files from source control, click **Tools > Source Control > Get Latest Version of Model**.

All of the above actions can be performed on subsets of the model using context menus in the browser.

See *Source Control Operations* on page 87 for more information.

Promoting Changes for Integration

When working in a single stream development process, there is no explicit integration step. Instead, submitting changes to the source control repository effectively integrates them with the existing file versions.

For an example of integration with a parallel stream development process, see *ClearCase Parallel Development: Sample Process* on page 171.



Chapter 6

Building and Integrating (Integrator Tasks)

The Integrator combines changes from multiple developers to produce a lineup to use as a basis for the next set of development activities. The Integrator will typically be responsible for the automated building process.

Some of the specific tasks involved in building an integrating are:

- *Building using Automated Scripts* on page 133
- *Building within a Larger Build Procedure* on page 135
- *Reuse of Build Artifacts* on page 136
- *Integrating Changes* on page 137
- *Automating Model Validation* on page 137

Building using Automated Scripts

The Rational Rose RealTime code generator assumes it is generating for a valid lineup of classes and packages. See *Automating Model Validation* on page 137 for an example of how you can validate the model as part of the automated build process.

Starting with a valid model, it is possible to initiate a build from a clean directory using the following two steps. These are effectively the same steps used by the Rational Rose RealTime Toolset.

Note: The "\ " character in the following command syntax represents the command line continuation character. This may be different on your system.

To initiate a build from a clean directory:

1. Build the **makefiles**:

```
 ${CodeGenMakeCommand} ${CodeGenMakeArguments} \  
   -f $ROSERT_HOME/codegen/bootstrap/${CodeGenMakeType}.mk \  
   "RTS_HOME=${TargetServicesLibrary}" \  
   "MODEL=${ModelFile}" "COMPONENT=${QualifiedName}" \  
 RTmakefiles
```

where **CodeGenMakeCommand**, **CodeGenMakeArguments**, **CodeGenMakeType**, and **TargetServicesLibrary** are replaced by the corresponding value in the component; **QualifiedName** is replaced by the fully qualified name for the component; and **ModelFile** is replaced by the file name for the model (.rtmdl) file.

2. Generate the code and compile using:

```
 ${CodeGenMakeCommand} ${CodeGenMakeArguments} \  
   -f Makefile RTcompile
```

For example, if the following substitutions are made:

Argument	Example Value
<code> \${CodeGenMakeCommand} </code>	<code> clearmake </code>
<code> \${CodeGenMakeArguments} </code>	<code> -k -J4 </code>
<code> \${CodeGenMakeType} </code>	<code> ClearCase_clearmake </code>
<code> \${TargetServicesLibrary} </code>	<code> \$ROSERT_HOME/C++/TargetRTS </code>
<code> \${ModelFile} </code>	<code> /my/path/MyModel.rtmdl </code>
<code> \${QualifiedName} </code>	<code> Component View::MyComponent </code>

The resulting commands are:

```
 clearmake -k -J4 \  
   -f $ROSERT_HOME/codegen/bootstrap/ClearCase_clearmake.mk \  
   "RTS_HOME=$ROSERT_HOME/C++/TargetRTS" \  
   "MODEL=/my/path/MyModel.rtmdl" \  
   "COMPONENT=Component View::MyComponent" \  
 RTmakefiles
```

```
 clearmake -k -J4 -f Makefile RTcompile
```

Note: Automated builds are not restricted to `clearmake`.

Virtual Path Map Symbols

If you wish to build a component outside of the Toolset, all virtual path map symbols used in the model must have corresponding environment variables defined.

Building within a Larger Build Procedure

For integration into a larger build procedure, automated builds can generate the code and compile the code in two separate steps. This involves a slight change to the steps listed above.

To integrate into a large build procedure:

1. Build the **Makefiles** using the same command as above.
2. Generate the code (without compiling it) by replacing “**RTcompile**” above with “**RTgenerate**”:

```
 ${CodeGenMakeCommand} ${CodeGenMakeArguments} \
  -f Makefile RTgenerate
```

3. Compilation of the generated code (without regenerating it) uses “**RTmycompile**”:

```
 ${CodeGenMakeCommand} ${CodeGenMakeArguments} \
  -f Makefile RTmycompile
```

Note: The “\” character in the command syntax represents the command line continuation character. This may be different on your system.

If we use the same example substitutions as above, then the resulting commands are:

```
clearmake -k -J4 \
  -f $ROSERT_HOME/codegen/bootstrap/ClearCase_clearmake.mk \
  "RTS_HOME=$ROSERT_HOME/C++/TargetRTS" \
  "MODEL=/my/path/MyModel.rtmdl" \
  "COMPONENT=Component View::MyComponent" \
  RTmakefiles
```

```
clearmake -k -J4 -f Makefile RTgenerate
```

```
clearmake -k -J4 -f Makefile RTmycompile
```

Reuse of Build Artifacts

Build artifact reuse is supported in Rational ClearCase environments only by using the ClearCase "wink-in" feature. Both "clearmake" (Unix, Windows NT, and Windows 200) and "omake" (Windows NT and Windows 2000 only) provide the wink-in mechanism.

Creating Reusable Build Artifacts

In order for build artifacts to be "wink-in-able", the following criteria must be met:

- The component's **OutputDirectory** must be in a view.
- All controlled units within the model must be version controlled in a ClearCase VOB.
- All controlled units must not be checked out to the view performing the build.
- The build must be performed from a clean directory. If a build is unsuccessful, the **OutputDirectory** must be completely cleaned in order to guarantee wink-in.
- In the component, the **CodeGenMakeType** and **CompilationMakeType** properties must both be set to either "ClearCase_clearmake" or "ClearCase_omake" as appropriate. Similarly, the **CodeGenMakeCommand** and **CompilationMakeCommand** properties must be set to something appropriate, typically either "clearmake" or "omake".

The **OutputDirectory** can be a view-private directory, but that requires every developer to create that directory in their view first. A recommended practice is to use a directory element that is stored in a VOB.

The following are encouraged practices:

- All external include files should be version-controlled in a ClearCase VOB.
- The **TargetServicesLibrary** should be version-controlled in a ClearCase VOB.
- Other linked libraries should be version-controlled in a ClearCase VOB.
- Optionally, \$ROSERT_HOME should be version-controlled in a ClearCase VOB.

Using Build Artifacts

A developer wishing to reuse the artifacts from a build should:

- assign his or her environment variables (such as \$ROSERT_HOME and \$PATH) appropriately,
- use the same versions of elements that the build used,
- create in his or her view, if it does not already exist, the same **OutputDirectory** used by the builder
- perform the same activity that the builder performed (a compile or a generate, from within the Toolset or from the command-line).

See *ClearCase Parallel Development: Sample Process* on page 171 for a description of a development process that provides significant build artifact reuse.

Integrating Changes

Integrating developer changes is highly dependent on the development process being used. The primary goal of the Integrator is to produce an updated lineup of model elements that can be used as a basis for subsequent development activities. This will often involve merging changes from multiple developers (using the Rational Rose RealTime Model Integrator) and performing local builds to verify sanity.

For an example of how integration can be performed in a parallel development environment with ClearCase, see *ClearCase Parallel Development: Sample Process* on page 171.

Automating Model Validation

Rational Rose RealTime provides an automated way of determining if a model is valid. These steps can be incorporated into an automated build process to determine if the code generation and compilation steps of the build should be performed.

Using the Rational Rose RealTime Extensibility Interface (RRTEI), you can write a script that:

- Opens a specified model (using the `Application.OpenModel` method).
- Saves the log to a specified file (using the `Application.SaveLogAs` method).
- Closes the Toolset (using the `Application.Exit` method).

For more information on the RRTEI, see the Rational Rose RealTime Extensibility Interface References Online Help.

You can invoke this script as part of an automated build. The automated build script can then search (for example, **grep**) the log file to determine if any errors or warnings were encountered when the model opens. If problems were encountered, then the build script can email the log file to the builder. If no problems were encountered, then the build script can continue with the code generation and compilation steps.



Chapter 7

Source Control Administration

The source control administrator provides the overall source control infrastructure and environment for the development team. It is assumed that the source control administrator is familiar with both Rational Rose RealTime and your source control tool.

Prior to any team development work with Rational Rose RealTime, the following tasks must be completed:

- *Set up a Source Control System and Repository* on page 140

For each project, the following tasks will be required:

- *Control Appropriate Model Elements as Units* on page 140
- *Create a Local Work Area* on page 140
- *Save Model to Local Work Area* on page 141
- *Configure the Workspace Source Control Options* on page 141
- *Add the Model to Source Control* on page 141
- *Make Default Workspace Available to Project Members* on page 141

After these steps are completed, development can start on the project. However, there are additional responsibilities to consider:

- *Defining Developer Work Areas* on page 142
- *Creation of Labels and Lineups* on page 142
- *Manipulation of the Source Control Repository* on page 142

The details of many of these tasks are dependent on the source control plan developed for the project.

Set up a Source Control System and Repository

Prior to placing Rational Rose RealTime models under source control, there are some setup steps that must be followed to configure the source control system to allow proper integration with Rational Rose RealTime. Most of these tasks are performed outside of Rational Rose RealTime and require knowledge of the source control tools you will be using. If you are unsure about the procedures, please see your source control tools documentation.

Before continuing, please review the tool-specific documentation in the sections referenced below:

- *Rational ClearCase* on page 144
- *Microsoft Visual SourceSafe* on page 152
- *RCS and SCCS* on page 155

After reviewing this material, ensure that a repository is properly set up for integration with Rational Rose RealTime.

Control Appropriate Model Elements as Units

Determine the granularity you require for your project and team environment at the current stage in development. Do this in collaboration with the architect(s) for the project. See *What Level of Granularity Should I Use?* on page 59 for information on choosing the right granularity.

See *Controlling All of the Controllable Elements* on page 66 and *Controlling a Subset of the Controllable Elements* on page 65 for a description of the mechanics involved in specifying which model elements should be controlled.

Create a Local Work Area

You will want to establish a local work area for you to save models. Setting up a work area is specific to each source control tool:

- *ClearCase Work Area Setup* on page 152
- *SourceSafe Work Area Setup* on page 155
- *RCS/SCCS Work Area Setup* on page 158

Save Model to Local Work Area

Before placing the model under source control, it must be saved to the local work area. Save the model to the directory you have associated with your source control repository.

Configure the Workspace Source Control Options

To enable source control, fill in the proper settings on the Source Control tab, as described in *Source Control Settings* on page 83.

Add the Model to Source Control

The simplest way to add all applicable units to source control is to use the **Submit All Changes to Source Control** tool as described in *Submit All Changes* on page 89. If you require finer control over which units will be added, see *Source Control Operations* on page 87 and *Add* on page 88.

Make Default Workspace Available to Project Members

The workspace (.rtwks) file contains information that is common to all users that will be working on the project. Settings in the workspace will rarely, if ever, change after it is initially set up. All developers on a project should use identical copies of the workspace file. For this reason, you may want to place this file under source control so that a fixed version is available to all project users. Rational Rose RealTime does not provide explicit support for checking in or checking out this file.

After the source control manager adds the model to source control, the workspace should be manually added using your source control tool. Other users should then retrieve the workspace as part of their initial update of their local work area. This will ensure that all team members use the same source control settings for the project.

Defining Developer Work Areas

At this point, the source control administrator should think about how each worker (developer, integrator, and so on) will work individually and access specific versions (lineups) of a model. This usually involves defining labelling policies.

The source control administrator should provide guidelines to the rest of the team as to how work areas should be created for each developer. In some cases the source control administrator may need to actually create the work areas.

Defining work areas is tool dependent, and the steps required for setting up a work area for single stream and parallel stream development can be quite different. See the section below that corresponds to your source control system for more information:

- *ClearCase Work Area Setup* on page 152
- *SourceSafe Work Area Setup* on page 155
- *RCS/SCCS Work Area Setup* on page 158

Creation of Labels and Lineups

Labels, and the use of labels to create lineups, are crucial to any successful development strategy. There are many ways to use labels and lineups, though, and the specifics of each are highly specific to each organizations development environment and source control tools.

For an example of an effective labelling and lineup strategy, see *ClearCase Parallel Development: Sample Process* on page 171.

Manipulation of the Source Control Repository

It may be necessary to move or rename files in the repository. This should only be performed by someone who is familiar with the source control tool being used. In many development environments, such moving and renaming is always carried out by the source control administrator, who will be able to carry out the task most effectively.

See *Moving Controlled Units Between Model Directories* on page 67 for details on relocating controlled units.



Chapter 8

Source Control Tools

This chapter contains information on integrating Rational Rose RealTime with the different supported source control tools. Each source control tool requires specific configuration for proper use with Rational Rose RealTime.

The following source control systems are supported:

- Rational ClearCase (Windows and Unix) - see *Rational ClearCase* on page 144
- Microsoft Visual SourceSafe (Windows only) - see *Microsoft Visual SourceSafe* on page 152
- RCS and SCCS (Unix only) - see *RCS and SCCS* on page 155
- PVCS (Windows only) - see *PVCS* on page 159

Before starting, understand how the tasks described in this chapter relate to the overall team development process, described in *Team Development* on page 1.

For details on adding support for other source control systems, see *Customizing Source Control Interface Scripts* on page 189.

Note: *These sections assume you are already familiar with the capabilities and terminology of your chosen source control tool.*

Rational ClearCase

ClearCase uses a view model combined with a virtual file system that allows users to specify the lineup of file versions with which they want to work (a **config** spec controls the lineup used for a particular view). Rational Rose RealTime then sees the files in the current view just as if they were stored on a regular (non-ClearCase) file system. Rational Rose RealTime specifies the set of files that make up the model, and ClearCase provides the versions of these files determined by the view's **config** spec. Thus the model must be saved to a view directory that is not view-private in order for the files to be added to source control.

As mentioned in *Working in Isolation* on page 93, it is important that each developer have their own work area. When working with ClearCase, a work area is a view. This means that each developer should use a view that is dedicated for their sole usage and that should not be shared with other developers.

ClearCase has a feature allowing a new element "type" to be defined that includes specifying a merge and differencing tool that should be used on files of the new type. Rational Rose RealTime uses this to define an element type that applies to all Rational Rose RealTime files placed under source control. With this element type defined, all new Rational Rose RealTime files that are placed into a VOB are associated with that file type and will use Rational Rose RealTime Model Integrator as their default merge and differencing tool.

Registering a new ClearCase element type involves two steps. First, each ClearCase installation must be set up with a "type manager" that will map file extensions to the new element type and indicates which executable to invoke for merge and **diff** operations. Second, the new element type must be registered in all VOB's in which it will be used. The setup required for these steps is detailed later in this section.

General Recommendations

Windows NT/2000

Users should not access views through the MVFS mount point or M: drive. Instead, use the views through explicit drive mountings (usually X:, Y:, Z:). This improves "wink-in" and eliminates dependencies on view names.

Source Control Operation Behavior with ClearCase

Certain operations behave differently in ClearCase than as described in *Source Control Operations* on page 87. These differences are detailed below.

Get

Get is not able to retrieve a specific version of a file to a view because the version being observed in a view can only be changed via the **config** spec for that view. However, if a file is checked out, then **Get** may be used to replace the checked out file with a copy of a particular version of the file.

In the case where a file is not checked out, performing a **Get** on that file is the same as performing a **Synchronize** on the file.

See *Snapshot Views* on page 147 for details about how the get command works with Snapshot Views.

Synchronize

If a dynamic view is being used and the version of a file available in the view changes, then **Synchronize** will detect this and reload the file. Synchronize is a safer operation to perform than Get, as Synchronize will not lose any checked out changes, while **Get** may replace your checked out changes with the most recent version in the VOB.

Add

When adding files to source control, the ClearCase integration assumes that the containing directory is under source control and not currently checked out. If the containing directory is already checked out, the add will fail.

Label

Labelling of a directory will only apply the label to the directory element itself. To apply the label to the files contained within a directory, the **Recursive** option must be used.

UCM Integration

The UCM integration allows users working in a UCM VOB to assign activities to revisions from within the Toolset. In addition, you can Rebase, Deliver, and launch the Project Explorer from within the Toolset.

Activity Selection Combination Box

If ClearCase is enabled and if the model is stored in a UCM VOB, an activity selection combination box appears in the **Add**, **Check in** and **Check out** dialog boxes. If the activity box appears, an activity must be selected or created to continue with the operation. To create a new activity simply type it into the activity combination box.

The activity selector contains the list of activities that exist in the view containing the model. The view's current activity is automatically selected.

Run Project Explorer

To load the ClearCase project explorer application within its own process, click **Run project explorer**. You can continue using Rational Rose RealTime while the project explorer is loaded.

If the project explorer binary cannot be found, due to a problem with the path, an error message is generated in Rational Rose RealTime.

Rebase

To start a Rebase, click **Rebase from Stream** from **Tools > Source Control**. The Rebase ClearCase dialog box appears.

1. After the rebase has started, a Rational Rose RealTime dialog box appears, prompting you to synchronize the model with rebase changes. Ignore this dialog for now. You will return to Rational Rose RealTime after the rebase operation has been started.
2. Change to the **Rebase** dialog box and proceed with the operation.
3. After the initial rebase is finished, the **Rebase** dialog box will suggest that you build and test before checking in any undelivered work.

4. Test the rebase from within Rational Rose RealTime, then return to the rebase **ClearCase** dialog box.
5. To keep the changes, click **Complete** from the **Rebase** dialog box.

Deliver

Before using the **Deliver** command, you need to check in all changes to be delivered. After you check in your changes, click **Deliver Stream** from **Tools > Source Control**. The **Deliver ClearCase** dialog box appears.

1. From the **Deliver ClearCase** dialog box, select the activities to deliver.
2. Merge changes if applicable.
3. The Rational Rose RealTime session from which the rebase was started will be unavailable until the rebase is completed. Load another Rational Rose RealTime session to test changes in your own integration stream.
4. When satisfied with deliver, return to the **Deliver ClearCase** dialog box.
5. To save the changes, click **Complete** from the **Deliver ClearCase** dialog box.

Snapshot Views

Snapshot views are supported by Rational Rose RealTime. With ClearCase, you initiate a snapshot view update from within the Toolset, to work on files that you did not check out. The snapshot view contains the directory tree of source files.

You will want to use snapshot views if any of the following conditions apply:

- your computer does not support dynamic views
- you want to optimize build performance to achieve native build speeds
- you want to work with source files under ClearCase control when you are either disconnected from the network that hosts the VOBs, or connected to the network intermittently
- you want access to a view from a computer that is not a ClearCase host
- your project does not use ClearCase build auditing and build avoidance

Certain operations behave differently in ClearCase snapshot views than as described in *Source Control Operations* on page 87 and *Source Control Operation Behavior with ClearCase* on page 145.

Check in

When checking in files, ClearCase copies the new version to the VOB, as long as there is no successor version already in VOB.

If there is a successor, an error is returned from the scripts an will appear in the log. In order to check in your changes, you must first merge the most recent version from the VOB into your local copy. There are a couple of methods to perform the merge:

1. Update your snapshot view by clicking **Tools > Source Control > Update Snapshot View...** The **Update Snapshot View...** command helps you merge any changes. This is the preferred method since your snapshot view will also get any new elements that appear in the VOB.
2. If you know that the only the one element has changed in the VOB, use the context-menu **Source Control > Get** to retrieve the most recent version and perform the merge.

Check out

When working with a snapshot view, ClearCase marks elements in VOB as checked out. When checking out an element you will not be warned if a more recent version exists in VOB.

Get

The **Get** command for snapshot views in Rational Rose RealTime uses the update command to copy elements to a snapshot view. Unless you are certain that there are no new elements in the VOB, you can use **Get** to update existing model elements in your view. However, to get all new elements that may of been added to a VOB, use the **Update Tool**.

1. If the element is checked in, the **Get** command updates that element with the most recent version from the VOB.
2. If the element is checked out and is not the most recent in the VOB, the **Get** command prompts you to merge.
3. You cannot update an element which is already the most recent version in the VOB.

It may happen that the Get command updates a model element which references new elements that have not been copied into your snapshot view. This will happen if, after the **Get** operation, a dialog box appears prompting for the location of elements the Toolset could not find. If this happens, simply run the **Update Tool** to copy all new elements into your snapshot view.

Update

If a model is version controlled in a ClearCase snapshot view, the **Update Snapshot View...** menu item appears from the **Tools > Source Control** menu. Update launches the ClearCase update tool. When the update is completed, a dialog is displayed to help you **resync** the model with the new elements that have been copied to your view and elements that have been checked in.

Hijacking a File

If you work in a snapshot view while not connected to a network, you can modify a loaded model element that you have not checked out. This is what ClearCase calls hijacking a file. Once reconnected to the network (VOB), launch the **Update** tool to resolve hijacked files.

Deliver

When delivering a stream that has associated snapshot views, use the **Tools > Source Control > Update Snapshot View...** command to update the snapshot view before delivering. Click **Tools > Source Control > Deliver Stream** to deliver the changes.

Rebase

Use the project explorer to rebase, then update your snapshot view from within the Toolset.

Activities

Activities work just like dynamic views. The **check in**, **check out** and **add** dialog boxes contain an activity combination box if the snapshot view is UCM enabled.

ClearCase Workstation Setup

The following setup must take place on all workstations that will be accessing a VOB or view. For Windows NT and Windows 2000, this includes all workstations used for development. For Unix, this includes all machines that are view servers.

These steps will also need to be run on all machines that act as view servers for the ClearCase views used by Rational Rose RealTime. If you use ClearCase MultiSite, you will need to do this at all the sites where the VOBs containing the Rose elements are replicated.

You can determine which machines are view servers by typing

```
cleartool lsview
```

in a command window. The second item on each output line indicates the machine name where the view server is running. For example, if you see the following line in the output of the **lsview** command:

```
myview \\mymachine\vws\myview.vws
```

then "mymachine" is the name of the machine where the view server for **myview** exists.

For further details, see your ClearCase administrator.

Command Line Access to the Source Control Tool

For any user wishing to use Rational Rose RealTime's integration with ClearCase, **cleartool** must be accessible from the command prompt.

Element type setup: type manager

The following steps are required for making ClearCase clients aware of the new element type.

Windows NT/2000

In the instructions below, <atria-home> refers to the ClearCase installation directory. For newer releases, this typically is c:\Program Files\Rational\ClearCase. For older releases, this typically was c:\Atria.

- From a command prompt, run

```
rtperl <ROBERT_HOME>\bin\<ROBERT_HOST>\cc\mi_typeman.pl  
-atriahome <atria-home>
```

Unix

Use the \$ROBERT_HOME/bin/\$ROBERT_HOST/cc/mi_typeman script to install the type manager in each ClearCase installation. To set up the extensions and tool mappings, the user executing the script must have access to the following directories in the ClearCase installation:

```
/lib/mgrs  
/config/ui/icons  
/config/ui/bitmaps  
/config/magic
```

Use the following command line to set up the proper file extensions and tool invocations:

```
<ROBERT_HOME>/bin/<ROBERT_HOST>/cc/mi_typeman.sh install  
-server
```

ClearCase Options

Windows NT/2000

Rational Rose RealTime is case sensitive when looking for file names, so you must turn on the preserve case option for the ClearCase MVFS on Windows:

1. In the ClearCase HomeBase tool, select the **MVFS** tab. (The ClearCase Control Panel tool can be started from either the Windows Control Panel or from the **Administration** tab in the **HomeBase** tool)
2. Make sure the "preserve case" check box is checked.
3. The MVFS service must be restarted for this change to take effect.

Unix

There are no options that need configuring for Unix ClearCase.

ClearCase Repository Setup

Each VOB must be set up to allow files of the new element type to be created. Follow the steps that apply to your platform below for each VOB that will be storing Rational Rose RealTime files.

Windows NT/2000

Open a command prompt window and change directory to a path within the VOB in which you wish to register the type. To create the element type, use the following command syntax:

```
cleartool mkeltype -supertype text_file -manager  
petalrt_file_delta -c "RoseRT files" rosert_unit
```

Unix

Use the \$ROBERT_HOME/bin/\$ROBERT_HOST/cc/mi_typeman script to register the rosert_unit element type in each VOB using the following syntax:

```
<ROBERT_HOME>/bin/<ROBERT_HOST>/cc/mi_typeman.sh install  
-eltype -vob <vob_path>
```

Test the Type Manager

To determine if the rosert_unit element type has been successfully registered in the VOB, perform the following command from a command prompt after changing to a directory contained in the VOB:

```
cleartool lstype -long eltype:rosert_unit
```

A listing of the type details will verify that it is correctly registered.

ClearCase Work Area Setup

With ClearCase, a work area is defined by a view. Each developer accessing Rational Rose RealTime files in a VOB should use their own dedicated view. For an example of a developer view that could be used in a parallel development process, see *Creating a Developer View* on page 184.

Microsoft Visual SourceSafe

Microsoft Visual SourceSafe (VSS) stores and retrieves files on your local disk. Each VSS “project” has a working folder specified for it. Rational Rose RealTime saves model elements to and load elements from this working folder. VSS then checks those local files into and out of its repository. After modifying the local file, Rational Rose RealTime invokes a script that instructs VSS to check in a file.

For Visual SourceSafe, this involves setting up a project and associating a folder on your local disk with that project.

General Recommendations

On some systems, command line access to SourceSafe is extremely slow if the Visual SourceSafe explorer is currently running. If you find SourceSafe access to be slow, try closing any open SourceSafe explorers.

Note: *SourceSafe settings are not saved to disk immediately when they are set. If you change a setting, close the Visual SourceSafe explorer to ensure that the change will be used by future invocations of the SourceSafe command line tool.*

Source Control Operation Behavior with SourceSafe

Certain operations behave differently in Visual SourceSafe than as described in *Source Control Operations* on page 87. These differences are detailed below.

Label

Visual SourceSafe allows labels to be applied only to the most recent versions in the database.

Labelling a directory automatically applies the label to everything recursively contained within it.

SourceSafe Workstation Setup

Command Line Access to the Source Control Tool

The **ss SourceSafe** tool must be available from the command line. To test this, open a command prompt and type “ss about”. If an error occurs, you will need to modify your path so that the ss tool can be found.

Set Project Mapping Option

Visual SourceSafe must be configured to determine which projects correspond to file system directories. Follow these steps to correctly set up Visual SourceSafe for this:

1. In Visual SourceSafe Explorer, click **Tools > Options**.
2. Click the **Command Line Options** tab.
3. Set the **Assume project based on working folder** check box.

Let Visual SourceSafe Know Which Database to Use

Rational Rose RealTime will not be able to determine which database to use if you have more than one SourceSafe database configured on your system unless the SSDIR environment variable is set. Visual SourceSafe uses SSDIR to determine which database to use. This variable tells Visual SourceSafe where to find the srcsafe.ini file for the database you wish to use.

You should set the SSDIR variable in the System control panel, or with a shell script. To set SSDIR in a shell, use the following command:

```
set ssdir=<path to srcsafe.ini>
```

The path given should be the directory that contains the srcsafe.ini file for the database you wish to use.

Note: Do not put a space between the equal sign and the location of the srcsafe.ini file.

SourceSafe Repository Setup

Rational Rose RealTime does not support multiple checkouts with single stream source control systems. For proper integration with Rational Rose RealTime, the Visual SourceSafe database should be configured to not allow multiple checkouts.

A common practice is to create a project in Visual SourceSafe that will serve as a container for all Rational Rose RealTime models that will be placed in the repository.

SourceSafe Work Area Setup

A local work area for SourceSafe is a directory that maps to a project in your SourceSafe database. For Rational Rose RealTime to integrate properly with SourceSafe, the working directory of the project must be set to the corresponding local directory. See the Visual SourceSafe documentation for details on setting up projects and working folders.

Test

To ensure that SourceSafe is correctly configured for your database and work area:

1. In to your local model directory, and type `ss project`.
2. Then type `ss dir -E`.

There should be no prompts for *username* or *password*.

RCS and SCCS

Rational Rose RealTime is designed to work with SCCS and RCS through a set of scripts that are provided. Rational Rose RealTime saves model elements as individual files which are stored and version controlled by SCCS/RCS.

Neither RCS nor SCCS directly support directory hierarchies, and Rational Rose RealTime uses hierarchical storage by default to store the model elements. To support a hierarchical repository, Rational Rose RealTime creates a separate RCS/SCCS storage directory for each level in the model hierarchy.

For example, the repository structure might look something like the following, where *<dir>* indicates a directory:

```
<repository>
  <models>
    <RCS>
      MyModel.rtmdl,v
    <MyModel>
      <RCS>
        LogicalView.rtlogpkg,v
        ComponentView.rtlogpkg,v
        UseCaseView.rtlogpkg,v
        DeploymentView.rtlogpkg,v
      <LogicalView>
        <RCS>
          ...
      <ComponentView>
        <RCS>
          ...
```

Repository Mapping Files (.rmf)

Each developer in a team will use their own local working directory for working on models. A special mapping file is then required to map the local working directory to the repository directory representing the root of the hierarchy. This map file is referred to as a Repository Mapping File (RMF). Each line in the RMF is a file name prefix mapping that works similar to the virtual **pathmap** mechanism within Rational Rose RealTime. Each entry consists of two path prefixes, separated by an equals sign (=).

Example:

```
/home/john_doe/RoseRT/models=/repository/models
```

By applying this map file, the Rational Rose RealTime **rsc** integration will map local working directory

```
/home/john_doe/RoseRT/models
```

to repository directory

```
/repository/models/RCS
```

The RMF may contain multiple entries. The first valid prefix will be used, and successive substitutions will not be applied.

Before determining if an RMF source prefix is valid for a given path, both the source and destination prefixes will have environment variable substitution performed on them. Thus, assuming every user had a RoseRT/models directory in their home directory, the following RMF file could be used by all users working from the given repository:

```
/home/$user/RoseRT/models=/repository/models
```

Note: *The RMF must not contain softlinks to directories. It must contain the actual path to the directory.*

Source Control Operation Behavior with SCCS

Certain operations behave differently in SCCS than as described in *Source Control Operations* on page 87. These differences are detailed below.

Label

SCCS does not support labelling. All labelling operations will be unavailable from the Toolset.

RCS/SCCS Repository Setup

The repository root directory must be created. Be sure to place appropriate access permissions on the directory so that the users will have the required access to the files in it.

If you will be using a global RMF for all users accessing the repository, you should create it now and place it in a location accessible by all users.

RCS/SCCS Workstation Setup

Command line access to the source control tool

The rcs/sccs executables must be available from your path in order for Rational Rose RealTime to integrate with them.

Create an RMF File

Use a text editor to create the RMF file that will contain the mapping between your local working directory and the RCS/SCCS repository. Create an entry in your RMF to point to the working directory set aside for your models (create a working directory if you do not already have one).

Set RMF Environment Variable

The RCS/SCCS scripts examine an environment variable to determine what RMF to use.

For RCS:

- Set the `ROBERT_RCS_MAPFILE` environment variable to the name of the file containing the map entry. For example:

```
setenv ROBERT_RCS_MAPFILE ~/MyRCSMap.txt
```

For SCCS:

- Set the `ROBERT_SCCS_MAPFILE` environment variable to the name of the file containing the map entry. For example:

```
setenv ROBERT_SCCS_MAPFILE ~/MySCCSMap.txt
```

RCS/SCCS Work Area Setup

To populate the local work area initially from the repository, use the provided `cm_update` script:

For RCS:

- Run the following command from a command line:

```
rtperl $ROBERT_HOME/bin/<platform>/cmscripts/rcs/cm_update  
-D <dir_name_and_path> -R
```

Where `<platform>` is the name of your platform (for example, `sun5`) and `<dir_name_and_path>` is the name of your local working directory with the full path to it, for example:

```
/home/john_doe/RoseRT/models
```

For SCCS:

- Run the following command from a command line:

```
rtperl $ROSERT_HOME/bin/<platform>/cmscripts/sccs/cm_update  
-D <dir_name_and_path> -R
```

Where *<platform>* is the name of you platform (for example, sun5) and *<dir_name_and_path>* is the name of your local working directory with the full path to it, for example:

```
/home/john_doe/RoseRT/models
```

PVCS

Rational Rose RealTime is designed to work with PVCS through a set of scripts. The PVCS source control scripts are supported on Windows only. PVCS lets you organize your versioned files using project databases, projects, and subprojects. Configuration files are used to add directives to PVCS commands. The PVCS scripts use configuration files to map the current working directory to a PVCS database.

Source Control Operation Behavior with PVCS

Certain operations behave differently in PVCS than as described in *Source Control Operations* on page 87.

Label

PVCS does support labeling, however the scripts do not. All labelling operations will be unavailable from the Toolset.

PVCS Workstation Setup

Command Line Access to the Source Control Tool

The PVCS command line tools must be available from the command line. To test this, open a command prompt and type "get -help". The command should return help for the get command. The first line of the help will read:

```
GET - extract revisions from PVCS archives
```

Let PVCS Know Which Database to Use

Before adding a Rational Rose RealTime model to a PVCS database, you will need to define the initial (root) directory to use for the version control repository (for example, database or archive). This is done by creating a file named `pvc.cfg` in the directory where you save your model (for example, the directory where the `.rtmdl` file is located). This file will contain a single directive on one line:

```
VCSDir C:\pvcs\rrtmodels
```

where `C:\pvcs\rrtmodels` is the root directory of the PVCS archive in which the versioned model files will be stored. This allows the scripts to map the work directory to a repository. A sample file is provided in the `scripts` directory.

Rational Rose RealTime uses a hierarchical structure to store files. When using these scripts, a `pvc.cfg` file will automatically be created in each of the Rational Rose RealTime sub-directory to create a comparable directory structure in the repository. If a `pvc.cfg` file already exists in any one of these sub-directories, it will not be overwritten. It is the responsibility of the user to make sure that such a file contains a valid **VCSDir** directive.

Note: Any sub-directory will automatically have repository sub-directory created according to its parent's **VCSDir**.

Note: The repository path has only been tested with mapped drives under Windows. No tests have been conducted using UNC paths.

PVCS Repository Setup

This assumes that you are familiar with PVCS configuration and have already created a database for your model. Before using PVCS with Rose Real Time you must change a small set of PVCS configuration parameters. A file called `pvcMaster.cfg` located with the `scripts` contains a sample configuration file that will work with Rational Rose RealTime. The configuration changes are described below.

Archive Suffixes

The default archive suffixes must be changed so that the versioned filename does not get changed. By default the `??v__` suffix template will have to be changed so that extensions of more than 3 characters are maintained. We suggest using `+`, `v` as the suffix template.

Write Protect Workfiles

Ensures that checked in files are write protected and not deleted.

One Lock Per Version/User

Rational Rose RealTime does not support multiple checkouts with single stream source control systems. For proper integration with Rational Rose RealTime, the **pvcs** database should be configured to not allow multiple checkouts.

Registering a New Configuration

There are a couple of ways to do this. This example demonstrates one method, which may or may not be appropriate for all project configurations.

The configuration parameters required for Rose Real Time are located in a file called `pvcsMaster.cfg`. This file should be located with the PVCS scripts. It is suggested that you do not modify the options marked as required.

Example of registering a master configuration file with PVCS: This file should be located in a write-protected directory so only the configuration manager can change it. The master configuration file should then be enabled in the following manner:

```
vconfig -cI:\PVCSRepository\pvcsMaster.cfg  
I:\bin\pvcs\vmwfv.c.dll
```

where `I:\PVCSRepository\pvcsMaster.cfg` is the master configuration file
`I:\bin\pvcs` is the directory where the PVCS binaries are found.
Alternately, this information can be added to PVCS' `master.cfg` file.

PVCS Work Area Setup

Creating a Working Directory Tree From an Existing Archive

To create a working directory tree from an archive, you can use the `cm_update` script. Create a starting `pvcs.cfg` file that points to the archive directory. You can then issue the following command:

```
rtperl -w %ROSET_HOME%\bin\win32\cmscripts\Pvcs\cm_update -R
```

This will recreate the correct directory tree for your project.PVCS repository setup.



Chapter 9

Model Validation

The purpose of model validation is to produce a consistent and complete model within the Toolset. Every time a model is loaded into Rational Rose RealTime, the Toolset does a complete pass over the model looking for model inconsistencies and unresolved references. If any inconsistencies or unresolved references are found, then the elements that have these problems are either deleted or repaired. The containing controllable units for the affected elements will also be marked as modified.

When individual controlled units are loaded into the model (e.g., when getting a new version of a class), the Toolset will validate only the elements affected by replacing this unit.

In order to avoid having the same validation problems reported in the future, the controlled units which were modified by the Toolset should be saved. If the model is under source control, then these units should be checked out and checked in.

Information about each validation problem is written to the Rational Rose RealTime log. If you have a large number of validation messages, then the log may overflow and some messages will be lost. You can change the log size by opening the **Tools > Options** dialog and editing the value in the **Log size** field in the **General** tab. Also, the **Log warnings** option in this dialog should be checked to ensure that all messages will appear in the log.

What is a Model Inconsistency?

A model inconsistency is reported when the combination of the elements in the model has resulted in a violation of a modeling constraint.

The following example scenario creates a model inconsistency by violating the constraint that the initial state must have at most one outgoing transition:

1. Create two capsule classes C1 and C2 where C2 is a subclass of C1.
2. Create a state S1 in the state machine for capsule C1.
3. Create a transition T1 from the initial state to S1 in the state machine for capsule C2.
4. Save C2 as a controlled unit.
5. Delete transition T1 from the state machine for C2.
6. Create a transition T2 from the initial state to S1 in the state machine for C1.
7. Reload C2 from the saved file.

These actions will result in a model inconsistency where the state machine for capsule C2 would have two transitions originating in the initial state. The Toolset resolves this inconsistency by excluding T2 from the state machine of C2.

These actions will result in the following message being written to the log:

```
Warning: Removed transition "T2" from class "C2".
```

A model inconsistency is often caused by saving changes to one controlled unit without saving the related changes to other controlled units. The related changes are usually classified by the Toolset as secondary edits. See *What are Primary and Secondary Edits?* on page 81. It can also be caused by opening a model that is composed of an inconsistent lineup of unit versions.

Model inconsistencies can also be reported for the following modeling constraints:

- Circular inheritance loop - there cannot be a cycle in the inheritance graph for a class
`Warning: Removed Generalization from class "C1" to class "Logical View::C2".`
- Multiple transitions from the same pseudostate - initial states and junction points can have at most one outgoing transition; also the true and false branches of a choice point can each have at most one outgoing transition
`Warning: Removed transition "T2" from class "C2".`
- Connector to an unwired port - a connector can only be attached to wired ports
`Warning: Removed connector "c1" from class "C2".`
- Event guard with no events - an event guard must have at least one event defined for it
`Warning: Removed empty trigger event on transition "t1" in class "C2".`

What is an Unresolved Reference?

An unresolved reference is reported when the combination of elements in the model has invalidated a reference from one element to another.

The following example scenario creates an unresolved reference from a transition to a state:

1. Create two capsule classes C1 and C2 where C2 is a subclass of C1.
2. Create two states S1 and S2 in the state machine for capsule C1.
3. Create a transition T1 from S1 to S2 in the state machine for capsule C2.
4. Save C2 as a controlled unit.
5. Delete state S1 from the state machine for C1.
6. Reload C2 from the saved file.

These actions will result in an unresolved reference related to the transition T1. Since state S1 has been deleted, the transition (and its associated junction points) cannot be properly created and so they are deleted. Similar unresolved references will exist because of the view elements in the state diagram for C2.

These actions will result in the following messages being written to the log (see the next subsection for a detailed description of these messages):

```
Error: Unresolved reference from Capsule "C2"
      to Item with name :TOP:S1
      by Refinement "<unnamed>".

Error: Unresolved reference from State "TOP"
      to StateVertex with name :TOP:S1:Junction1
      by Transition "t1".

Warning: Removed transition "t1" from class "C2".

Warning: Unresolved reference to State with name S1.
         in StateView S1 in State Diagram: Logical View / C2 -
         Top State

Warning: Unresolved reference to State with name S1.
         in StateView S1 in State Diagram: Logical View / C2 -

Warning: Unresolved reference to JunctionPoint with name
         :TOP:S1:Junction1.
         in JunctionPointView :TOP:S1:Junction1 in State
         Diagram: Logical View / C2 -

Warning: Unresolved reference to InitialPoint with name
         Initial.
         in InitialPointView Initial in State Diagram: Logical
         View / C2 -

Warning: Unresolved reference to JunctionPoint with name
         :TOP:S1:Junction1.
         in JunctionPointView :TOP:S1:Junction1 in State
         Diagram: Logical View / C2 - Top State
```

```
Warning: Unresolved reference to JunctionPoint with name
:TOP:S2:Junction1.
    in JunctionPointView :TOP:S2:Junction1 in State
Diagram: Logical View / C2 - Top: S2
```

```
Warning: Unresolved reference to JunctionPoint with name
:TOP:S2:Junction1.
    in JunctionPointView :TOP:S2:Junction1 in State
Diagram: Logical View / C2 - Top State
```

As with a model inconsistency, an unresolved reference is often caused by saving changes to one controlled unit without saving the related changes to other controlled units. It can also be caused by opening a model that is comprised of an inconsistent lineup of units.

Unresolved references can also be reported for the following situations:

- classifier role referencing a missing classifier
- connector referencing a missing port or port role
- interaction instance referencing to a missing classifier role
- generalization, realization, association, aggregation, or dependency relationship referencing a missing class, capsule, protocol, use case, package, or component
- port referencing a missing protocol
- signal referencing a missing class
- component referencing a missing class, capsule, protocol, or package
- port event referencing a missing port or signal
- protocol role event referencing a missing signal
- message referencing a missing interaction instance
- component instance referencing a missing component
- refinement of a missing inherited state, transition, capsule role, port, or connector

What do the Errors/Warnings Mean?

The error/warning messages for a model inconsistency should be self explanatory. Each message typically describes the deletion or exclusion of a model element.

The error/warning messages for an unresolved reference may require a bit more explanation. The elements in a model fall into two categories: model elements and view elements. A model element is the underlying UML element (for example, class, state, transition, classifier role). A view element is the a graphic object representing a model element within a diagram.

Unresolved References Between Model Elements

These messages tend to use the following templates:

```
Error: Unresolved reference from <element type> <element name>
      to <element type> with name <element name>
      by <element type> <element name>
```

where:

<element type> describes a kind of model element, for example, Capsule, State

<element name> is the name of a model element, for example, C1, S1

Unresolved References from a View Element

These messages tend to use the following template:

```
Warning: Unresolved reference to <element type> with name
<element name>
        in <view type> <element name> in <diagram type>
<diagram name>
```

where:

<element type> describes a kind of model element, for example, Capsule, State

<element name> is the name of a model element, e.g., C1, S1

<view type> describes a kind of view element which is a presentation of a model element in a diagram; these are usually formed by adding View to the end of the element type, for example, **StateView**.

Validating Names

In conjunction with model validation, the Rational Rose RealTime Toolset checks the names of the model elements to ensure that they are valid. If a name conflict is detected, then the Toolset will rename one of the conflicting elements. This ensures that names are unique where required.

As with model validation problems, a name conflict can be caused by saving changes to one controlled unit without saving the related changes to other controlled units. It can also be caused by opening a model that is comprised of an inconsistent lineup of units.

An example scenario to create a name conflict is:

1. Create two capsule classes C1 and C2 where C2 is a subclass of C1.
2. Create a state S1 in the state machine for capsule C1.
3. Save C1 as a controlled unit.
4. Delete state S1 from the state machine for C1.
5. Create another state S1 in the state machine for C2.
6. Reload C1 from the saved file.

In the state machine for C2 we have a locally defined state S1 and an inherited state S1. These two states have a name conflict since a state name must be unique among all states in its containing state. The Toolset will rename one of the states and output the following message to the log:

```
Warning: Renamed State "S1" to "S1_0" in class/package "C1".
```

A name conflict can also be caused by multiple users making changes to related classes. For example:

1. Assume we have a model where class C2 is a subclass of class C1.
2. One user adds an attribute named 'm_name' to a C1
3. Another user adds an attribute with the same name to C2

If the modified versions of C1 and C2 are loaded in as part of the same model, then there will be a name conflict between the attributes.

See Naming Guidelines in the *Toolset Guide* for more information on the naming rules.



Chapter 10

ClearCase Parallel Development: Sample Process

This chapter details how to set up a parallel development process to use Rational Rose RealTime with Rational ClearCase. The process presented here is an illustrative example meant to explain parallel development and is not in any way a definitive guide for working with ClearCase. Feel free to use this process as is, or to modify and customize it as necessary to fit your project's needs.

Many of the techniques presented in this example are not specific to either ClearCase or parallel development, although the details certainly are. This example assumes a homogeneous ClearCase installation (for example, Windows NT, Windows 2000, or Unix) and does not address the details of how to setup ClearCase in a multi-sited environment. It should be noted that view profiles are not recommended in a mixed ClearCase installation and are used in this example for simplicity only. The process of installing more advanced configurations of ClearCase does not affect the usage of Rational Rose RealTime, but requires more advanced knowledge of ClearCase itself. For that reason, this example uses a simple ClearCase configuration to illustrate the parallel development process. Please refer to the ClearCase product documentation for help with multi-site and heterogeneous installations and administrations.

Note: Throughout this example, the prefix *TC* is used to indicate an identifier that is unique to the project being worked on. Using distinct labels for each project will help keep their development progress self-contained and more manageable.

Parallel Development Overview

The benefits of a proper parallel development process are:

- reduced contention for checkouts
- private version streams for development activities
- shared build results to reduce incremental development times
- stable and controlled evolution of the system being developed

As explained in *Parallel Stream Versioning* on page 94, the integration branch plays a central role in most parallel development strategies. In this example, /main is used as the integration branch. All automated builds are generated from the integration branch, all lineups are created from elements on the integration branch, and all development is based on the integration branch.

Automated builds are performed on the contents of the integration branch. To ensure reproducible builds (and provide wink-in of build artifacts), the latest version of each file and directory on the integration branch is labelled with an identifier such as TC_BUILDFILES. Using a label instead of a timestamp or whatever happens to be in view insures that a build is completely reproducible. If the version of a file labelled with TC_BUILDFILES causes compile problems, then a previous version of the file can be used simply by applying TC_BUILDFILES to the appropriate version and re-building incrementally.

When the build is successful, a new label is generated of the form TC_BASELINE_NNN. The label is then applied to the exact version of each file that was included in the build (for example, every version that labelled with TC_BUILDFILES is now labelled TC_BUILD_NNN).

As far as development is concerned, no actual development occurs on the integration branch. All development is carried out on private branches, one per development activity. Each private branch is based off of a lineup on the integration branch, conveniently labelled by the automated build process. Since the file versions used in the build are also used by developers, wink-in of build artifacts comes for free.

After a development activity finishes, an integrator is given the branch name and merges the changes for that activity onto the integration branch when time permits.

The following diagram illustrates a typical version tree for an element in this process:

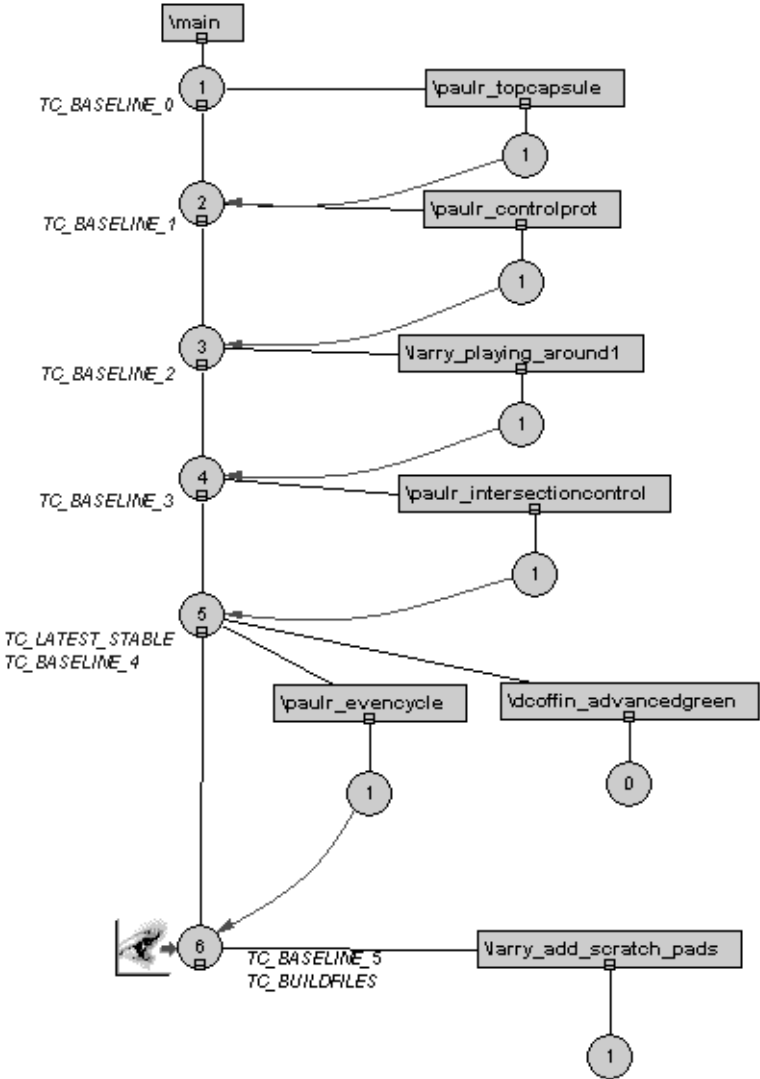


Figure 44 Version Tree Example

The remainder of this chapter explains the details behind the process just described. See the following sections:

- *Using View Templates* on page 175
- *ClearCase Entities* on page 176
- *Initial Setup* on page 176
- *Automated Builds* on page 181
- *Developer Process* on page 184
- *Integration Process* on page 186

Making Design Changes in Parallel

Generally, if the editing you do on a private branch causes a problem in the C++ environment, it will also cause a problem in the Rational Rose RealTime environment. Considerations should be made regarding the number of elements that must be checked out to make design changes versus syntax changes. As a rule, do not make design changes in parallel because you face the danger of having difficulty merging them together.

Things that should be safe to perform in parallel:

- modify transition code
- modify member function code
- add attributes/members to classes

Things that should not be performed in parallel:

- deletion of model elements
- renaming of model elements
- relocation of model elements

By saying these operations should not be done in parallel, this means that the developer/designer making these changes should ensure that no-one else will be modifying the elements affected before the next integration. It certainly should not be interpreted as "don't use a private branch for this work".

Using View Templates

To ensure that developers use a common base for their view's **config** spec, and to make it easier to work on private branches, view templates are used. A view template specifies the integration branch to work from, lists labelled checkpoints that can be used to base a private branch on, and includes a **config** spec template that can be filled in with additional **config** spec rules.

Windows NT/2000

This functionality is provided with ClearCase 3.2.1 for Windows NT/2000 through **View Profiles**.

Unix

ClearCase for Unix does not include support for View Profiles. To replicate similar functionality, Perl scripts exist to provide essentially the same functionality. The **vtadmin** and **vtsetview** scripts are located in the \$ROSET_HOME/bin/\$ROSET_HOST/cc/vt directory.

Every developer will need access to a common location from which the view templates will be accessed. The view template scripts look for the view templates in the directory named by the CCVIEWTEMPLATES environment variable.

Each view template consists of the following parts:

- A list of labels that indicate integration branch lineups
- A **config** spec for browsing any specific integration branch lineup
- A **config** spec for performing a development activity on a private branch
- A **config** spec used by the integrator
- A **config** spec used by the builder

Since the **config** specs for each project will be different, a view template must be generated for each project.

See “View Template Script Usage” on page 187 for complete details on how to use the view template scripts.

ClearCase Entities

This development process will require the creation and usage of the following ClearCase entities.

Views

A separate view will be needed for the integrator, for the builder, and for each developer.

View Template

A view template will be needed to provide a standard **config** spec for each developer.

Labels

Labels will be used to define various lineups. Significant labels include:

- **TC_BASELINE_0**: Represents the initial state of the project.
- **TC_BUILDFILES**: This label indicates what element versions should be included in the next automated build. Only the builder should use this label.
- **TC_LATEST_STABLE**: This label is applied to the most recent stable lineup on the integration branch.

Note: *This label is not fixed. The elements it refers to will change whenever a new stable lineup is established.*

Initial Setup

Before starting with the parallel development process outlined below, it is assumed that the model that will be worked on is already under source control in a VOB. See *Set up a Source Control System and Repository* on page 140 for details on this.

Create the Integrator View

All project setup can occur from the integrator view. The integrator view will see the latest versions of elements on the integration branch, which in this case is /main. The **config** spec should look like this:

```
element * CHECKEDOUT
element * /main/LATEST
```

Views are created with this **config** spec by default, so create a view with the name tc_int. If the integrator role will be played by multiple team members, be sure to choose a storage location for the view that will provide suitable performance for all. As always, integrators should not share views and so no two integrators should use this view at the same time.

Create Project Labels

The standard project labels mentioned above should now be created. These labels include TC_BASELINE_0, TC_BUILDFILES, and TC_LATEST_STABLE.

Each of these labels should be created before starting work on the project. A label type can be created with the following **cleartool** syntax:

```
[x:\dev]cleartool mklbtype -c "Initial Project State"
TC_BASELINE_0
```

Created label type "TC_BASELINE_0".

Create Initial Lineup

After the labels have been created, the initial lineup label should be applied to the VOB (dev is the BOB being used in this example):

```
[x:\dev]cleartool mklabel -recurse TC_BASELINE_0 \dev
```

The initial model should be a valid stable model, so the TC_LATEST_STABLE label should be applied to all versions that are covered by the initial lineup:

```
[x:\dev]cleartool mklabel -recurse -version TC_BASELINE_0 -
replace TC_LATEST_STABLE \dev
```

Creating the Developer View Template

To ensure consistent and controlled access to the model, and to ease the usage of lineups and private branches, all developers should derive their **config** specs from a common base.

There are two primary functions that developers will be performing, and each requires a different **config** spec:

- Browsing: allows the view to see the latest stable lineup on the integration branch.
- Development: this sees a snapshot of the integration branch based on a labelled stable lineup, and branches files to a developer-private branch when files are checked out.

The rules for the browsing **config** spec are as follows:

```
element * TC_LATEST_STABLE
element * /main/LATEST
```

The TC_LATEST_STABLE label in the rule above can be changed to a different label if a developer wishes to view a lineup other than the latest. Optionally, the **-nocheckout** modifier can be added to the above rules so that checkouts can not occur accidentally while browsing.

For the development **config** spec, the rules should be:

```
element * CHECKEDOUT

element * ...\paulr_timing\LATEST
mkbranch paulr_timing

element * TC_BASELINE_5
element * \main\LATEST
```

In these rules, paulr_timing is the name of the private branch on which the development is taking place and TC_BASELINE_5 is the stable lineup that the development is based on. The rules have the following meaning:

- All versions checked out to the view will be seen
- If there is no checked out version, then the latest version on the private branch will be seen.
- If there is no version on the private branch, then take the version labelled by the lineup.

- If an element from the lineup is checked out, immediately branch it to the private branch, and check out the newly branched version.
- If an element does not exist on the private branch and does not have the lineup label applied to it, simply choose the latest version on the main branch.

Windows NT/2000

The developer view template can be implemented using view profiles by creating and maintaining a view profile, and having each developer associate their view with the view profile. Using the ClearCase View Profiles tool, create a new view profile using the supplied wizard, entering the following details:

- Name: tc_dev_profile
- Include the storage VOB for the model
- The work for the profile will not be done on a branch. (Though private branches will be used by developers, the view profile itself will provide a **config** spec to be used only for browsing the integration branch, not for making changes on it.)
- Give the label for the initial lineup, TC_BASELINE_0, as the checkpoint label for creating private branches. This is not used for the default **config** spec, but instead marks TC_BASELINE_0 as a possible branching point.
- The diagram annotation can be modified as appropriate.

The default browsing **config** spec produced will look similar to the following:

```
# [CC_PROJECT - Checked Out Rule
element * CHECKEDOUT
#
#       Any modifications to the Profile config spec should
#       be made following this comment.
# CC_PROJECT]

# [CC_PROJECT - Profile Config Spec
#       Do not directly modify the text below, it has been
#       automatically generated by the ClearCase View Profile
#       Tool. To change the Profile config spec, use the
#       ClearCase View Profile Wizard to update the Profile
#       status as needed.
element * \main\LATEST
# CC_PROJECT]
```

Unfortunately, this **config** spec will let developers see changes that have been merged to the integration branch but that have not yet been built and tested. What is wanted instead is a **config** spec that shows the latest stable build at any point in the development process. The change required is:

```
# [CC_PROJECT - Checked Out Rule
element * CHECKEDOUT
#
#       Any modifications to the Profile config spec should
#       be made following this comment.
# CC_PROJECT]

element * TC_LATEST_STABLE
# [CC_PROJECT - Profile Config Spec
#       Do not directly modify the text below, it has been
#       automatically generated by the ClearCase View Profile
#       Tool. To change the Profile config spec, use the
#       ClearCase View Profile Wizard to update the Profile
#       status as needed.
element * \main\LATEST
# CC_PROJECT]
```

The view profile is now ready for developers to use.

Unix

Use the supplied **vtadmin** script to create a new template. The following command syntax can be used:

```
vtadmin -mktemplate -template tc -lateststable TC_LATEST_STABLE
-buildlabel TC_BUILDFILES -integrationbranch /main -snapshot
/vobs/TrafficControl
```

After the command finishes, a template with the supplied parameters will have been created in the \$CCVIEWTEMPLATES directory, and is now ready for use in the project.

To add the initial lineup label as a supported branching point, use the following **vtadmin** invocation:

```
vtadmin -addlineup -template tc -baselinelabel TC_BASELINE_0
```

Automated Builds

To provide the ability to selectively choose the versions of files that go into the build, the builder will select all versions that are labelled with the build label `TC_BUILDFILES`. This allows flexibility in changing the exact versions that go into the build should it be needed (for example, if the most recent version of a file contains code that does not compile, then the previous version can be labelled instead).

There are several steps involved in the build:

- Label Build Files
- Perform Build
- When the Build Completes Successfully
 - Create a new lineup label and apply to build file versions
 - Apply `TC_LATEST_STABLE` to build file versions
 - Make New Lineup Available to Developers

Before any of this can occur, though, the build view must first be created.

Create the Build View

The build view is similar to the integrator view in that it selects files from the integration branch, but different in that it needs to select labelled versions when performing the build.

When performing the labelling, the latest version of files on the integration branch need to be in view for the labelling to select the correct file versions. This **config** spec is identical to the one presented above for the integrator.

When performing a build, the build view must see the labelled version of all files that are contained in the build. For files and directories that are not labelled, it suffices to select the latest version on the main branch. The following **config** spec rules capture these requirements:

```
element * TC_BUILDFILES
element * \main\LATEST
```

For the build view to be used for both labelling and building, the **config** spec for the view must be switched back and forth. This can be done by having text files that contain the two **config** specs and using **cleartool setcs** to invoke the appropriate **config** spec.

Depending on your development environment, it may be possible to use the integrator view for labelling and leave the build view always configured to pick up the TC_BUILDFILES labelled files.

A typical name for the build view is tc_build.

Unix

The view template scripts produce a text version of the build and integrator **config** spec rules indicated above. Use the **vtsetview** script to select the appropriate **config** spec rules into the build view.

Label Build Files

After ensuring that the current view has the integrator **config** spec, apply the TC_BUILDFILES label to the latest version of each element on the integration branch. The following command will do this:

```
cleartool mklabel -recurse -replace -version \main\LATEST  
TC_BUILDFILES \dev
```

Perform Build

After ensuring that the current view has the builder **config** spec, perform the build.

If the build does not complete successfully, or if the produced build does not pass sanity testing, determine if it is possible to fix the problem simply by backing up the version of a file used. If so, apply the TC_BUILDFILES label to the earlier version of the file and restart the build. Continue until a successful build is produced.

If there are build problems that cannot be resolved in the above manner, then ensure that the developers responsible for the problem are notified so that the next build will be successful.

When the Build Completes Successfully

Create a New Lineup Label and Apply to Build File Versions

Create a label that will encompass all versions used in the build just completed. This should be a unique label in a regular form, such as TC_BASELINE_ANN, where ANN is an integer preferably generated automatically in an incremental manner from the previous lineup label.

Apply the label to all versions that were used in the build:

```
cleartool mklabel -recurse -replace -version TC_BUILDFILES  
TC_BASELINE_NNN \dev
```

If you wish to prevent the lineup contents from being changed in the future, you may wish to lock the TC_BASELINE_NNN build label at this point.

Apply TC_LATEST_STABLE to Build File Versions

As a convenience, the TC_LATEST_STABLE label is used to show the most recent successful stable build. To update the versions that TC_LATEST_STABLE applies to, use a similar **mklabel** invocation to the one presented above.

Make New Lineup Available to Developers

The newly labelled lineup should now be exposed for developers to use as a branching point for private branches. This is done by adding the TC_BASELINE_NNN label to the view template.

Although it may seem that TC_LATEST_STABLE could be added as a potential branching point label, this is not the case. Branching points are intended to be unchanging specifications of a lineup of versions. However, TC_LATEST_STABLE changes with every build, and is therefore not appropriate for use as a branching point.

Windows NT/2000

Using view profiles, the build label should be added to the tc_dev_profile view profile. This is done in the ClearCase View Profiles editor by using the context menu on the tc_dev_profile profile.

Unix

Use **vtsetadmin** to add the build label to the view template:

```
vtadmin -addlineup -template tc -baselinelabel TC_BASELINE_NNN
```

Developer Process

Each development activity is completed by a single developer and is performed on a private branch specific to that activity. Again, each developer requires their own view. The view is based on a branching point on the integration branch identified by a build label.

A unique branch name must be chosen that identifies the work being performed (such as paulr_timing). The view's **config** spec rules are set up to automatically check out and branch files from the branching point to the private branch. As well, new elements created during the development activity are immediately branched to the private branch.

Because the branch is hidden from other developers, the user may check in incremental changes to the branch. When the developer is satisfied that their changes are completed and ready to be integrated, the developer informs the integrator that all changes on the private branch are ready for integration.

By basing developer private branches off of labels that correspond to the versions used by automated builds, each developer will be able to reuse most of the build results in the form of winked-in derived objects. This significantly reduces the amount of building that is required by each developer when they make changes.

Creating a Developer View

It is important to note that *each developer needs their own view*. Under no circumstances should multiple users work from the same view.

Windows NT/2000

After creating the view, associate the view with the tc_dev_profile View Profile. The view will be set up for browsing as per the description in *Creating the Developer View Template* on page 178.

Unix

After creating the view, use the **vtsetview** script to set the view **config** spec to the default browsing **config** spec using the following command:

```
vtsetview -setview browse -template tc
```

The view will now show the latest stable build of the model.

Starting a Development Activity

Each development activity is performed on a private branch. The name of the private branch should be appropriate to the activity being worked on. One strategy for avoiding branch name clashes is to start each branch name with the user id of the developer doing the work (for example, paulr_timing).

Windows NT/2000

To start an activity, use the **Set Up Private Branch** wizard that is available from ClearCase HomeBase. Rather than base the branch on the elements currently in view, choose to use a different branch point. On the version selection page, click by **View Profile checkpoint**, and select the integration branch label you wish to work from, which is likely the most recent label in the list.

Unix

Use the **vtsetview** script with the **-listbaselines** option to see what lineups are available for basing the private branch on. To start the private branch, use the following invocation of **vtsetview**:

```
vtsetview -startbranch -template tc  
          -brname paulr_timing -brpoint TC_BASELINE_4
```

Working on a Development Activity

After the view has been set up like this, the model should be loaded into Rational Rose RealTime. Work now proceeds until the entire development activity is complete. The developer may check in intermediate results, as they will not be seen by other developers since the changes will all occur on the private branch.

Finishing a Development Activity

When all development is complete on the activity, and everything submitted to source control, the changes are ready to be propagated to the integration branch. The propagation is performed by the integrator, so the only task remaining for the developer is to end the private branch and notify the integrator that the changes on the completed branch are ready for integration.

Windows NT/2000

Use the **Finish Private Branch** wizard in ClearCase HomeBase. Since integration of the changes made onto the integration branch will be done by the integrator, choose to leave the changes on the branch.

Unix

Use the following invocation of **vtsetview** to finish the private branch:

```
vtsetview -endbranch -template tc -brname paulr_timing
```

Integration Process

Each development activity must eventually be merged into the integration branch. ClearCase has several tools available for performing such a merge. The `cleartool findmerge` command can be used to merge all changes from a branch onto another branch. From the integrator view, the following command syntax can be used:

```
cleartool findmerge \dev -all -fversion ../paulr_timing/LATEST  
-merge
```

Alternately, Windows NT and Windows 2000 users can use the ClearCase Merge Manager to perform the same merge.

Both of these methods will merge directory versions and also use Rational Rose RealTime Model Integrator to merge changes in model files. After performing the merge, the integrator should load the model into Rational Rose RealTime and verify that no merge errors have occurred. If the model loads correctly, the changes should be checked in by clicking **Tools > Source Control > Submit All Changes to Source Control**.

The following sequence of steps is quite efficient when integrating a series of development activities:

1. Load the model from the integrator's view.
2. Perform the merge as detailed above.
3. Click **Tools > Source Control > Synchronize Entire Model**. This command reloads all files that changed in the merge.
4. Ensure that the merged differences are as desired.
5. Click **Tools > Source Control > Submit All Changes to Source Control** to accept the changes and check them into source control.
6. Repeat steps 2 through 5 for each activity that requires integration.

Integrating Intermediate Changes

It is quite common that when a developer is working on feature X on branch Y, they may require that intermediate versions of the files modified integrated back to the integration branch. This enables other developers to have access to their changes, but the original developer can continue working on the classes. To accomplish this, the recommendation is as follows:

1. Developer creates a new label.
2. Developer applies the label to the versions of elements on their branch which they want integrated.
3. Developer tells the integrator which label/branch combination specifies the changes to be merged.
4. Integrator uses available CC tools (**findmerge** or merge manager) to perform the integration.

View Template Script Usage

vtadmin

The **vtadmin** script is used to list, create, delete, and update view templates. Each usage of **vtadmin** is detailed below:

```
vtadmin -lstemplates
```

This invocation lists the available view templates.

```
vtadmin -mktemplate -template <templatename>  
  -lateststable <stablelabel> -buildlabel <buildlabel>  
  [-integrationbranch <intbranch>] [-snapshot <vob directory>]
```

This invocation creates a new template with the specified name, latest stable label, build label and integration branch. If the integration branch is not supplied, then /main is assumed.

Note: *Creating a view template does not create the labels and branches indicated; they are assumed to already exist. You can also specify that a load rule be added to the templates so that you can create and use snapshot views.*

```
vtadmin -lslineups -template <templatename>
```

This invocation lists the lineup labels associated with the specified view template.

```
vtadmin -addlineup -template <templatename>
        -lineuplabel <lineuplabel>
```

This invocation adds a lineup label to the specified view template.

```
vtadmin -rmlineup -template <templatename>
        -lineuplabel <lineuplabel>
```

This invocation removes the indicated lineup label from the specified view template.

When invoked with no parameters the script will output usage help.

vtsetview

The `vtsetview` script is used to configure **config** spec and perform common developer queries. Each usage of **vtsetview** is detailed below:

```
vtsetview -startbranch -template <templatename>
        -brname <branchname> -brpoint <labelname>
```

This invocation attempts to start a private branch using the supplied parameters.

```
vtsetview -endbranch -template <templatename>
        -brname <branchname>
```

This invocation is used to end the indicated private branch.

```
vtsetview -setview (integrate | build | browse)
        -template <templatename>
```

This invocation is used to set a specific **config** spec into the current view.

```
vtsetview -lslineups -template <templatename>
```

This invocation lists the available lineups for the specified view template.

When invoked with no parameters, the script will output usage help.



Chapter 11

Customizing Source Control Interface Scripts

Rational Rose RealTime implements source control through a generic script interface that allows it to work with many source control systems. Each Rational Rose RealTime source control action has an associated script that is executed when that action is performed in the Toolset. Rational Rose RealTime looks for the script in the directory selected in Source Control configuration. It executes the script (passing certain information to the script via command-line options), and reads the results from the standard output stream.

A list of scripts follows:

- `cm_getcaps`
- `cm_status`
- `cm_get`
- `cm_add`
- `cm_checkout`
- `cm_checkin`
- `cm_uncheckout`
- `cm_history`
- `cm_extract`
- `cm_label`
- `cm_diff`
- `cm_merge`

Customizing Scripts

Input Parameters

Parameters are categorized as optional or required. This is not an indication of whether the script needs to support the parameter. All parameters must be supported. Optional parameters are those that may be passed by the Toolset in a particular invocation of the script. If the parameter is not passed this indicates that some default behavior is expected. The default behavior is described for each parameter. Required parameters are those that the script can expect the Toolset will always pass to it. There is no default behavior for required parameters, as they will always be present. The detailed information about each parameter is provided below in the section *Script Parameters* on page 191.

Output Expected

All output is to `stdout`.

Output Format

The format of the information in the script output expected by the Toolset.

Script Return Code

All scripts should return either 0 or > 0. If the return code is non-zero, the Toolset interprets this to mean that the operation failed. In this case, the Toolset displays whatever the script writes to `stderr` as an error message.

Notes

Any more detailed notes or warnings.

Note: All scripts must be written in Perl. They will be invoked as

```
rtperl -w <scriptname> <args>
```

Script Parameters

Each of these scripts is passed one or more parameters from Rational Rose RealTime. Values occupy the next argument position, for example, **-T data**.

The following is a description of each of the parameters that can be passed to the various source control scripts. Not every parameter is applicable for every source control script. See the description of each script for the list of parameters that may be passed to that script.

-D <directory>

<directory> is a string containing the path to the directory where the files to be operated on are contained. The default if no **-D** is given is the current working directory.

-E <element>, -S <element>

<element> is a string containing the name of the file to be checked out, unchecked out, submitted, extracted, and so forth by the script (for example, MyCapsule). There is no default. If **-E** is specified, <element> indicates a file; if **-S** is specified, <element> indicates a directory.

For scripts that can operate on multiple elements at the same time, all elements passed to the script will be located in the same directory. This means that all elements specified will be located in the directory specified by **-D**.

-O <file>

<file> is the file name (including path if necessary) where the result of the operation should be written.

-C <commentfile>

<commentfile> is name of a file containing the user supplied reason for the operation. This parameter is only submitted to the script if **cm_getcaps** indicates that the script supports it and the user enters something valid in the dialog displayed when the operation is carried out.

-V <version>

<version> is the version tag of the element to be operated on. If the Toolset asks for a particular version, the script must attempt to return the requested version. If no version is specified (no **-V** is given), the default is the latest.

A Note About Version Tags

For each version in the source control repository there is a unique version tag that the scripts return to the Toolset. Version tags get passed to and from the scripts when the Toolset performs source control operations. The tag may be numerical or an arbitrary string.

cm_getcaps

Returns the set of capabilities supported by the source control system.

Sometimes, it is necessary to specify other information during a particular source control operation. For example, some source control systems must be given an 'update number' when an object is submitted. By simply defining a collection of prompts for a source control operation, the user will be prompted for this additional information and then passed to the corresponding script.

Input parameters

None

Output format

```

<output>::= <cap_entry>*
<cap_entry>::= <capability> '\n'
<capability>::= "Parameter" <operation_param>
                | "Comment" <operation_comment>
                | "Option" <option>

<operation_param>::= <operation_name> PromptString
                    <argument_type> <required> CommandFlag <default>
                    <saveDefault>

<operation_name>::= "Checkout"
                  | "CheckIn"
                  | "Add"
                  | "Get"
                  | "Label"
                  | "UnCheckout"

<argument_type>::= "string"
                  | "integer"
                  | "list" "( " <paramlist> " )"

<paramlist>::= <listelement>
              | <listelement> <paramlist>

<list element>::= "(" value string ")"

<required>::= "mandatory"
            | "optional"

<default>::= Default
           | "NONE"

<saveDefault>::= "saveDefault"
                | "noSaveDefault"

```

```
<operation_comment>::= <operation_name> PromptString <required>  
                        CommandFlag <default> <saveDefault>
```

```
<option>::= "CanDetermineFileVersions" <boolean>  
           | "ScriptTimeout" Integer  
           | "BatchedStatus" <boolean>  
           | "IsClearCase" <boolean>  
           | "SupportsDiff" <boolean>  
           | "SupportsMerge" <boolean>  
           | "SupportsLabel" <boolean>
```

```
<boolean>::= "TRUE"  
            | "FALSE"
```

Notes

- Specifying `Checkout` as the operation means that a new argument is being defined for the `cm_checkout` script. Similarly, `add` defines a new argument for the `cm_add` script.
- `<argument_type>` defines the valid type of the argument. If additional validation is required, it must be done by the script that is given this argument.
- `<required>` defines whether the user must specify a value. If the user does not specify a value for an optional argument, then that argument will not be passed to the corresponding script.
- `<flag>` defines the command line flag to use when this value is passed to the corresponding script.
- `<default>` defines the default value to be displayed in the prompting dialog. A value of `NONE` means that there is no default.
- `<saveDefault>` defines whether the previously entered value is remembered and used as the default the next time this operation is performed.

Example

To specify that the user should be prompted for a mandatory update number and an optional reason for use during check-in, modify the `cm_getcaps` script to produce output similar to the following:

```
parameter checkin "Update No" Integer mandatory -U NONE saveDefault  
Comment checkin "Comment" optional -C NONE saveDefault
```

cm_status

Used to sync the status of files in the Toolset with the state of the files in the source control repository.

Input parameters

-D <directory>
-S <dir element>
-E <file element>

Output format

<item> <status> <user> <version>

See supplied scripts for the allowable values of these fields.

cm_get

Retrieves the latest version of a file from the source control repository.

Input parameters

-D <directory>
-E <filename>
-V <version>

Output format

<item> <status> <user> <version>

See supplied scripts for the allowable values of these fields.

cm_add

Used to add a file into the source control system.

Input parameters

-D <directory>
-S <dir element>
-E <file element>
-C <commentfile>

Output format

<item> <status> <version>

See supplied scripts for the allowable values of these fields.

cm_checkout

Used to lock an element (file) in the source control repository. Upon successful completion, the specified element should be reserved for modification by a particular user.

Input parameters

-D <directory>
-E <filename>
-C <commentfile>

Output format

<item> <status> <version>

See supplied scripts for the allowable values of these fields.

cm_checkin

Used to submit a new version of an element to the source control repository. Upon successful completion, the repository should be updated to include the new version of the specified element.

Input parameters

-D <directory>
-E <filename>
-C <commentfile>

Output format

<item> <status> <version>

See supplied scripts for the allowable values of these fields.

cm_uncheckout

Used to unlock an element in the source control repository. Upon successful completion, the specified element should no longer be reserved for modification by the user.

Input parameters

-D <directory>
-E <filename>

Output format

<item> <status> <version>

See supplied scripts for the allowable values of these fields.

cm_history

Used to produce the list displayed by the History Browser in Rational Rose RealTime. The script should output one line for each version of the specified element.

Input parameters

-D <directory>
-E <filename>

Output format

<version> <author> <date> <time> <locked-by>

See supplied scripts for the allowable values of these fields.

cm_extract

Used to extract a version of an element from the source control repository. Upon successful completion, a copy of the specified element version should be written to a file.

Input parameters

-D <directory>
-E <filename>
-O <output file>
-V <version>

Output format

<item> <status>

See supplied scripts for the allowable values of these fields.

cm_label

Used to apply a label to an element or directory in the source control repository. The labelling operation will not be exposed in Rational Rose RealTime unless the **SupportsLabel** option is set to TRUE in the cm_getcaps file.

Input parameters

-D <directory>
-E <filename>
-S <dir_element>
-L <label>
-C <commentfile>
-V <version>

Output format

<item> <status>

See supplied scripts for the allowable values of these fields.

cm_diff

Used to difference two versions of a file in the repository. This script will only be called for **diffing** operations if the **SupportsDiff** options is set to TRUE in the cm_getcaps file. Only source control systems that can integrate with Rational Rose RealTime Model Integrator should use this mechanism, as a standard textual **diff** does not provide useful results for Rational Rose RealTime model files.

Input parameters

-D <directory>
-E <filename>
-V <version>

Output format

<item> <status>

See supplied scripts for the allowable values of these fields.

cm_merge

Used to merge a version of a file from the repository into the currently checked out file. This script will only be called for merging operations if the **SupportsMerge** options is set to TRUE in the cm_getcaps file. Only source control systems that can integrate with Rational Rose RealTime Model Integrator should use this mechanism, as a standard textual merge does not provide useful results for Rational Rose RealTime model files.

Input parameters

-D <directory>
-E <filename>
-V <version>

Output format

<item> <status>

See supplied scripts for the allowable values of these fields.



Index

A

- access violations 101
- accessing source control operations 85
- adding
 - controlled units 62, 65
 - controlled units to source control 65
 - existing controlled units to models 70
 - files to source control 84
- Apply Label operation 89
- architect role 19
- automated builds 181
- automated scripts for building 133
- automating model validation 137

B

- blue delta 35
- build files 182
- build settings 110
- build views 181
- building
 - components 129
 - creating reusable build artifacts 136
 - reusing build artifacts 136

- using automated scripts 133
- using build artifacts 137
- within a larger build 135

C

- C parameter 191
- changing
 - controlled unit granularity 66
 - granularity of controlled units 60
- Check in operation 88
- Check out operations 88
- checking dependencies 101
- checking in
 - controlled units 126
 - elements 126
 - model elements 126
- checking out files
 - when edited 84
 - with secondary edits 84
- child controlled elements 51
- ClearCase 144
 - activities 149
 - Add 145
 - check in 148
 - check out 148

- command line access 150
 - Deliver 147, 149
 - element type 150
 - entities 176
 - Get 145, 148
 - hijacking 149
 - Label 145
 - Rebase 146
 - Rebase (snapshot) 149
 - recommendations 144
 - repository setup 151
 - snapshot views 147
 - Synchronize 145
 - UCM 38
 - UCM Integration 146
 - Update 149
 - work area setup 152
 - workstation setup 150
- ClearCase options
 - Unix 151
 - Windows 151
- cm_add 197
- cm_checkin 199
- cm_checkout 198
- cm_diff 204
- cm_extract 202
- cm_get 196
- cm_getcaps 193
- cm_history 201
- cm_label 203
- cm_merge 205
- cm_status 195
- cm_uncheckout 200
- code generation performance 61
- command line access to ClearCase 150
- component instances 112
- component packages 97
- composition of a model 98
- configuration management 47, 97, 123, 133
- configuration manager role 21
- configuring for PVCS 161
- configuring source control tools 123
- controllable element 48
 - child 51
 - controlling a subset 65
 - controlling all elements 66
 - directory structure 53
 - exporting from model to a file 68
 - importing from a file to a model 69
 - influence on code generation 61
 - parent 51
- controlled unit 48
 - adding a controlled unit 62
 - adding existing units to models 70
 - changing granularity 66
 - common tasks 65
 - creating sharable units 64
 - granularity 59
 - importing a file 62
 - moving between model directories 67
 - moving elements between 67
 - owned by model 57
 - problems when saving 56
 - reducing number of 54
 - reloading 68
 - saving 56
 - sharing 61
 - sharing a controlled package 62
 - sharing an existing unit into a model 71

- sharing model properties 64
- summary 63
- unresolved references 64
- version identifier 58
- controlled units
 - moving 31
- controlling
 - child element 51
 - element
 - files 50
 - types 50
 - model elements as units 140
 - new child units 58
 - parent element 51
- controlling a subset 65
- converting
 - model 34
- creating
 - build view 181
 - class diagram 102
 - developer view template 178
 - initial lineup 177
 - integrator view 177
 - labels 142
 - lineups 142
 - local work area 140
 - project labels 177
 - reusable build artifacts 136
 - RMF file 158
 - scratch pad package 105
 - work area 140
- creating an rmf file 157
- creating sharable controlled units 64
- cross-references 81
- customer roles 22
- customizing scripts 189

D

- D parameter 191
- decomposing a model into subsystems
 - 100
- default workspace 125, 141
- defining
 - developer work areas 142
 - new path maps 76
 - parameterized path map 77
 - path map 77
 - path map using another symbol 76
 - subsystem interface 104
- defining a virtual path 74
- Deliver 149
- delivering (ClearCase) 147
- deployment packages 97
- developer processes 184
 - creating
 - developer view 184
 - finishing a development activity 185
 - starting a development activity 185
 - working on a development activity
 - 185
- developer role 19
- developer view template 178
- developer work areas 142
- differencing model elements 131
- directory structure for model data 52

E

- E parameter 191
- edit types
 - primary 32
 - secondary 32

editing

- checked out files 84

element type setup

- Unix 151

- Windows 150

elements

- child 51

- controlling 50

- file types 50

- parent 51

enable source control 83

exporting controlled elements 68

external dependencies 102

external dependency 64

F

file based source control 91

file history 90

G

Get operation 88

granularity of controlled units 59

- architecture 59

- code generation performance 61

- implications of changing 60

- modifying

 - elements in same package 60

- number of users 60

- size of model 60

H

hijacking a file 149

I

implicitly defined path map symbols 76

importing

- controllable elements 69

importing a file 62

inject messages 130

input parameters 190

integrating changes 137

integrating immediate changes 187

integration

- promoting changes 132

integration process 186

integrator role 20

interface scripts 85

L

labelling build files 182

labels 142

lineups 142

logical packages 97

M

Make File Writable command 91

Make Files Read Only command 91

making design changes in parallel 174

mapping architecture to subsystems 100

mapping files 156

merging model elements 131

Microsoft Visual SourceSafe 152

- command line access 153

- databases 154

- recommendations for Rational Rose

 - RealTime 153

- repository setup 154
 - setting project mapping 154
 - work area setup 155
 - model 71
 - adding an existing controlled unit 70
 - automating validation 137
 - blue delta 35
 - composition 98
 - controllable element 48
 - controlled unit 48
 - controlled units 48
 - controlling all controllable elements 66
 - controlling elements as units 140
 - converting 34
 - creating a single model 73
 - cross-references 81
 - decomposing into subsystems 100
 - export elements to a file 68
 - granularity of controlled units 60
 - importing controllable elements 69
 - inconsistency 164
 - lineup 124
 - mapping architecture to subsystems 100
 - moving controlled units 67
 - one versus multiple 99
 - opening under source control 125
 - recommended layout 45
 - setting to improve opening time 85
 - splitting 101
 - storing data 48
 - subsystems 98
 - synchronizing 68
 - unique Id 39
 - unit 48
 - unit testing 110
 - unresolved reference 165
 - validating names 169
 - validation scenarios 82
 - Model Conversion 34
 - model data
 - controlled units 48
 - directory structure 52
 - guidelines 48
 - model elements
 - differencing 131
 - merging 131
 - model elements, checking in 126
 - Model Integrator 37
 - model structure 45
 - model validation 163
 - model-relative path names 58
 - models
 - synchronizing 132
 - modifying
 - elements in a package 60
 - moving controlled units 31
 - multiple models 99
- O**
- O parameter 191
 - opening model under source control 125
 - operations
 - Apply Label 89
 - Check in 88
 - check out 88
 - Get 88
 - Refresh status 87
 - Show Differences 89
 - Show History 90

- Submit all Changes 89
- Synchronize 88
- Uncheckout 88
- organizing a model
 - build settings 110
 - component instances 112
 - decomposing a model into sub-systems 100
 - model composition 98
 - one versus multiple 99
 - packages 97
 - processors 111
 - property sets 110
 - splitting 101
 - subsystem 98
 - unit testing 110
 - verifying self-containment 104
- output format 190

P

- package
 - dependencies 101
 - determine external dependencies 102
- packages
 - check out parent 125
 - component 97
 - creating a scratch pad 105
 - deployment 97
 - determine external dependencies 102
 - logical 97
 - parent 125
 - scratch pad 58, 105
 - scratch pad considerations 107
 - Services Library 68
 - show access violations 101

- parallel design changes 174
- parallel development 36
- parallel stream versioning 94
- parent controlled elements 51
- paths
 - defining a parameterized path map 77
 - defining using another path map symbol 76
- primary edit guidelines 34
- primary edits 32, 81
- private component setup 130
- probes 130
- processors
 - project level 111
 - subsystem level 112
- product tester role 20
- project level processors 111
- project manager role 21
- promoting changes 132
- property sets 110
- PVCS 159
 - archive suffixes 160
 - command line access 159
 - database 160
 - locking 161
 - registering a new configuration 161
 - repository setup 160
 - source control operation behavior 159
 - work area setup 161
 - write-protect work files 161

R

- Rational ClearCase Multi-Site 38
- Rational Quality Architect 43

- RCS 155
 - command line access 157
 - creating an rmf file 157
 - repository setup 157
 - setting environment variable 158
 - work area setup 158
- Rebase (snapshot) 149
- rebasing in ClearCase 146
- reducing controlled units 54
- Refresh status operations 87
- refreshing shared unit status 85
- relationships
 - managing between configuration items 12
- reloading controlled units 68
- repository mapping Files 156
- repository setup for ClearCase 151
- reusing build artifacts 136
- reusing build settings 129
- rmf files 156
- roles 18
 - architect 19
 - configuration manager 21
 - customer 22
 - developer 19
 - integrator 20
 - product tester 20
 - project manager 21
 - source control administrator 21
 - team size 18
 - tester 20
- rtwks (workspace) 141
- S**
- S parameter 191
- saving controlled units 56
- SCCS 155
 - command line access 157
 - creating and rmf file 157
 - repository setup 157
 - setting environment variable 158
 - source control operation behavior 157
 - work area setup 159
- scratch pad 58
- scratch pad packages 58, 105
 - considerations 107
- script parameters 191
- script return code 190
- scripts
 - cm_add 197
 - cm_checkin 199
 - cm_checkout 198
 - cm_diff 204
 - cm_extract 202
 - cm_get 196
 - cm_getcaps 193
 - cm_history 201
 - cm_label 203
 - cm_merge 205
 - cm_status 195
 - cm_uncheckout 200
 - creating 190
 - input parameters 190
 - version tags 192
 - vtadmin 187
 - vtsetview 188
 - written using Perl 190
- scripts directory 85
- secondary edits 32, 81
- Services Library packages 68

- Set MSVSS options 154
- setting rmf environment variable 158
- setting up source control 123
- sharing
 - Add 63
 - controlled package 62
 - controlled units 61, 63
 - existing units 71
 - model properties 64
 - Import 63
 - model properties 64
 - Share 63
- sharing and existing controlled unit 71
- sharing controlled units
 - adding 62
 - external dependency 64
 - import 62
 - sharing 62
 - unresolved references 64
- sharing packages
 - using path maps 76
- Show Differences operation 89
- Show History operation 90
- single models 99
- single stream versioning 93
- snapshot views 147
- Source control
 - repository setup for RCS 157
- source control
 - accessing operations 85
 - adding files 84
 - Apply Label 89
 - building a component 129
 - Check in 88
 - Check out 88
 - checking in controlled units 126
 - checking out files
 - automatically 84
 - checking out files automatically 84
 - ClearCase 145
 - command line access to ClearCase 150
 - command line access to PVCS 159
 - command line access to RCS 157
 - command line access to SCCS 157
 - creating a local work area 140
 - customizing 189
 - development concepts 92
 - enabling 83
 - file based 91
 - Get 88
 - interface scripts 85, 189
 - location of interface scripts 85
 - Make Files Read Only 91
 - Make Files Writable 91
 - Microsoft Visual SourceSafe 152
 - operations 87
 - primary edits 81
 - PVCS 159
 - RCS 155
 - Refresh status 87
 - repository 140
 - repository setup for SCCS 157
 - SCCS 155
 - scratch pad packages 106
 - scripts directory 85
 - secondary edits 81
 - settings 83
 - Show Differences 89
 - Show History 90
 - splitting a model 118
 - status options 80

- Submit all Changes 89
- supported systems 85
- Synchronize 88
- types of 91
- Uncheckout 88
- undo a check out 128
- versionable elements 85
- versioning strategies 93
- view based systems 92
- source control administration 139
- source control administrator role 21
- source control development concepts 92
 - development activity 92
 - integration 92
 - lineup 92
 - working in isolation 93
- source control interface scripts 189
 - cm_add 197
 - cm_checkin 199
 - cm_checkout 198
 - cm_diff 204
 - cm_extract 202
 - cm_get 196
 - cm_getcaps 193
 - cm_history 201
 - cm_label 203
 - cm_merge 205
 - cm_status 195
 - cm_uncheckout 200
 - input parameters 190
 - script parameters 191
 - script return code 190
- source control status 80
- source control tools 143
- sources control
 - repository setup for VSS 154
- splitting a model 101, 114
 - in source control 118
 - not in source control 115
 - tasks 115
- stdout output 190
- storing model data
 - controlled units 48
 - granularity of controlled units 59
 - guidelines 48
- Submit all Changes operation 89
- submitting changes to source control
 - source control
 - submit changes 126
- subsystem
 - consumer 113
 - define interface 104
 - verify self-containment 104
- subsystem level processors 112
- subsystems
 - components of 108
 - preparing and releasing 113
 - splitting a model
 - into subsystem models 114
 - supplier 113
- summary
 - sharing controlled units 63
- supported source control systems 85
- synchronize operations 88
- synchronizing
 - models with source control 132
- synchronizing models 68

T

tasks 101

- architect role 19
- component instances 112
- developer role 19, 20
- integrator role 20
- private component setup 131
- source control administrator role 21
- splitting a model 115
- splitting a model in source control 118
- working with controlled units 65

team development

- architect role 19
- configuration manager role 21
- customer role 22
- developer role 19
- heuristics 43
- integrator role 20
- parallel development 36
- product tester role 20
- project manager role 21
- roles 18
- source control administrator role 21
- team size 18
- tester role 20
- typical roles 18

tester roles 20

troubleshooting

- Managing Relationships Between Configuration Items 12
- When more than one user needs to make changes to the same artifact 36

types of source control systems 91

U

UCM 38

UCM integration 146

Uncheckout operations 88

undo a check out 128

Unified Change Management 38

unique id collisions 70

unique Id's 39

unit information 85

unit testing 110, 130

- best practices 130

unresolved model reference 165

updating cross-references 81

using path maps 76

using view templates 175

using virtual paths 77

V

-V parameter 192

validating names 169

verifying dependencies 101

version tags 192

versioning strategies 93

- parallel stream 94

- single stream 93

view based source control system 92

virtual path

- defining 74

- defining a new path map 76

- implicitly defined symbols 76

virtual path map

- symbols 135

virtual path maps 73

vtadmin script 187

vtsetview script 188

W

working in isolation 93

workspace file 125

